Chapter 1 and 7 are licensed under the following terms:
Copyright © 2006-2010 QUALCOMM Incorporated.
Copyright © 2010 Qualcomm Innovation Center, Inc.
Copyright © 2010-2011 Code Aurora Forum.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License,
Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being “GNU General Public
License” and “Funding Free Software”, the Front-Cover Texts being (a) (see below), and with the Back-Cover Texts being (b) (see
below). A copy of the license is included in the section entitled “GNU Free Documentation License”.
(a) The FSF’s Front-Cover Text is:
A GNU Manual
(b) The FSF’s Back-Cover Text is:
You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation
raise funds for GNU development.

Chapters 2-6 and 8 are a Qualcomm-modified version of the document entitled Using the GNU Compiler Collection and are
licensed under the following terms:
Foundation, Inc.
Copyright © 2006-2010 QUALCOMM Incorporated.
Copyright © 2010 Qualcomm Innovation Center, Inc.
Copyright © 2010-2011 Code Aurora Forum.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License,
Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being “GNU General Public
License” and “Funding Free Software”, the Front-Cover Texts being (a) (see below), and with the Back-Cover Texts being (b) (see
below). A copy of the license is included in the section entitled “GNU Free Documentation License”.
(a) The FSF’s Front-Cover Text is:
A GNU Manual
(b) The FSF’s Back-Cover Text is:
You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation
raise funds for GNU development.
Contents

1 Introduction...................................................................................................................... 12
  1.1 Overview ................................................................................................................... 12
  1.2 Features .................................................................................................................... 12
  1.3 Languages ................................................................................................................ 13
  1.4 Runtime systems ...................................................................................................... 13
  1.5 Processor versions ................................................................................................. 13
  1.6 Processor-specific features ...................................................................................... 13
  1.7 Language standards ............................................................................................... 14
  1.8 Using the document ............................................................................................... 16
  1.9 Notation .................................................................................................................... 17
  1.10 Feedback ............................................................................................................... 17

2 Getting Started.............................................................................................................. 18
  2.1 Overview ................................................................................................................... 18
  2.2 Create source file .................................................................................................... 18
  2.3 Compile program ..................................................................................................... 18
  2.4 Execute program ..................................................................................................... 19

3 Using The Compilers.................................................................................................... 20
  3.1 Overview ................................................................................................................... 20
  3.2 Using the compilers ............................................................................................... 21
  3.3 Starting the compilers ............................................................................................ 23
  3.4 Compiler options .................................................................................................... 24
    3.4.1 Display .............................................................................................................. 30
    3.4.2 Compilation ...................................................................................................... 32
    3.4.3 C dialect .......................................................................................................... 34
    3.4.4 C++ dialect .................................................................................................... 38
    3.4.5 Diagnostic message formats ........................................................................... 46
    3.4.6 Warning messages ......................................................................................... 47
    3.4.7 Advisory warning messages ......................................................................... 56
    3.4.8 Debugging ....................................................................................................... 63
    3.4.9 Optimization ................................................................................................. 68
    3.4.10 Specific optimizations .................................................................................. 70
    3.4.11 Math optimizations ...................................................................................... 88
    3.4.12 Preprocessor ............................................................................................... 91
3.4.13 Assembly ................................................................. 101
3.4.14 Linking ................................................................. 102
3.4.15 Directory search .................................................... 106
3.4.16 Processor version .................................................. 108
3.4.17 Target machine .................................................... 108
3.4.18 Code generation .................................................... 109
3.5 Screen messages ......................................................... 117
3.6 Warning and error messages ....................................... 117
3.7 List files ....................................................................... 118
3.8 Spec files ...................................................................... 118
3.9 GCC environment variables .......................................... 126
3.10 Using precompiled headers ........................................ 129
3.11 Using code optimization ............................................ 130
  3.11.1 Basic optimization ............................................... 131
  3.11.2 Global data optimization ....................................... 131
  3.11.3 Function inlining .................................................. 133
  3.11.4 Dead function elimination ..................................... 133
  3.11.5 Unused data removal ............................................. 134
  3.11.6 Fast math ............................................................ 134
3.12 Processor-specific features .......................................... 135

4 Language Implementation ............................................ 136
  4.1 Overview ................................................................. 136
  4.2 Translation ............................................................... 137
  4.3 Environment ............................................................ 137
  4.4 Identifiers ................................................................. 137
  4.5 Characters ............................................................... 138
  4.6 Integers ................................................................. 139
  4.7 Floating point ........................................................... 140
  4.8 Arrays and pointers ................................................ 141
  4.9 Hints ................................................................. 142
  4.10 Structures, unions, enumerations, and bit-fields .......... 142
  4.11 Qualifiers ............................................................... 143
  4.12 Declarators ............................................................. 144
  4.13 Statements ............................................................. 144
  4.14 Preprocessing directives ........................................... 144
  4.15 Library functions .................................................... 145
  4.16 Architecture .......................................................... 145
  4.17 Locale-specific behavior .......................................... 146
5 Language Extensions ................................................................. 147
5.1 Overview ....................................................................................... 147
5.2 General extensions........................................................................ 148
5.2.1 Alternate keywords .................................................................. 148
5.2.2 Dollar signs in identifier names .................................................. 148
5.2.3 <ESC> characters in constants .................................................. 148
5.2.4 Looser rules for escaped newlines ............................................. 149
5.2.5 Statements and declarations in expressions .............................. 149
5.2.6 Locally declared labels ............................................................. 150
5.2.7 Labels as values .......................................................................... 152
5.2.8 Referring to a type with typeof ................................................ 153
5.2.9 Inquiring on alignment of types or variables .............................. 154
5.2.10 Conditionals with omitted operands ........................................ 155
5.2.11 Double-word integers .............................................................. 155
5.2.12 Complex numbers ................................................................. 156
5.2.13 Hex floats ................................................................................. 157
5.2.14 Binary constants ........................................................................ 157
5.2.15 Arrays of length zero ............................................................... 157
5.2.16 Variable-length arrays ............................................................. 159
5.2.17 Macros with a variable number of arguments ......................... 160
5.2.18 Non-Lvalue arrays may have subscripts .................................. 161
5.2.19 Arithmetic on void- and function-pointers ............................... 161
5.2.20 Compound literals ................................................................. 161
5.2.21 Case ranges .............................................................................. 162
5.2.22 Cast to a union type ............................................................... 163
5.2.23 Unnamed struct/union fields within structs/unions .................. 163
5.2.24 Attributes ................................................................................. 164
5.2.25 Function attributes ................................................................. 165
5.2.26 Type attributes ......................................................................... 174
5.2.27 Variable attributes ................................................................. 179
5.2.28 Attribute syntax ........................................................................ 184
5.2.29 Pragmas .................................................................................... 188
5.2.30 Structure-packing pragmas ...................................................... 188
5.2.31 Diagnostic pragmas ................................................................. 189
5.2.32 Visibility pragmas ...................................................................... 190
5.2.33 Push/pop macro pragmas ........................................................ 191
5.2.34 Function-specific option pragmas .......................................... 191
5.2.35 Inline functions ......................................................................... 192
5.2.36 Inline assembly language ....................................................... 194
5.2.37 Constructing function calls ...................................................... 194
5.2.38 Controlling names used in assembly code ............................... 195
5.2.39 Function names as strings ....................................................... 196
5.2.40 Getting the return or frame address of a function .................... 197
5.2.41 Offsetof ......................................................................................................... 198
5.2.42 Offset size checking ..................................................................................... 198
5.2.43 Other built-in functions provided by GCC ................................................... 200
5.2.44 Thread-local storage .................................................................................... 206
  5.2.44.1 ISO/IEC 9899:1999 edits for thread-local storage .............................. 207
  5.2.44.2 ISO/IEC 14882:1998 edits for thread-local storage ............................ 208
5.3 C-specific extensions ........................................................................................ 209
  5.3.1 Nested functions ......................................................................................... 209
  5.3.2 Generalized lvalues .................................................................................... 212
  5.3.3 Structures with no members ....................................................................... 213
  5.3.4 Non-constant initializers ............................................................................. 213
  5.3.5 Designated initializers ................................................................................ 213
  5.3.6 Mixed declarations and code .................................................................... 215
  5.3.7 Prototypes and old-style function definitions .......................................... 215
  5.3.8 C++ style comments ................................................................................... 216
  5.3.9 Incomplete enum types .............................................................................. 216
5.4 C++-specific extensions ..................................................................................... 217
  5.4.1 When is a volatile object accessed? ............................................................. 217
  5.4.2 Restricting pointer aliasing ........................................................................ 218
  5.4.3 Vague linkage .............................................................................................. 219
    5.4.3.1 Inline functions ................................................................................... 219
    5.4.3.2 VTables .............................................................................................. 219
    5.4.3.3 type_info objects .............................................................................. 219
    5.4.3.4 Template instantiations ...................................................................... 219
  5.4.4 #pragma interface and implementation ..................................................... 220
  5.4.5 Where’s the template? ............................................................................... 222
  5.4.6 Extracting the function pointer from a bound pointer to member function... 224
  5.4.7 C++-specific attributes .............................................................................. 225
  5.4.8 Strong using .............................................................................................. 227
  5.4.9 Deprecated features ................................................................................... 227
  5.4.10 Backwards compatibility ......................................................................... 228
5.5 Processor-specific extensions ......................................................................... 229
  5.5.1 Predefined symbols .................................................................................... 229
  5.5.2 Memory access ........................................................................................... 230
  5.5.3 Instruction access ....................................................................................... 231
  5.5.4 Vector access (32-bit) ............................................................................... 231
    5.5.4.1 Construct vector ................................................................................ 233
    5.5.4.2 Extract word ...................................................................................... 233
    5.5.4.3 Extract half-word ............................................................................... 233
    5.5.4.4 Extract byte ....................................................................................... 233
    5.5.4.5 Set word ............................................................................................. 233
    5.5.4.6 Set half-word ..................................................................................... 234
    5.5.4.7 Set byte .............................................................................................. 234
5.5.5 Vector access (32-bit; C++) ............................................................................ 234
  5.5.5.1 Constructors ................................................................. 234
  5.5.5.2 Extract word ...................................................................... 234
  5.5.5.3 Extract half-word ............................................................. 235
  5.5.5.4 Extract byte ....................................................................... 235
  5.5.5.5 Set word ............................................................................ 235
  5.5.5.6 Set half-word .................................................................... 235
  5.5.5.7 Set byte .............................................................................. 235

5.5.6 Vector access (64-bit) ...................................................................................... 236
  5.5.6.1 Construct vector ............................................................... 237
  5.5.6.2 Extract double-word .......................................................... 237
  5.5.6.3 Extract word ...................................................................... 238
  5.5.6.4 Extract half-word ............................................................... 238
  5.5.6.5 Extract byte ....................................................................... 238
  5.5.6.6 Set double-word ............................................................... 239
  5.5.6.7 Set word ............................................................................ 239
  5.5.6.8 Set half-word .................................................................... 239
  5.5.6.9 Set byte .............................................................................. 239

5.5.7 Vector access (64-bit; C++) ............................................................................ 240
  5.5.7.1 Constructors ..................................................................... 240
  5.5.7.2 Extract double-word .......................................................... 240
  5.5.7.3 Extract word ...................................................................... 240
  5.5.7.4 Extract half-word ............................................................... 240
  5.5.7.5 Extract byte ....................................................................... 241
  5.5.7.6 Set double-word ............................................................... 241
  5.5.7.7 Set word ............................................................................ 241
  5.5.7.8 Set half-word .................................................................... 241
  5.5.7.9 Set byte .............................................................................. 242

5.5.8 Circular addressing ......................................................................................... 242
  5.5.8.1 Circular load ...................................................................... 245
  5.5.8.2 Circular store ..................................................................... 247
  5.5.8.3 Circular addressing with variable increment ......................... 249
  5.5.8.4 Data type specification ...................................................... 250
  5.5.8.5 Circular buffer alignment .................................................. 251
  5.5.8.6 Special issues ................................................................. 253

5.5.9 Bit-reversed addressing ................................................................................... 254
  5.5.9.1 Bit-reversed load ............................................................... 257
  5.5.9.2 Bit-reversed store .............................................................. 258
  5.5.9.3 Data type specification ...................................................... 259
  5.5.9.4 Bit-reversed buffer alignment .......................................... 260
  5.5.9.5 Special issues ................................................................. 261
6 Inline Assembly Language .............................................................. 262
   6.1 Overview .................................................................................................................. 262
   6.2 The asm statement .................................................................................................... 263
   6.3 Operands ................................................................................................................... 264
   6.4 Accessed memory ................................................................................................. 266
   6.5 Clobbered registers ............................................................................................... 267
   6.6 Side effects ............................................................................................................... 268
   6.7 Condition codes ........................................................................................................ 268
   6.8 Multiple instructions ............................................................................................... 269
   6.9 Branch instructions ................................................................................................. 270
   6.10 Coding conventions ............................................................................................... 270
   6.11 Assembly size ......................................................................................................... 271
   6.12 Operand constraints ............................................................................................... 272
      6.12.1 Simple constraints ......................................................................................... 272
      6.12.2 Multiple alternative constraints ..................................................................... 274
      6.12.3 Constraint modifier characters ...................................................................... 275
   6.13 Example program ................................................................................................... 276

7 Libraries ............................................................................................ 277
   7.1 Overview .................................................................................................................. 277
   7.2 C/C++ standard libraries .......................................................................................... 278
   7.3 Dynamic loading library ........................................................................................ 278
      7.3.1 Load shared object .......................................................................................... 279
      7.3.2 Unload shared object ....................................................................................... 279
      7.3.3 Find symbol in shared object .......................................................................... 280
      7.3.4 Find symbol by address .................................................................................. 280
      7.3.5 Return error status ........................................................................................... 281
   7.4 Processor-specific libraries ....................................................................................... 282
      7.4.1 Instruction intrinsics (hexagon_protos.h) ...................................................... 282
      7.4.2 Vector intrinsics (hexagon_types.h) ............................................................... 282
      7.4.3 Circular/bitrev intrinsics (hexagon_circ_brev_intrinsics.h) ......................... 283
      7.4.4 Simulation timer (hexagon_sim_timer.h) ...................................................... 283
   7.5 Hexagon intrinsics emulation library ....................................................................... 284
      7.5.1 Application build procedure ........................................................................... 284
      7.5.2 Restrictions ..................................................................................................... 284
8 Binary Compatibility ........................................................................................................ 285

A Acknowledgments ........................................................................................................ 288

B License Statements ..................................................................................................... 301

B.1 History .................................................................................................................. 301
B.2 Funding free software ............................................................................................. 302
B.3 GNU GENERAL PUBLIC LICENSE ........................................................................ 303
   B.3.1 Preamble ....................................................................................................... 303
   B.3.2 How to apply these terms to your new programs ............................................. 308
B.4 GNU FREE DOCUMENTATION LICENSE .......................................................... 309
   B.4.1 ADDENDUM: How to use this license for your documents ......................... 315
# Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3-1</td>
<td>Using the compiler</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5-1</td>
<td>Vector access (32-bit)</td>
<td>232</td>
</tr>
<tr>
<td>Figure 5-2</td>
<td>Vector access (64-bit)</td>
<td>237</td>
</tr>
<tr>
<td>Figure 5-3</td>
<td>Circular addressing program example</td>
<td>243</td>
</tr>
<tr>
<td>Figure 5-4</td>
<td>Circular addressing program output</td>
<td>244</td>
</tr>
<tr>
<td>Figure 5-5</td>
<td>Buffer alignment</td>
<td>252</td>
</tr>
<tr>
<td>Figure 5-6</td>
<td>Invalid circular buffer access</td>
<td>253</td>
</tr>
<tr>
<td>Figure 5-7</td>
<td>Bit-reversed addressing program example</td>
<td>255</td>
</tr>
<tr>
<td>Figure 5-8</td>
<td>Bit-reversed addressing program example</td>
<td>256</td>
</tr>
</tbody>
</table>
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3-1</td>
<td>Compiler file types</td>
<td>22</td>
</tr>
<tr>
<td>Table 3-2</td>
<td>Compiler input and output files</td>
<td>23</td>
</tr>
<tr>
<td>Table 5-1</td>
<td>Enabling function inlining</td>
<td>193</td>
</tr>
<tr>
<td>Table 5-2</td>
<td>Version-specific predefined symbols</td>
<td>229</td>
</tr>
<tr>
<td>Table 5-3</td>
<td>Hexagon processor memory sections</td>
<td>230</td>
</tr>
<tr>
<td>Table 5-4</td>
<td>Circular load intrinsics</td>
<td>250</td>
</tr>
<tr>
<td>Table 5-5</td>
<td>Circular store intrinsics</td>
<td>250</td>
</tr>
<tr>
<td>Table 5-6</td>
<td>Bit-reversed load intrinsics</td>
<td>259</td>
</tr>
<tr>
<td>Table 5-7</td>
<td>Bit-reversed store intrinsics</td>
<td>260</td>
</tr>
<tr>
<td>Table 7-1</td>
<td>Dynamic loading functions</td>
<td>278</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Overview

The C and C++ compilers for the Qualcomm® Hexagon™ processor are based on the GNU C and C++ compilers (which are collectively referred to as GCC).

The GCC compilers work with the Hexagon processor software development tools to provide a complete programming system for developing high-performance software.

The compilers run on the Windows® and Linux® platforms.

NOTE The compilers for the Hexagon processor are based on GNU GCC 4.4.0. For more information see http://gcc.gnu.org.

1.2 Features

The GCC compilers offer the following features:

- **ISO C conformance**
  Supports the International Standards Organization (ISO) C language standard

- **Assembly language support**
  Supports standard GNU-style inline assembly language

- **System library**
  Supports standard libraries for C and C++

- **Processor-specific libraries**
  Provides libraries which support software development for the Hexagon processor

- **Intrinsics**
  Provides library functions which map directly to Hexagon processor instructions for efficient operation (including circular and bit-reversed addressing)
1.3 Languages

The GCC compilers support the GNU versions of the C and C++ programming languages.

The compilers use a common code base, with language-independent code optimizers and a “back end” which generates code for the Hexagon processor. The compilers have separate “front ends” which process their respective languages.

The C compiler is named GCC (though this term is also used as a general term for both compilers). The C++ compiler is named G++.

**NOTE** Historically, many C++ compilers have been implemented as “preprocessors” which emit another high level language such as C. The G++ compiler is not implemented this way; it generates machine code directly. (This sort of preprocessor should not be confused with the C preprocessor, which is an integral feature of the C and C++ languages.)

1.4 Runtime systems

GCC supports the development of both stand-alone programs and RTOS applications.

The build procedure for an RTOS application is a superset of the standard procedure for building a C/C++ application.

**NOTE** For details see the *Hexagon QuRT RTOS Application System Build Guide*.

1.5 Processor versions

GCC supports version V2, V3, and V4 of the Hexagon processor. The command options -mv2, -mv3, and -mv4 specify the Hexagon processor version that the compilers will generate output files for.

For more information on these (and related) command options see Section 3.4.16.

For more information on the Hexagon processor versions see the *Hexagon V4 Programmer’s Reference Manual*.

1.6 Processor-specific features

The GCC compilers are feature-compatible with the standard GNU compilers, except for the following features which are specific to the Hexagon processor:

- The compiler options described in Section 3.12.
- The language extensions described in Section 5.5.
- The libraries described in Section 7.4.
1.7 Language standards

GCC supports three versions of the C standard, although support for the most recent version is not yet complete.

The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990. This standard was ratified as an ISO standard (ISO/IEC 9899:1990) later in 1990. There were no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. This standard, in both its forms, is commonly known as C89, or occasionally as C90, from the dates of ratification. The ANSI standard, but not the ISO standard, also came with a Rationale document.

NOTE To select the C89 standard in GCC, use one of the options -ansi, -std=c89 or -std=iso9899:1990. To obtain all the diagnostics required by the standard, you should also specify -pedantic (or -pedantic-errors to generate errors rather than warnings). See Section 3.4.3.

Errors in the 1990 ISO C standard were corrected in two Technical Corrigenda published in 1994 and 1996. GCC does not support the uncorrected version.

An amendment to the 1990 standard was published in 1995. This amendment added digraphs and __STDC_VERSION__ to the language, but otherwise concerned the library. This amendment is commonly known as AMD1; the amended standard is sometimes known as C94 or C95.

NOTE To select the C94 standard, use the option -std=iso9899:199409 (with, as for other standard versions, -pedantic to receive all required diagnostics).


NOTE To select the C99 standard, use the option -std=c99 or -std=iso9899:1999. (While in development, drafts of this standard version were referred to as “C9X”.)

Errors in the 1999 ISO C standard were corrected in a Technical Corrigendum published in 2001. GCC does not support the uncorrected version.

By default, GCC provides some extensions to the C language that on rare occasions conflict with the C standard (Section 3.4.3). Using any of the C standards described above will disable these extensions where they conflict with the specified standard.
NOTE To explicitly specify an extended version of the C language, use the option
-std=gnu89 (for C89 with GNU extensions) or -std=gnu99 (for C99 with
GNU extensions). The default (i.e., where no C language dialect options are
specified) is -std=gnu89; this will change to -std=gnu99 in some future
release when the C99 support is complete. Some features that are part of the
C99 standard are accepted as extensions in C89 mode.

The ISO C standard defines (in clause 4) two classes of conforming implementation. A
conforming hosted implementation supports the whole standard including all the library
facilities; a conforming freestanding implementation is only required to provide certain
library facilities: those in float.h, limits.h, stdarg.h, and stddef.h; since
AMD1, also those in isof646.h; and in C99, also those in stdbool.h and
stdint.h. In addition, complex types, added in C99, are not required for freestanding
implementations.

The ISO standard also defines two environments for programs, a freestanding
environment, required of all implementations and which may not have library facilities
beyond those required of freestanding implementations, where the handling of program
startup and termination are implementation-defined, and a hosted environment, which is
not required, in which all the library facilities are provided and startup is through a
function int main (void) or int main (int, char *[]). An OS kernel would be a
freestanding environment; a program using the facilities of an operating system would
normally be in a hosted implementation.

GCC aims towards being usable as a conforming freestanding implementation, or as the
compiler for a conforming hosted implementation. By default, it will act as the compiler
for a hosted implementation, defining __STDC_HOSTED__ as 1 and presuming that when
the names of ISO C functions are used, they have the semantics defined in the standard. To
make it act as a conforming freestanding implementation for a freestanding environment,
use the option -ffreestanding; it will then define __STDC_HOSTED__ to 0 and not make
assumptions about the meanings of function names from the standard library, with
exceptions noted below. To build an OS kernel, you may well still need to make your own
arrangements for linking and startup. See Section 3.4.3.

GCC does not provide the library facilities required only of hosted implementations, nor
yet all the facilities required by C99 of freestanding implementations; to use the facilities
of a hosted environment, you will need to find them elsewhere.

Most of the compiler support routines used by GCC are present in libgcc, but there are a
few exceptions. GCC requires the freestanding environment to provide memcpy, memmove,
memset and memcmp. Additionally, if __builtin_trap is used and the target does not
implement the trap pattern, then GCC will emit a call to abort.

For references to Technical Corrigenda, Rationale documents and information concerning
the history of C that is available online, see http://gcc.gnu.org/readings.html.
1.8 Using the document

This document is designed as a reference for experienced C language programmers. It describes the GCC compilers and language implementations.

The document contains eight chapters and two appendices:

- **Chapter 1, Introduction**, presents an overview of the compiler and the document.
- **Chapter 2, Getting Started**, explains how to compile and execute a simple C program.
- **Chapter 3, Using The Compilers**, describes the command line syntax, screen messages, and input and output files.
- **Chapter 4, Language Implementation**, describes how the compilers implement the C language standard.
- **Chapter 5, Language Extensions**, describes extensions to the C language standard.
- **Chapter 6, Inline Assembly Language**, describes how to embed assembly language instructions directly into C programs.
- **Chapter 7, Libraries**, describes the libraries provided with GCC, including the C/C++ standard library and the processor-specific libraries.
- **Chapter 8, Binary Compatibility**, describes the application binary interface (ABI) used by the compilers.
- **Chapter A, Acknowledgments**, lists the people who have contributed to the development of the compilers.
- **Chapter B, License Statements**, lists the license statements for this document.

**C language reference**

This document does not describe the C language. The suggested language reference is:


**Compiler references**

This document does not provide detailed descriptions of the code optimizations performed by GCC. Suggested compiler references are:

- *Advanced Compiler Design and Implementation*, Steven Muchnick.
- *Engineering a Compiler*, Keith Cooper and Linda Torczon.
1.9 Notation

This document uses italics for terms and document names:

_Hexagon V4 Programmer’s Reference Manual_

Courier font is used for computer text:

```c
void main()
{
  printf("Hello world\n");
}
```

The following notation is used to define the syntax of functions and commands:

- Square brackets enclose optional items (e.g., `help [command]`).
- **Bold** is used to indicate literal symbols (e.g., `array[index]`).
- The vertical bar character `|` is used to indicate a choice of items.
- Parentheses are used to enclose a choice of items (e.g., `(on|off)`).
- An ellipsis, `...`, follows items that can appear more than once.
- **Italics** are used for terms that represent categories of symbols.

Examples:

```c
#define name(parameter1[, parameter2...]) definition
logging (on|off)
```

In the above examples `#define` is a preprocessor directive and `logging` is an interactive compiler command.

`name` represents the name of a defined symbol.

`parameter1` and `parameter2` are macro parameters. The second parameter is optional since it is enclosed in square brackets. The ellipsis indicates that the macro accepts more than parameters.

`on` and `off` are bold to show that they are literal symbols. The vertical bar between them shows that they are alternative parameters of the `logging` command.

1.10 Feedback

If you have any comments or suggestions regarding the GCC compilers (or this document), please send them to:

[https://support.cdmatech.com](https://support.cdmatech.com)
2 Getting Started

2.1 Overview

This chapter explains how to compile and execute a simple C program using the GCC compiler.

To simplify this procedure, the program will be built as a stand-alone (non-RTOS) application and executed directly on the Hexagon processor simulator.

NOTE The Hexagon processor software development tools are assumed to be already installed on your computer.

2.2 Create source file

Create the following C source file using a plain-text editor:

```c
#include <stdio.h>

void main()
{
    printf(“Hello world\n”);
}
```

Save the text file as `hello.c`.

2.3 Compile program

To compile the program, type:

```
hexagon-gcc hello.c -o hello.obj
```

This compiles the C source file `hello.c` as a stand-alone application, and generates the processor object file `hello.obj`. 
### 2.4 Execute program

The object file does not require linking because the simulator will directly handle the standard library calls.

To execute the compiled program on the simulator, type:

```plaintext
dotbin-sim hello.obj
```

The program prints its output message in the terminal window, followed by the simulator-generated message “Done!” and some basic execution statistics:

```
hello, world

Done!
T0: Insns=5648 Tcycles=7931
T1: Insns=0 Tcycles=0
T2: Insns=0 Tcycles=0
T3: Insns=0 Tcycles=0
T4: Insns=0 Tcycles=0
T5: Insns=0 Tcycles=0
Total: Insns=5648 Pcycles=47587
Simulator speed=0.314599 Mips
Ratio to Real Time (600 MHz) = 1/226 (elapsed time = 0.017953s)
```

The statistics show the number of instructions and cycles executed on each thread; since the program uses only one thread, the others remain inactive.

You have now compiled and executed a C program using the GCC compiler. For more information on using the compiler see the following chapter.
3 Using The Compilers

3.1 Overview

The GCC compilers translate C and C++ programs into Hexagon processor code. C and C++ programs are stored in source files, which are text files created with a text editor. Hexagon processor code is stored in object files, which are binary files processed by the binary utility programs.

This chapter covers the following topics:

- Using the compilers
- Starting the compilers
- Compiler options
- Screen messages
- Warning and error messages
- List files
- Spec files
- GCC environment variables
- Using precompiled headers
- Using code optimization
- Processor-specific features
3.2 Using the compilers

Figure 3-1 Using the compiler

1. A C or C++ source file is first created with a text editor.

2. The C or C++ compiler translates a C or C++ source file into an output file which (depending on whether the compiler invokes the assembler or linker) may be an assembly source file, object file, or executable file. The compiler optionally inputs header files which define library functions.

3. The assembler translates an assembly language source file into an object file. It optionally produces a list file.

Object files contain the assembled object code, information used by the linker to create executable programs, and (optionally) symbolic information for use by the debugger.

List files list the assembly language source text along with information on how it was translated into object code.

4. The linker links object files into executable programs. It inputs one or more object files and a script file, and outputs an executable file.

Linker script files specify how the input files are to be linked.

5. The debugger simulates the execution of programs. It inputs an executable file and optional script file.

Debugger script files contain debugger commands, and are used to automate debug tasks.

Depending on the command option used (Section 3.4), invoking the compiler can perform up to four stages of processing:

- Preprocessing
- Compilation (proper)
- Assembly
- Linking
GCC is capable of preprocessing and compiling several files either into several assembler input files, or into one assembler input file; then each assembler input file produces an object file, and linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

Table 3-1 lists the compiler file types, including the standard file name extension and the tool that processes the file. The compilers use the file name extension to determine how to process a file.

**Table 3-1  Compiler file types**

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>C source file (must be preprocessed)</td>
<td>C Compiler</td>
</tr>
<tr>
<td>.i</td>
<td>C source file (should not be preprocessed)</td>
<td>C Compiler</td>
</tr>
<tr>
<td>.h</td>
<td>C header file (to be turned into precompiled header)</td>
<td></td>
</tr>
<tr>
<td>.cc .cp</td>
<td>C++ source file (must be preprocessed)</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>.cxx .cpp .CPP .c++ .C</td>
<td>C++ source file (must be preprocessed)</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>.ii</td>
<td>C++ source file (should not be preprocessed)</td>
<td>C++ Compiler</td>
</tr>
<tr>
<td>.h .hh .H</td>
<td>C++ header file (to be turned into precompiled header)</td>
<td></td>
</tr>
<tr>
<td>.s</td>
<td>Assembly source file (should not be preprocessed)</td>
<td>Assembler</td>
</tr>
<tr>
<td>.S</td>
<td>Assembly source file (must be preprocessed)</td>
<td>Assembler</td>
</tr>
<tr>
<td><em>other</em></td>
<td>Binary object file.</td>
<td>Linker</td>
</tr>
</tbody>
</table>

**NOTE** For more information on assembly and object files see the *Hexagon Binutils* document.
3.3 Starting the compilers

To start the C compiler from a command line, type:

```
hexagon-gcc [options...] input_files...
```

To start the C++ compiler from a command line, type:

```
hexagon-g++ [options...] input_files...
```

The compilers accept one or more input files on the command line. Input files can be C/C++ source files or object files. For example:

```
hexagon-gcc test.c driver.c
```

Table 3-2 describes how input files are specified for each output file type.

<table>
<thead>
<tr>
<th>Output File</th>
<th>Default File Name</th>
<th>Input Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable file</td>
<td>a.out</td>
<td>The specified source files are compiled and linked to a single executable file.</td>
</tr>
<tr>
<td>Object file</td>
<td>file.o</td>
<td>Each specified source file is compiled to a separate object file (where file is the source file name).</td>
</tr>
<tr>
<td>Assembly source file</td>
<td>file.s</td>
<td>Each specified source file is compiled to a separate assembly source file (where file is the source file name).</td>
</tr>
<tr>
<td>Preprocessed C/C++ source file</td>
<td>stdout</td>
<td>The preprocessor output is written to the standard output.</td>
</tr>
</tbody>
</table>

Command switches are used to control various compiler options (Section 3.4). A switch consists of a dash character (‘-’) followed by a switch name and optional parameter. Note that switch names are case-sensitive. Switches must be separated by at least one space. For example:

```
hexagon-gcc hello.c -o hello.obj
```

To list the available command options, use the --help option:

```
hexagon-gcc --help
hexagon-g++ --help
```

The compilers display on the console the proper command line syntax, followed by a list of the available command options.

**Command-line defaults**

If you don’t specify an output file name (with the -o option) the compilers name the output file as shown in Table 3-2.
3.4  Compiler options

Compiler options are used to control various compiler features from the command line. For example:

```
hexagon-gcc hello.c -o hello.obj -O2
```

Options have the following properties:

- Options and non-option filename arguments can be mixed on the command line. Options can be specified in any order, except when using multiple instances of a single option; for example, when using multiple instances of `-L`, the directories are searched in the order specified.
- Many options have multi-letter names; therefore, multiple single-letter options cannot be grouped together. For example, `-dr` is not equivalent to `-d -r`.
- Many of the options beginning with `-f`, `-m`, and `-W` are used to control binary settings and thus can be specified in both positive and negative forms. For example, the negative form of `-fasm` is specified as `-fno-asm`. This manual describes only one of the two option forms: namely, the one that specifies the non-default setting.
- Most options can be used with both C and C++; however, some are specific to C, and others to C++. Language-specific options are labelled explicitly in their descriptions; if no language is mentioned, the option can be used with either.

When you invoke the compiler, it normally performs preprocessing, compilation, assembly, and linking (Section 3.3). The compilation options (Section 3.4.2) enable you to stop this process at an intermediate stage. For example, the `-c` option says not to run the linker, which causes the output to consist of object files generated by the assembler.

Other compiler options control a specific stage of processing. For example, some options control the preprocessor, while others control the compiler itself. Yet other options control the assembler and linker: most of these latter options are not described in this manual, since they are rarely used.

The following is a summary of all the options, grouped by type. Explanations are presented in the following sections.

**Display**

See Section 3.4.1

```
--help  --help={class[^]qualifier},...
--target-help  --version
```
Compilation

See Section 3.4.2

- c  -E  -o file  -pass-exit-codes  -pipe
- S  -v  -W[pal],arg[,arg...]  -x language  -###

C dialect

See Section 3.4.3

-ansi  -aux-info  filename  -fno-asm  -fno-built-in
- fno-built-in-function  -fcond-mismatch  -ffreestanding
- fhosted-fms-extensions  -fsigned-bitfields
- fs signed-char  -fun signed-bitfields  -fun signed-char
- no-integrated-cpp  -std=standard
- traditional  -traditional-cpp  -trigraphs
- Wmissing-parameter-type  -Wold-style-declaration
- Wpointer-sign  -Wtraditional-conversion

C++ dialect

See Section 3.4.4

-fabi-version=n  -fno-access-control  -fcheck-new
- fconserve-space  -fno-const-strings  -fno-default-inline
- fno-else-constructors  -fno-enforce-eh-specs
- ffor-scope  -fno-for-scope  -fno-gnu-keywords
- fno-implement-inlines  -fno-implicit-inline-templates
- fno-implicit-templates  -fms-extensions
- fno-nonansi-builtins  -fno-operator-names
- fno-optimal-diags  -fpermissive
- frepo  -fno-rtti  -fstats  -ftemplate-depth=n
- fno-threadsafe-statics  -fvisibility-inlines-hidden
- fuse-cxa-attributes  -fno-weak  -nostdinc+
-Wabi  -Wc++0x-compat  -Wctor-dtor-privacy
-Wno-deprecated  -Weffc++  -Wno-non-template-friend
-Won-virtual-dtor  -Wold-style-cast
-Woverloaded-virtual  -Wno-pmf-conversions
-Wreorder  -Wsign-promo  -Wsynth

Diagnostic message formatting

See Section 3.4.5

-fdiagnostics-show-location=[once|every-line]
-fmessage-length=n

Warning messages

See Section 3.4.6

-fsyntax-only  -pedantic  -pedantic-errors
-w  -Wall  -Waddress  -Warray-bounds  -Wchar-subscripts
-Wcomment  -Werror-implicit-function-declaration
-Wformat  -Wformat=2  -Wno-format-extra-args
-Wformat-nonliteral -Wformat-security -Wformat-y2k
-implicit -Wimplicit-function-declaration -Wimplicit-int
-Wno-import -Winit-self -Wmain
-Wmissing-braces -Wnonnull -Wparentheses
-Wreturn-type -Wsequence-point -Wstrict-aliasing [=2]
-Wswitch -Wswitch-default -Wswitch-enum
-Wtrigraphs -Wuninitialized -Wunknown-pragmas
-Wunused -Wunused-function -Wunused-label
-Wunused-parameter -Wunused-value -Wunused-variable
-Wvolatile-register-var

Advisory warning messages

See Section 3.4.7

-Whave-generate-return -Wbad-function-cast
-Whave-align -Wcast-qual -Wclobbered -Wconversion
-Wdeclaration-after-statement -Wempty-body
-Wframe-large-than=[len]
-Wno-deprecated-declarations -Wdisabled-optimization
-Wno-div-by-zero -Werror -Wextra
-Wfloat-equal -Wignored-qualifiers -Winline
-Wlong-long -Wmissing-declarations
-Wno-invalid-offsetof -Winvalid-pch -Wlarger-than=[len]
-Wpadded
-Wpointer-arith -Wredundant-decls -Wshadow
-Wsign-compare -Wstrict-prototypes -Wstrict-overflow [=level]
-Wsystem-headers -Wtraditional -Wtype-limits
-Wused -Wunreachable-code -Wwrite-strings

Debugging

See Section 3.4.8

-O -O0 -O1 -O2 -O3 -Os

Optimization

See Section 3.4.9
Specific optimizations

See Section 3.4.10

- `-falign-functions=n` -falign-jumps=n
- `-falign-labels=n` -falign-loops=n
- `-fbranch-count-reg` -fbranch-probabilities
- `-fbranch-target-load-optimize`
- `-fbranch-target-load-optimize2` -fcaller-saves
- `-fconserve-stack`
- `-fno-cprop-registers` -fcrossjumping
- `-fcse-follow-jumps` -fcse-skip-blocks
- `-fdelayed-branch` -fdelete-null-pointer-checks
- `-fexpensive-optimizations` -ffloat-store -fforce-addr
- `-fforce-mem` -fno-function-cse -ffunction-sections
- `-fgcse` -fgcse-las -fgcse-lm -fgcse-sm
- `-fif-conversion2` -findirect-inlining
- `-finline-functions` -finline-functions -finline-limit=n
- `-fkeep-inline-functions` -fkeep-static-consts
- `-floop-optimize` -fmerge-all-constants -fmerge-constants
- `-fmodulo-sched` -fold-unroll-all-loops -fold-unroll-loops
- `-fomit-frame-pointer` -foptimize-register-move
- `-fpeel-loops` -fno-peephole
- `-fprofile-generate` -fprofile-use -fprofile-values
- `-fregmove` -frename-registers -freorder-blocks
- `-freorder-functions` -frerun-cse-after-loop
- `-frerun-loop-opt` -fschedule-insns
- `-fschedule-insns2` -fno-sched-interblock
- `-fsched-stalled-insns=n` -sched-stalled-insns-dep=n
- `-fsched2-use-superblocks` -fsched2-use-traces
- `-ftree-switch-conversion` -ftrue-built-in-call-dce -ftracer
- `-funit-at-a-time` -funroll-all-loops -funroll-loops
- `-funsched-loops` -fvpt -fweb -fno-zero-initialized-in-bss
- `-param name=value`

Math optimizations

See Section 3.4.11

- `-fassociative-math` -fcx-limited-range -ffast-math
- `-ffinite-math-only` -fno-math-errno -freciprocal-math
- `-frounding-math` -fsignaling-nans -fno-signed-zeros
- `-fsingle-precision-constant` -fno-trapping-math
- `-funsafe-math-optimizations`
Preprocessor

See Section 3.4.12

-A pred=ans -A -pred=ans -ansi -C -CC
-dCHARS -D name = D name= definition
-fdollars-in-identifiers -fexec-charset= charset
-finput-charset= charset -fpch-deps -fpreprocessed
-fno-show-column -ftabstop= width
-fwide-exec-charset= charset
-fstrict-overflow -fno-strict-overflow
-fworking-directory --help -H -I dir -I-
idirafter dir -imacros file -include file
-iprefix prefix -isystem dir -iwithprefix dir
-iwithprefixbefore dir -M -MD -MF file
-MG -MM -MMD -MP -MQ target -MT target
-nostdinc -nostdinc++ -o file -P
-pedantic -pedantic-errors -remap
-std=standard --target-help -traditional-cpp
-trigraphs -U name -undef -v -version
--version -w -Wall -Wcomment
-Wccomments -Wendif-labels -Werror -Wimport
-Wsystem-headers -Wtraditional
-Wtrigraphs -Wundef -Wunused-macros -x language
-Xpreprocessor option

Assembly

See Section 3.4.13

-Xassembler option

Linking

See Section 3.4.14

object_file_name -c -dynamic -E
-l library -mG0lib -moslib=library
-nodefaultlibs -nostartfiles -nostdlib
-plie -s -S -shared -shared-libgcc
-static -static-libgcc
-symbolic -u symbol -Xlinker option

Directory

See Section 3.4.15

-Bprefix -Idir -I- -Ldir -specs=file
Processor version

See Section 3.4.16

-march=archname  -mcpu=archname
-mv2  -mv3  -mv4

Target machine

See Section 3.4.17

-b machine  -V version

Code generation

See Section 3.4.18

-fargument-alias  -fargument-noalias
-fargument-noalias-global
-fasynchronous-unwind-tables
-fcall-used-reg  -fcall-saved-reg  -fno-common
-fno-exceptions  -ffixed-reg  -fno-ident
-finhibit-size-directive  -finstrument-functions
-fleading-underscore  -fnon-call-exceptions
-fpack-struct=n  -fpcc-struct-return
-fpic  -fPIC  -fpie  -fPIE  -freg-struct-return
-fshared-data  -fno-short-nums  -fshort-double
-fshort-wchar  -fstack-check
-fstack-limit-register=reg
-fstack-limit-symbol=sym
-fno-stack-limit  -ftls-model=model  -ftrapv
-funwind-tables  -fverbose-asm  -fwrapv
-fvisibility=[default|internal|hidden|protected]
-G size
-mno-dot-new
-mno-falign
-mno-hardware-loops
-mliteral-intrinsics
-mliteral-pool
-mliteral-pool-addresses
-mv1-mv2-uncached-data
3.4.1 Display

The following options write compiler-related information to the standard output.

--help
Display description of the command line options understood by GCC. If the -v option is also specified then --help will also be passed on to the various processes invoked by GCC, so they can display the command line options that they accept. If the -Wextra option is also specified then the command line options that have no documentation associated with them will also be displayed.

--help=[class[^]qualifier][,...]
Display description of the command line options based on the specified class and qualifier. This enables users to display only the command line options of interest.

class can be one of the following values:

optimizers
Display all code optimization options.

warnings
Display all warning message options.

target
Display target-specific compiler options.

params
Display values recognized by --param.

language
Display the options supported for C and C++.

common
Display language-independent options.

qualifier can be one of the following values:

undocumented
Display only the undocumented options.

joined
Display options that accept arguments preceded by =.

separate
Display options that accept arguments preceded by whitespace.
For example, the following option displays all the undocumented target-specific switches:

```
--help=target,undocumented
```

A qualifier can be logically inverted by prefixing it with the caret (^). For example, the following option displays all undocumented binary warning options (i.e., options that are either on or off, and do not take an argument):

```
--help=warnings,^joined,^undocumented
```

**NOTE**  
--help= arguments must not consist solely of inverted qualifiers.

Classes can be combined; however, this is generally too restrictive to be useful. It is useful, however, when one of the classes is target. For example, the following option displays all the target-specific optimization options:

```
--help=target,optimizers
```

--help= can be specified more than once on a command line. Each successive use displays the specified class of options, while skipping the ones already displayed.

If --help= is preceded on a command line by -Q, the descriptive text changes to show whether an option is enabled, disabled, or set to a specific value (assuming the compiler knows this information at the time).

If --help= is preceded on a command line by a specific command line option, it displays only the options that are enabled by the specified option. For example, the following option sequence displays only the optimization options enabled by -O2:

```
-Q -O2 --help=optimizers
```

Similarly, here is a procedure for determining which binary optimizations are enabled by -O3:

```
gcc -c -Q -O3 --help=optimizers > /tmp/O3-opts
gcc -c -Q -O2 --help=optimizers > /tmp/O2-opts
diff /tmp/O2-opts /tmp/O3-opts | grep enabled
```

**--target-help**  
Display a description of the target-specific command line options for each tool.

**--version**  
Display version number and copyright message of the invoked compiler.
### 3.4.2 Compilation

To perform only some of the stages of compilation, you can use `-x` (or filename suffixes) to inform GCC where to start, and `-c`, `-S`, or `-E` to indicate where to stop. Note that some combinations (e.g., `-x cpp-output -E`) instruct GCC to do nothing at all.

**NOTE** The `-save-temps` option (Section 3.4.8) can be used to save the intermediate files (.i, .s) generated by the various compilation stages.

- `-c`
  Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

  By default, the object file name for a source file is made by replacing the suffix .c, .i, .s, etc., with .o.

  Unrecognized input files, not requiring compilation or assembly, are ignored.

- `-E`
  Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

  Input files which don’t require preprocessing are ignored.

- `-o file`
  Place output in file `file`. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

  If you specify `-o` when compiling more than one input file, or you are producing an executable file as output, all the source files on the command line will be compiled at once.

  If `-o` is not specified, the default is to put an executable file in `a.out`, the object file for `source.suffix` in `source.o`, its assembler file in `source.s`, and all preprocessed C source on standard output.

- `-pass-exit-codes`
  Normally the compiler will exit with the code of 1 if any phase of the compilation returns a non-success return code. If you specify `-pass-exit-codes`, the compiler will instead return with the numerically highest error produced by any phase that returned an error indication.

- `-pipe`
  Use pipes rather than temporary files for communication between the various stages of compilation.

- `-S`
  Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

  By default, the assembler file name for a source file is made by replacing the suffix .c, .i, etc., with .s.

  Input files that don’t require compilation are ignored.
-v
Print (on standard error output) the commands executed to run the stages of
compilation. Also print the version number of the compiler driver program and of
the preprocessor and the compiler proper.

-W[pal],arg[,arg...]  
Pass the specified arguments directly to the specified stage of the compiler (as
described below).

The specified arguments can be passed to multiple compiler stages by combining
the stage option letters. For example, -Wpa,foo passes the argument foo to both
the preprocessor and the assembler.

NOTE  The CPP interface is undocumented and subject to change, so whenever
possible avoid using -Wp and let the compiler driver handle the options.

-Wp
Pass arguments to preprocessor. See -v above for details.

-Wa
Pass arguments to assembler.

-Wl
Pass arguments to linker.

-x  language
Specify explicitly the input language language for the following input files
(rather than letting the compiler choose a default based on the file name suffix). 
This option applies to all following input files until the next -x option.

The language parameter has the following possible values:
  c
  c-header
  cpp-output
  c++
  c++-header
  c++-cpp-output
  assembler
  assembler-with-cpp

-x none
Turn off any specification of a language, so that subsequent files are handled
according to their file name suffixes (as they are if -x has not been used at all).

-###
Like -v except the commands are not executed and all command arguments are
quoted. This is useful for shell scripts to capture the driver-generated command
lines.
3.4.3 C dialect

The following options control the dialect of C (or languages derived from C, such as C++) that the compiler accepts:

-ansi

In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions that conflict with ISO C++.

This turns off certain features of GCC that are incompatible with ISO C90 (when compiling C code), or of standard C++ (when compiling C++ code), such as the \texttt{asm} and \texttt{typeof} keywords, and predefined macros such as \texttt{unix} that identify the type of system you are using. It also enables the undesirable and rarely used ISO trigraph feature. For the C compiler, it disables recognition of C++ style // comments as well as the \texttt{inline} keyword.

The alternate keywords \texttt{__asm__}, \texttt{__extension__}, \texttt{__inline__} and \texttt{__typeof__} continue to work despite -ansi. You would not want to use them in an ISO C program, of course, but it is useful to put them in header files that might be included in compilations done with -ansi. Alternate predefined macros such as \texttt{__unix__} are also available, with or without -ansi.

The -ansi option does not cause non-ISO programs to be rejected gratuitously. For that, -pedantic is required in addition to -ansi. See Section 3.4.3.

The macro \texttt{__STRICT_ANSI__} is predefined when the -ansi option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ISO standard doesn’t call for; this is to avoid interfering with any programs that might use these names for other things.

Functions which would normally be built-in but do not have semantics defined by ISO C (such as \texttt{alloca} and \texttt{ffs}) are not built-in functions when -ansi is used. See Section 5.2.43 for details of the functions affected.

-aux-info filename

Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C.

Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (I, N for new or o for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (C or F, respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.
-fno-asm
Do not recognize \texttt{asm}, \texttt{inline} or \texttt{typeof} as a keyword, so that code can use these words as identifiers. You can use the keywords \texttt{__asm__}, \texttt{__inline__} and \texttt{__typeof__} instead. -ansi implies -fno-asm.

In C++, this switch only affects the \texttt{typeof} keyword, since \texttt{asm} and \texttt{inline} are standard keywords. You may want to use the -fno-gnu-keywords flag instead, which has the same effect. In C99 mode (-std=c99 or -std=gnu99), this switch only affects the \texttt{asm} and \texttt{typeof} keywords, since \texttt{inline} is a standard keyword in ISO C99.

-ffreestanding
Assert that compilation takes place in a freestanding environment. This implies -fno-builtin. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at \texttt{main}. The most obvious example is an OS kernel. This is equivalent to -fno-hosted.

See Section 1.7 for details of freestanding and hosted environments.

-fhosted
Assert that compilation takes place in a hosted environment. This implies -fbuiltin. A hosted environment is one in which the entire standard library is available, and in which \texttt{main} has a return type of \texttt{int}. Examples are nearly everything except a kernel. This is equivalent to -fno-freestanding.
-fms-extensions
Accept some non-standard constructs used in Microsoft header files.

-fsigned-bitfields
Define bit-fields as signed.

By default the Hexagon processor defines bit-fields as unsigned (Section 4.10).

If a program depends on the signedness of a bit-field value without explicitly specifying the bit-field type as signed or unsigned, then the program is nonportable because it depends on the machine it was written for. This option (and its inverse) enable such programs to work with the opposite default.

Note that this option does not affect bit-fields explicitly declared as signed or unsigned.

-fsigned-char
Define type char as signed char.

By default the Hexagon processor defines char as unsigned char (Section 4.5).

If a program depends on the signedness of a character value while only specifying the type as plain char, then the program is nonportable because it depends on the machine it was written for. This option (and its inverse) enable such programs to work with the opposite default.

Note that this option does not affect types explicitly declared as signed char or unsigned char, even though char is always functionally equivalent to one of these types.

-funsigned-bitfields
Define bit-fields as unsigned.

Note that this option is equivalent to -fno-signed-bitfields, the negative form of -fsigned-bitfields. Similarly, the option -fno-unsigned-bitfields is equivalent to -fsigned-bitfields.

-funsigned-char
Define type char as unsigned char.

Note that this option is equivalent to -fno-signed-char, the negative form of -fsigned-char. Similarly, the option -fno-unsigned-char is equivalent to -fsigned-char.

-no-integrated-cpp
Performs a compilation in two passes: preprocessing and compiling. This option allows a user supplied ccl, cclplus, or cclobj via the -B option. The user supplied compilation step can then add in an additional preprocessing step after normal preprocessing but before compiling. The default is to use the integrated cpp (internal cpp)

The semantics of this option will change if ccl, cclplus, and cclobj are merged.
-std=standard
Determine the language standard. This option is currently only supported when compiling C or C++. A value for this option must be provided.

The standard parameter has the following possible values:

  c89
  iso9899:1990
  ISO C90 (same as -ansi).
  iso9899:199409
  ISO C90 as modified in amendment 1.
  c99
  c9x
  iso9899:1999
  iso9899:199x
  ISO C99. Note that this standard is not yet fully supported; see http://gcc.gnu.org/gcc-3.4/c99status.html for more information. The names c9x and iso9899:199x are deprecated.
  gnu89
  Default, ISO C90 plus GNU extensions (including some C99 features).
  gnu99
  gnu9x
  ISO C99 plus GNU extensions. When ISO C99 is fully implemented in GCC, this will become the default. The name gnu9x is deprecated.
  c++98
  The 1998 ISO C++ standard plus amendments.
  gnu++98
  The same as -std=c++98 plus GNU extensions. This is the default for C++ code.

Even when this option is not specified, you can still use some of the features of newer standards in so far as they do not conflict with previous C standards. For example, you may use __restrict__ even when -std=c99 is not specified.

The -std options specifying some version of ISO C have the same effects as -ansi, except that features that were not in ISO C90 but are in the specified version (for example, // comments and the inline keyword in ISO C99) are not disabled.

See Section 1.7 for details of these standard versions.

-traditional
-traditional-cpp
Formerly, these options caused GCC to attempt to emulate a pre-standard C compiler. They are now only supported with the -E switch. The preprocessor continues to support a pre-standard mode. See the CPP manual for details.
-trigraphs

-Wmissing-parameter-type
Warn if a parameter is declared without a type specifier in K&R-style functions:
void foo(bar) { }
This option is enabled by -Wextra.

-Wold-style-declaration
Warn for obsolete C declaration syntax. For example, warn if storage class specifiers such as static are not the first items in a declaration.
This option is enabled by -Wextra.

-Wtraditional-conversion
Warn if the presence of a prototype alters a type conversion. This includes conversions between fixed point and floating, and conversions changing the width or sign of a fixed point argument except when the same as the default promotion.

3.4.4 C++ dialect
This section describes the command-line options that are only meaningful for C++ programs; but you can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file firstClass.C like this:

```
hexagon-cpp -g -frepo -O -c firstClass.C
```
In this example, only -frepo is an option meant only for C++ programs; you can use the other options with any language supported by GCC.

Here is a list of options that are only for compiling C++ programs:

-fabi-version=n
Use version n of the C++ ABI. Version 2 is the version of the C++ ABI that first appeared in G++ 3.4. Version 1 is the version of the C++ ABI that first appeared in G++ 3.2. Version 0 will always be the version that conforms most closely to the C++ ABI specification. Therefore, the ABI obtained using version 0 will change as ABI bugs are fixed.

The default version is 2.

-fno-access-control
Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.
-fcheck-new
Check that the pointer returned by operator new is non-null before attempting to modify the storage allocated. This check is normally unnecessary because the C++ standard specifies that operator new will only return 0 if it is declared throw(), in which case the compiler will always check the return value even without this option. In all other cases, when operator new has a non-empty exception specification, memory exhaustion is signalled by throwing std::bad_alloc. See also new (nothrow).

-fconserve-space
Put uninitialized or runtime-initialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. If you compile with this flag and your program mysteriously crashes after main() has completed, you may have an object that is being destroyed twice because two definitions were merged.

This option is no longer useful on most targets, now that support has been added for putting variables into BSS without making them common.

-fno-const-strings
Give string constants type char * instead of type const char *. By default, GCC C++ uses type const char * as required by the standard. Even if you use -fno-const-strings, you cannot actually modify the value of a string constant.

This option might be removed in a future release of GCC C++. For maximum portability, you should structure your code so that it works with string constants that have type const char *.

-fno-default-inline (C++ only)
Do not assume inline for functions defined inside a class scope. (This option is also listed in Section 3.4.9 as an optimization option.) Note that such functions will have linkage like inline functions; they just won’t be inlined by default.

-fno-elide-constructors
The C++ standard allows an implementation to omit creating a temporary which is only used to initialize another object of the same type. Specifying this option disables that optimization, and forces GCC C++ to call the copy constructor in all cases.

-fno-enforce-eh-specs
Don’t check for violation of exception specifications at runtime. This option violates the C++ standard, but may be useful for reducing code size in production builds, much like defining NDEBUG. The compiler will still optimize based on the exception specifications.
-ffor-scope
-ffor-scope
If -ffor-scope is specified, the scope of variables declared in a for-init-
statement is limited to the for loop itself, as specified by the C++ standard. If -
ffor-scope is specified, the scope of variables declared in a for-init-
statement extends to the end of the enclosing scope, as was the case in old
versions of GCC C++, and other (traditional) implementations of C++.

The default if neither flag is given to follow the standard, but to allow and give a
warning for old-style code that would otherwise be invalid, or have different
behavior.

-fno-gnu-keywords
Do not recognize typeof as a keyword, so that code can use this word as an
identifier. You can use the keyword __typeof__ instead. -ansi implies
-fno-gnu-keywords.

-fno-implement-inlines
To save space, do not emit out-of-line copies of inline functions controlled by
#pragma implementation. This will cause linker errors if these functions are
not inlined everywhere they are called.

-fno-implicit-inline-templates
Don’t emit code for implicit instantiations of inline templates, either. The default
is to handle inlines differently so that compiles with and without optimization will
need the same set of explicit instantiations.

-fno-implicit-templates
Never emit code for non-inline templates which are instantiated implicitly (i.e. by
use); only emit code for explicit instantiations. See Section 5.4.5 for more
information.

-fms-extensions
Disable pedantic warnings about constructs used in MFC, such as implicit int and
getting a pointer to member function via non-standard syntax.

-fno-nonansi-builtins
Disable built-in declarations of functions that are not mandated by ANSI/ISO C.
These include ffs, alloca, __exit, index, bzero, conjf, and other related
functions.

-fno-operator-names
Do not define the symbols and, bitand, bitor, compl, not, or, and xor as
keyword synonyms for the corresponding operator symbols.

-fno-optional-diags
Disable diagnostics that the standard says a compiler does not need to issue.
Currently, the only such diagnostic issued by GCC C++ is the one for a name
having multiple meanings within a class.

-fpermissive
Downgrade some diagnostics about nonconformant code from errors to warnings.
Thus, using -fpermissive will allow some nonconforming code to compile.

-frepo
Enable automatic template instantiation at link time. This option also implies
-fno-implicit-templates. See Section 5.4.5 for more information.
-fno-rtti
Disable generation of information about every class with virtual functions for use by the C++ runtime type identification features (dynamic_cast and typeid). If you don’t use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed.

-fstats
Emit statistics about front-end processing at the end of the compilation. This information is generally only useful to GCC C++ development team.

-ftime-template-depth-n
Set the maximum instantiation depth for template classes to n. A limit on the template instantiation depth is needed to detect endless recursions during template class instantiation. ANSI/ISO C++ conforming programs must not rely on a maximum depth greater than 17.

-fno-threadsafe-statics
Do not generate the extra code necessary for ensuring thread-safe initialization of local static variables. This results in a slight reduction in code size for code that does not need to be thread-safe.

By default the compiler generates code which uses the routines specified in the C++ ABI for thread-safe initialization of function-scope static variables.

-fuse-cxa-atexit
Register destructors for objects with static storage duration with the __cxa_atexit function rather than the atexit function. This option is required for fully standards-compliant handling of static destructors, but will only work if your C library supports __cxa_atexit.

-fno-weak
Do not use weak symbol support, even if it is provided by the linker. By default, GCC C++ will use weak symbols if they are available. This option exists only for testing, and should not be used by end-users; it will result in inferior code and has no benefits. This option may be removed in a future release of GCC C++. 
-fvisibility-inlines-hidden

Using this option prevents pointers from being compared to inline methods when the addresses of the two functions were taken in different shared objects.

The benefit of using this option is that the compiler can mark inline methods to have hidden visibility (Section 5.4.7); as a result, they do not appear in the export table of a dynamic shared object (DSO), and do not require a PLT indirection when used within the DSO. Enabling this option can significantly reduce the load and link times of a DSO when it makes heavy use of templates.

Using his option differs slightly from explicitly marking methods as hidden: it does not affect local static variables in a function, and does not cause the compiler to assume that the function is defined in only one shared object.

This option can be selectively negated by explicitly setting the visibility of specific methods. For example, to compare pointers to a particular inline method, mark the method as having default visibility. (Note that marking the enclosing class with default visibility has no effect.)

Explicitly instantiated inline methods are unaffected by this option as their linkage might otherwise cross a shared library boundary (Section 5.4.3.4).

-nostdinc++

Do not search for header files in the standard directories specific to C++, but do still search the other standard directories. (This option is used when building the C++ library.)

-Wabi (C++ only)

Warn when GCC C++ generates code that is probably not compatible with the vendor-neutral C++ ABI. Although an effort has been made to warn about all such cases, there are probably some cases that are not warned about, even though GCC C++ is generating incompatible code. There may also be cases where warnings are emitted even though the code that is generated will be compatible.

You should rewrite your code to avoid these warnings if you are concerned about the fact that code generated by GCC C++ may not be binary compatible with code generated by other compilers.

The known incompatibilities at this point include:

Incorrect handling of tail-padding for bit-fields. GCC C++ may attempt to pack data into the same byte as a base class. For example:

```c
struct A { virtual void f(); int f1 : 1; }
struct B : public A { int f2 : 1; }
```

In this case GCC C++ will place B::f2 into the same byte as A::f1; other compilers will not. You can avoid this problem by explicitly padding A so its size is an integral multiple of the byte size on your platform; that will cause GCC C++ to lay out B identically.
Incorrect handling of tail-padding for virtual bases. GCC C++ does not use tail padding when laying out virtual bases. For example:

```c
struct A { virtual void f(); char c1; };
struct B { B(); char c2; };
struct C : public A, public virtual B {};
```

In this case, GCC C++ will not place B into the tail-padding for A; other compilers will. You can avoid this problem by explicitly padding A so that its size is a multiple of its alignment (ignoring virtual base classes); that will cause GCC C++ to layout C identically.

Incorrect handling of bit-fields with declared widths greater than that of their underlying types, when the bit-fields appear in a union. For example:

```c
union U { int i : 4096; };
Assuming that an int does not have 4096 bits, GCC C++ will make the union too small by the number of bits in an int.
```

Empty classes can be placed at incorrect offsets. For example:

```c
struct A {};
struct B {
    A a;
    virtual void f();
};
struct C : public B, public A {};
```

GCC C++ will place the A base class of C at a nonzero offset; it should be placed at offset zero. GCC C++ mistakenly believes that the A data member of B is already at offset zero.

Names of template functions whose types involve typename or template template parameters can be mangled incorrectly.

```c
template <typename Q>
void f(typename Q::X) {}

template <template <typename> class Q>
void f<typename Q<int>::X> {}  
Instantiations of these templates may be mangled incorrectly.
```

-Wc++0x-compat (C++ only)
Warn about C++ constructs whose meaning differs between ISO C++ 1998 and ISO C++ 200x; for example, identifiers in ISO C++ 1998 that become keywords in ISO C++ 200x. This warning is enabled by -Wall.

-Wctor-dtor-privacy (C++ only)
Warn when a class seems unusable because all the constructors or destructors in that class are private, and it has neither friends nor public static member functions.

-Wno-deprecated (C++ only)
Do not warn about usage of deprecated features. See Section 5.4.9.
-Wefc++ (C++ only)
Warn about violations of the following style guidelines from Scott Meyers’ Effective C++ book:
- Item 11: Define a copy constructor and an assignment operator for classes with dynamically allocated memory.
- Item 12: Prefer initialization to assignment in constructors.
- Item 14: Make destructors virtual in base classes.
- Item 15: Have `operator=` return a reference to `*this`.
- Item 23: Don’t try to return a reference when you must return an object.

Also warn about violations of the following style guidelines from Scott Meyers’ More Effective C++ book:
- Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.
- Item 7: Never overload `&&`, `||`, or `.`

When selecting this option, be aware that the standard library headers do not obey all of these guidelines; use `grep -v` to filter out those warnings.

-Wno-non-template-friend (C++ only)
Disable warnings when non-templatized friend functions are declared within a template. Since the advent of explicit template specification support in GCC C++, if the name of the friend is an unqualified-id (i.e., `friend foo(int)`), the C++ language specification demands that the friend declare or define an ordinary, non-template function. (Section 14.5.3). Before GCC C++ implemented explicit specification, unqualified-ids could be interpreted as a particular specialization of a templatized function. Because this non-conforming behavior is no longer the default behavior for GCC C++, `-Wnon-template-friend` allows the compiler to check existing code for potential trouble spots and is on by default. This new compiler behavior can be turned off with `-Wno-non-template-friend` which keeps the conformant compiler code but disables the helpful warning.

-Wnon-virtual-dtor (C++ only)
Warn when a class appears to be polymorphic, thereby requiring a virtual destructor, yet it declares a non-virtual one. This warning is enabled by `-Wall`.

-Wold-style-cast (C++ only)
Warn if an old-style (C-style) cast to a non-void type is used within a C++ program. The new-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are less vulnerable to unintended effects and much easier to search for.
-Woverloaded-virtual (C++ only)
Warn when a function declaration hides virtual functions from a base class. For example, in:

```c
struct A {
    virtual void f();
};

struct B: public A {
    void f(int);
};
```

the A class version of f is hidden in B, and code like:

```c
B* b;
b->f();
```

will fail to compile.

-Wno-pmf-conversions (C++ only)
Disable the diagnostic for converting a bound pointer to member function to a plain pointer.

-Wreorder (C++ only)
Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

```c
struct A {
    int i;
    int j;
    A(): j (0), i (1) { }
};
```

The compiler will rearrange the member initializers for i and j to match the declaration order of the members, emitting a warning to that effect. This warning is enabled by -Wall.

The following -W options are not affected by -Wall.

-Wsign-promo (C++ only)
Warn when overload resolution chooses a promotion from unsigned or enumerated type to a signed type, over a conversion to an unsigned type of the same size. Previous versions of GCC C++ would try to preserve unsignedness, but the standard mandates the current behavior.
-Wsynth (C++ only)
Warn when GCC C++’s synthesis behavior does not match that of cfront. For instance:

```c
struct A {
    operator int ()
    A& operator = (int);
};
```

```c
main ()
{
    A a,b;
    a = b;
}
```

In this example, C++ will synthesize a default `A& operator = (const A&);`, while cfront will use the user-defined `operator =`.

### 3.4.5 Diagnostic message formats

Diagnostic messages have traditionally been formatted independently of the output device properties (e.g., display width). The options described below control how diagnostic messages are generated: for example, how many characters per line, or how often source location information should be reported.

- **-fdiagnostics-show-location=every-line (C++ only)**
  Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit the same source location information (as prefix) for physical lines that result from the process of breaking a message which is too long to fit on a single line.

- **-fdiagnostics-show-location=once (C++ only)**
  Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit only once the source location information; that is, in case the message is too long to fit on a single physical line and has to be wrapped, the source location won’t be emitted (as prefix) again, over and over, in subsequent continuation lines. This is the default behavior.

- **-fmessage-length=n (C++ only)**
  Try to format error messages so that they fit on lines of about `n` characters. The default is 72 characters for GCC. If `n` is zero, then no line-wrapping will be done; each error message will appear on a single line.
### 3.4.6 Warning messages

Warnings are diagnostic messages which identify programming language constructions that while not incorrect may be risky to use, or suggest that a coding error may exist.

You can request many specific warnings by using the corresponding warning options (which conventionally start with -W). For example, -Wimplicit requests that warnings be displayed for implicit variable declarations.

GCC supports two types of warnings:

- **Regular warnings** flag language constructions that some users consider questionable, and that are easy to avoid (or to modify to eliminate the warning) even in conjunction with macros. All regular warnings are enabled with the -Wall option.

- **Advisory warnings** flag language constructions that are necessary or in some cases hard to avoid, and that are difficult to avoid by modifying the code.

This section describes the options for displaying regular warning messages only; the options for advisory warnings are described in Section 3.4.7.

- **-fsyntax-only**
  Check the code for syntax errors, and exit.

- **-pedantic**
  Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any -std option used.

Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few will require -ansi or a -std option specifying the required version of ISO C). However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are rejected.

- **-pedantic** does not cause warning messages for use of the alternate keywords whose names begin and end with __. Pedantic warnings are also disabled in the expression that follows __extension__. However, only system header files should use these escape routes; application programs should avoid them. See Section 5.2.1.

Some users try to use -pedantic to check programs for strict ISO C conformance. They soon find that it does not do quite what they want: it finds some non-ISO practices, but not all: only those for which ISO C requires a diagnostic, and some others for which diagnostics have been added.

A feature to report any failure to conform to ISO C might be useful in some instances, but would require considerable additional work and would be quite different from -pedantic. There are no plans to support such a feature in the near future.
Hexagon GNU C/C++ Compiler

Where the standard specified with `--std` represents a GNU extended dialect of C, such as `gnu89` or `gnu99`, there is a corresponding *base standard*, the version of ISO C on which the GNU extended dialect is based. Warnings from `--pedantic` are given where they are required by the base standard. (It would not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

`--pedantic-errors`
Like `--pedantic`, except that errors are produced rather than warnings.

`--w`
Inhibit all warning messages.

`--Wall`
Enable all of the regular `--w` options (Section 3.4.6) at once. Also enable some of the language-specific warnings described in Section 3.4.4.

Note that the advisory `--w` options (Section 3.4.7) are not specified by `--Wall`.

`--Waddress`
Warn about the questionable use of memory addresses, including:

- Using a function address in a conditional expression. For example:
  ```c
  void func(void); if (func)
  ```

- Comparing to the memory address of a string literal. For example:
  ```c
  if (x == "abc").
  ```

Such uses typically indicate a program error. A function address always evaluates to `true`, so its use in a conditional suggests a mistyped function call. Similarly, comparing a value to a string literal results in undefined behavior and is nonportable in C, suggesting the programmer intended to use `strcmp`.

`--Warray-bounds`
Warn if array subscripts are always out of bounds. This warning is active only when `--ftree-vrp` is active (default for `-O2` and above).

`--Wchar-subscripts`
Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

`--Wcomment`
Warn whenever a comment-start sequence `/*` appears in a `/*` comment, or whenever a backslash-newline character sequence appears in a `//` comment.

`--Werror-implicit-function-declaration`
Generate warning (or error) whenever a function is used before being declared. Same as `--Wimplicit-function-declaration`.

`--Wformat`
Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense. This includes standard functions, and others specified by format attributes (Section 5.2.25) in the `printf`, `scanf`, `strftime` and `strftime` (an X/Open extension, not in the C standard) families.
The formats are checked against the format features supported by GNU libc version 2.2. These include all ISO C90 and C99 features, as well as features from the Single Unix Specification and some BSD and GNU extensions. Other library implementations may not support all these features; GCC does not support warning about features that go beyond a particular library’s limitations. However, if -pedantic is used with -Wformat, warnings will be given about format features not in the selected standard version (but not for strftime formats, since those are not in any version of the C standard). See Section 3.4.3.

Since -Wformat also checks for null format arguments for several functions, -Wformat also implies -Wnonnull.

**NOTE** The following six format options offer additional control over certain aspects of the format checking performed by -Wformat. Note however that unlike -Wformat these options are not specified by -Wall.

- **-Wformat=2**
  Enable -Wformat plus format checks not included in -Wformat. Currently equivalent to “-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k”.

- **-Wno-format-extra-args**
  If -Wformat is specified, do not warn about excess arguments to a printf or scanf format function. The C standard specifies that such arguments are ignored. Where the unused arguments lie between used arguments that are specified with $ operand number specifications, normally warnings are still given, since the implementation could not know what type to pass to va_arg to skip the unused arguments. However, in the case of scanf formats, this option will suppress the warning if the unused arguments are all pointers, since the Single Unix Specification says that such unused arguments are allowed.

- **-Wformat-nonliteral**
  If -Wformat is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a va_list.

- **-Wformat-security**
  If -Wformat is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to printf and scanf functions where the format string is not a string literal and there are no format arguments, as in printf (foo);. This may be a security hole if the format string came from untrusted input and contains %n. (This is currently a subset of what -Wformat-nonliteral warns about, but in future warnings may be added to -Wformat-security that are not included in -Wformat-nonliteral.)

- **-Wformat-y2k**
  If -Wformat is specified, also warn about strftime formats which may yield only a two-digit year.

- **-Wno-format-zero-length**
  If -Wformat is specified, do not warn about zero-length formats. The C standard specifies that zero-length formats are allowed.
-Wimplicit
  Same as -Wimplicit-int and -Wimplicit-function-declaration.

- Wimplicit-function-declaration
  Generate warning (or error) whenever a function is used before being declared.
  Same as -Werror-implicit-function-declaration.

- Wimplicit-int
  Warn when a declaration does not specify a type.

- Wno-import
  Inhibit warning messages about the use of #import.

- Winit-self
  Warn about uninitialized variables which are initialized with themselves. Note this option can only be used with the -Wuninitialized option, which in turn only works with -O1 and above.

  For example, GCC will warn about `i` being uninitialized in the following snippet only when -Winit-self has been specified:

  ```
  int f()
  {
    int i = i;
    return i;
  }
  ```

- Wmain
  Warn if the properties of function `main` are suspicious. `main` should be a function with external linkage, returning int, and taking either zero, two, or three arguments of appropriate types.

- Wmissing-braces
  Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, but that for `b` is fully bracketed.

  ```
  int a[2][2] = { 0, 1, 2, 3 };
  int b[2][2] = { { 0, 1 }, { 2, 3 } };
  ```

- Wnonnull
  Warn about passing a null pointer for arguments marked as requiring a non-null value by thenonnull function attribute.

  - Wnonnull is included in -Wall and -Wformat. It can be disabled with the -Wnononnull option.

- Wparentheses
  Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

  Also warn for comparisons such as `x<=y<=z`. This comparison is equivalent to:

  ```
  (x<=y ? 1 : 0) <= z
  ```

  which is a different interpretation from ordinary mathematical notation.
Also warn about constructions where there may be confusion to which if statement an else branch belongs. Here is an example of such a case:

```c
{
    if (a)
    {
        if (b)
            foo ();
        else
            bar ();
    }
}
```

Every else branch belongs to the innermost possible if statement, which in this example is if (b). This is often not what the programmer expected, as illustrated in the above example by indentation the programmer chose. When there is the potential for this confusion, the compiler generates a warning when this flag is specified. To eliminate the warning, add explicit braces around the innermost if statement so there is no way the else could belong to the enclosing if. The resulting code would look like this:

```c
{
    if (a)
    {
        if (b)
            foo ();
        else
            bar ();
    }
}
```

This option can be disabled with -Wno-parentheses when -Wall is used.

- Wpointer-sign
Warn for pointer argument passing or assignment with different signedness. This option is supported for C only. It is enabled by -Wall and -pedantic, and can be disabled with -Wno-pointer-sign.

- Wreturn-type
Warn whenever a function is defined with a return-type that defaults to int. Also warn about any return statement with no return-value in a function whose return-type is not void.

For C++, a function without return type always produces a diagnostic message, even when -Wno-return-type is specified. The only exceptions are `main` and functions defined in system headers.

- Wsequence-point
Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard.
The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a &&, ||, ?, : or (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places.

Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that “Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.” If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Examples of code with undefined behavior are a = a++; a[n] = b[n++] and a[i++] = i;. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

The C standard is worded confusingly, therefore there is some debate over the precise meaning of the sequence point rules in subtle cases. Links to discussions of the problem, including proposed formal definitions, can be found on the GCC readings page, at http://gcc.gnu.org/readings.html.

**NOTE**  The present implementation of this option only works for C programs. A future implementation may also work for C++ programs.

- **-Wstrict-aliasing**
- **-Wstrict-aliasing=2**

Warn about code which might break the strict aliasing rules that the compiler is using for optimization. The warning does not catch all cases, but does attempt to catch the more common pitfalls. It is included in -Wall.

When the optional argument is specified this option catches more cases, but will also generate warnings for some ambiguous cases that are safe.

Enabled when -fstrict-aliasing is specified.
-Wstrict-overflow
-Wstrict-overflow=level
Warn about cases where the compiler optimizes with the assumption that signed overflow does not occur. This option is active only when -fstrict-overflow is enabled. Because this option warns only in cases of optimization, its behavior depends on the optimization level.

This option can generate warnings about code which is not actually a problem (i.e., false positive warnings where no overflow occurs). No warnings are generated for the use of undefined signed overflow when estimating how many iterations a loop will require (and in particular when determining whether a loop will be executed at all).

The optional argument level can assume the following values:

1
Warn about cases that are both questionable and easy to avoid, such as the expression \( x + 1 > x \) (which the compiler simplifies to the value 1) with -fstrict-overflow enabled. Note that level 1 is enabled by -Wall; any higher levels are not, and must be explicitly specified.

2
Additionally warn about cases where a comparison is simplified to a constant, such as the expression abs(x) >= 0. Note that this simplification can occur only when -fstrict-overflow is enabled, because abs(INT_MIN) overflows to INT_MIN, which is less than zero. (Level 2 is the default level value.)

3
Additionally warn about other cases where a comparison is simplified, such as when \( x + 1 > 1 \) gets simplified to \( x > 0 \).

4
Additionally warn about other simplifications not covered by the above cases, such as when \( (x*10)/5 \) gets simplified to \( x*2 \).

5
Additionally warn about cases where the compiler reduces the magnitude of a constant involved in a comparison, such as when \( x+2 > y \) gets simplified to \( x+1 >= y \). Note that these cases are reported only at the highest warning level: because this simplification applies to so many comparisons, warning level 5 can generate a large number of false positives.

-Wswitch
Warn whenever a switch statement has an index of enumerated type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) case labels outside the enumeration range also provoke warnings when this option is used.

-Wswitch-default
Warn whenever a switch statement does not have a default case.
-Wswitch-enum
Warn whenever a switch statement has an index of enumerated type and lacks a case for one or more of the named codes of that enumeration. case labels outside the enumeration range also provoke warnings when this option is used.

-Wtrigraphs
Warn if any trigraphs are encountered that might change the meaning of the program (trigraphs within comments are not warned about).

-Wuninitialized
Warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a setjmp call. In C++, warn if a non-static reference or non-static const member appears in a class without constructors.

If you want to warn about code which uses the uninitialized value of the variable in its own initializer, use the -Winit-self option.

Warnings are generated for the individual uninitialized or clobbered elements of structure, union, and array variables, as well as for variables that are uninitialized or clobbered as a whole. Warnings are not generated for variables or elements declared volatile.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because the compiler is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```c
{  
    int x;  
    switch (y)
    {  
        case 1: x = 1;  
            break;
        case 2: x = 4;  
            break;
        case 3: x = 5;  
    }  
    foo (x);  
}
```

If the value of y is always 1, 2 or 3, then x is always initialized, but GCC doesn’t know this. Here is another common case:

```c
{  
    int save_y;  
    if (change_y) save_y = y, y = new_y;  
    ...  
    if (change_y) y = save_y;
}
```

This has no bug because save_y is used only if it is set.

This option also warns when a non-volatile automatic variable might be changed by a call to longjmp. These warnings as well are possible only in optimizing compilation.
The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 5.2.25.

Warnings are generally more accurate when some level of optimization is enabled (Section 3.4.9).

-Wunknown-pragmas
Warn when a `#pragma` directive is encountered which is not understood by GCC. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the `-Wall` command line option.

-Wunused
All of the `-Wunused` options below, combined.

In order to get a warning about an unused function parameter, you must either specify `-Wextra -Wunused` (note that `-Wall` implies `-Wunused`), or separately specify `-Wunused-parameter`.

-Wunused-function
Warn whenever a static function is declared but not defined or a non-inline static function is unused.

-Wunused-label
Warn whenever a label is declared but not used.

To suppress this warning use the `unused` attribute (Section 5.2.25).

-Wunused-parameter
Warn whenever a function parameter is unused aside from its declaration.

To suppress this warning use the `unused` attribute (Section 5.2.25).

-Wunused-variable
Warn whenever a local variable or non-constant static variable is unused aside from its declaration

To suppress this warning use the `unused` attribute (Section 5.2.25).

-Wunused-value
Warn whenever a statement computes a result that is explicitly not used.

To suppress this warning cast the expression to `void`.

-Wvolatile-register-var
Warn if a register variable is declared volatile. Note that the `volatile` modifier does not inhibit all optimizations that may eliminate reads or writes to register variables.
3.4.7 Advisory warning messages

This section describes the options for displaying advisory warning messages. For more information on warnings see Section 3.4.6.

-Waggregate-return
Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

-Wbad-function-cast (C only)
Warn whenever a function call is cast to a non-matching type. For example, warn if `int malloc()` is cast to anything `*`.

-Wcast-align
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

-Wcast-qual
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

-Wclobbered
Warn whenever a variable might be changed by `longjmp` or `vfork`.

This option is enabled by -Wextra.

-Wconversion
Warn whenever an implicit conversion may alter a value. Conversions include:

- Real and integer (`abs(x)`, when `x` is double)
- Signed and unsigned (`unsigned ui = -1`)
- Larger and smaller types (`sqrtf(M_PI)`) (C++)

This option does not warn for explicit casts, or if the value is not altered by the conversion (`abs(2.0)`). Warnings between signed and unsigned integers can be disabled by using -Wno-sign-conversion.

In C++ this option additionally warns for the following conversions:

- NULL and non-pointer types
- Confusing overload resolutions for user-defined conversions
- Non-operator conversions: to `void`, the same type, a base class, or a reference to them.

In C++ warnings between signed and unsigned integers are disabled by default unless -Wsign-conversion is used.

-Wdeclaration-after-statement (C only)
Warn when a declaration is found after a statement in a block. This construct, known from C++, was introduced with ISO C99 and is by default allowed in GCC. It is not supported by ISO C90. See Section 5.3.6.

-Wno-deprecated-declarations
Do not warn about uses of functions, variables, and types marked as deprecated by using the `deprecated` attribute (Section 5.2.25).
-Wdisabled-optimization
Warn if a requested optimization pass is disabled. This warning does not generally indicate that there is anything wrong with your code; it merely indicates that GCC’s optimizers were unable to handle the code effectively. Often, the problem is that your code is too big or too complex; GCC will refuse to optimize programs when the optimization itself is likely to take inordinate amounts of time.

-Wno-div-by-zero
Do not warn about compile-time integer division by zero. Floating point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs.

-Wempty-body
Warn if an if, else, or do while statement contains an empty body.
This option is enabled by -Wextra.

-Wendif-labels
Warn whenever an #else or an #endif are followed by text.

-Werror
Make all warnings into errors.

-Wextra
Enable the following warning options:
- Wclobbered
- Wempty-body
- Wignored-qualifiers
- Wmissing-field-initializers
- Wmissing-parameter-type (C only)
- Wold-style-declaration (C only)
- Woverride-init
- Wsign-compare
- Wtype-limits
- uninitialized
- Wunused-parameter (only with -Wunused or -Wall)

Also print warning messages for the following events:
❖ Comparing a pointer against integer zero with <, <=, >, or >=
❖ (C++ only) An enumerator and an non-enumerator both appear in a conditional expression.
❖ (C++ only) A non-static reference or non-static const member appears in a class without constructors.
❖ (C++ only) Ambiguous virtual bases.
❖ (C++ only) Subscripting an array which has been declared register.
- (C++ only) A base class is not initialized in a derived class’s copy constructor.
- (C++ only) Taking the address of a variable which has been declared register.
- (C++ only) A base class is not initialized in a derived class' copy constructor.

**NOTE** -Wextra was formerly named -W. The older option name is still supported, but the newer name is more descriptive.

- **-Wfloat-equal**
  Warn if floating point values are used in equality comparisons.
  The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analyzing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that’s a different problem). In particular, instead of testing for equality, you would check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

- **-Wframe-larger-than=len**
  Warn if the size of a function frame exceeds len bytes.
  Note that this option is approximate and not conservative – the actual requirements may be greater than len even if a warning is not generated.
  Note that any space allocated by alloca, variable-length arrays, or related constructs is not included when determining if a warning should be generated.

- **-Wignored-qualifiers**
  Warn whenever a function return type has a qualifier such as const. In ISO C such type qualifiers have no effect because the value returned by a function is not an lvalue. In C++ the warning is generated only for scalar types or for void.
  Note that ISO C prohibits qualified void return types on functions; therefore, such return types always generate a warning even without this option.
  This option is enabled by -Wextra.

- **-Winline**
  Warn if a function can not be inlined and it was declared as inline. Even with this option, the compiler will not warn about failures to inline functions declared in system headers.
  The compiler uses a variety of heuristics to determine whether or not to inline a function. For example, the compiler takes into account the size of the function being inlined and the amount of inlining that has already been done in the current function. Therefore, seemingly insignificant changes in the source program can cause the warnings produced by -Winline to appear or disappear.
-Wno-invalid-offsetof (C++ only)
Suppress warnings from applying the offsetof macro to a non-POD type.
According to the 1998 ISO C++ standard, applying offsetof to a non-POD type is undefined. In existing C++ implementations, however, offsetof typically gives meaningful results even when applied to certain kinds of non-POD types. (Such as a simple struct that fails to be a POD type only by virtue of having a constructor.) This flag is for users who are aware that they are writing nonportable code and who have deliberately chosen to ignore the warning about it.

The restrictions on offsetof may be relaxed in a future version of the C++ standard.

-Winvalid-pch
Warn if a precompiled header (see Section 3.10) is found in the search path but can’t be used.

-Wlarger-than-len
Warn whenever an object of larger than len bytes is defined.

-Wlong-long
Warn if long long type is used. This is the default setting; to inhibit the warning messages, use -Wno-long-long. Flags -Wlong-long and -Wno-long-long are taken into account only when -pedantic flag is used.

-Wmissing-declarations
Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files.

-Wmissing-format-attribute
If -Wformat is enabled, also warn about functions which might be candidates for format attributes. Note these are only possible candidates, not absolute ones. GCC will guess that format attributes might be appropriate for any function that calls a function like vprintf or vscanf, but this might not always be the case, and some functions for which format attributes are appropriate may not be detected. This option has no effect unless -Wformat is enabled (possibly by -Wall).

-Wmissing-noreturn
Warn about functions which might be candidates for attribute noreturn. Note these are only possible candidates, not absolute ones. Care should be taken to manually verify functions actually do not ever return before adding the noreturn attribute, otherwise subtle code generation bugs could be introduced. You will not get a warning for main in hosted C environments.

-Wmissing-prototypes (C only)
Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

-Wno-multichar
Do not warn if a multicharacter constant (‘FOOF’) is used. Usually they indicate a typo in the user’s code, as they have implementation-defined values, and should not be used in portable code.
-Wnested-externs (C only)
Warn if an extern declaration is encountered within a function.

-Wold-style-definition (C only)
Warn if an old-style function definition is used. A warning is given even if there is a previous prototype.

-Wpacked
Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure. Such structures may be mis-aligned for little benefit. For instance, in this code, the variable f.x in struct bar will be misaligned even though struct bar does not itself have the packed attribute:

```c
struct foo {
    int x;
    char a, b, c, d;
} __attribute__((packed));
struct bar {
    char z;
    struct foo f;
};
```

-Wpadded
Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller.

-Wpointer-arith
Warn about anything that depends on the “size of” a function type or of void. GNU C assigns these types a size of 1, for convenience in calculations with void * pointers and pointers to functions.

-Wredundant-decls
Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

-Wshadow
Warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed.

-Wsign-compare
Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by -Wextra; to get the other warnings of -Wextra without this warning, use -Wextra -Wno-sign-compare.

-Wsign-conversion
Warn for implicit conversions that may change the sign of an integer value, like assigning a signed integer expression to an unsigned integer variable. An explicit cast silences the warning. In C, this option is also enabled by -Wconversion.

-Wstrict-prototypes (C only)
Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)
**-Wsystem-headers**
Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells GCC to emit warnings from system headers as if they occurred in user code. However, note that using -Wall in conjunction with this option will *not* warn about unknown pragmas in system headers—so that, -Wunused-pragmas must also be used.

**-Wtraditional (C only)**
Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and/or problematic constructs which should be avoided.

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but does not in ISO C.
- In traditional C, some preprocessor directives did not exist. Traditional preprocessors would only consider a line to be a directive if the # appeared in column 1 on the line. Therefore -Wtraditional warns about directives that traditional C understands but would ignore because the # does not appear as the first character on the line. It also suggests you hide directives like #pragma not understood by traditional C by indenting them. Some traditional implementations would not recognize #elif, so it suggests avoiding it altogether.
- A function-like macro that appears without arguments.
- The unary plus operator.
- The U integer constant suffix, or the F or L floating point constant suffixes. (Traditional C does support the L suffix on integer constants.) Note, these suffixes appear in macros defined in the system headers of most modern systems, e.g. the _MIN/ _MAX macros in <limits.h>. Use of these macros in user code might normally lead to spurious warnings, however GCC’s integrated preprocessor has enough context to avoid warning in these cases.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type long.
- A non-static function declaration follows a static one. This construct is not accepted by some traditional C compilers.
- The ISO type of an integer constant has a different width or signedness from its traditional type. This warning is only issued if the base of the constant is ten. I.e. hexadecimal or octal values, which typically represent bit patterns, are not warned about.
- Usage of ISO string concatenation is detected.
- Initialization of automatic aggregates.
- Identifier conflicts with labels. Traditional C lacks a separate namespace for labels.
Initialization of unions. If the initializer is zero, the warning is omitted. This is done under the assumption that the zero initializer in user code appears conditioned on e.g. __STDC__ to avoid missing initializer warnings and relies on default initialization to zero in the traditional C case.

Conversions by prototypes between fixed/floating point values and vice versa. The absence of these prototypes when compiling with traditional C would cause serious problems. This is a subset of the possible conversion warnings, for the full set use -Wconversion.

Use of ISO C style function definitions. This warning intentionally is not issued for prototype declarations or variable-argument functions because these ISO C features will appear in your code when using the library’s traditional C compatibility macros, __PARAMS and __VPARAMS. This warning is also bypassed for nested functions because that feature is already a GCC extension and thus not relevant to traditional C compatibility.

-Wtype-limits
Warn if a comparison is always true or always false because of the limited range of the data type. For example, warn if an unsigned variable is compared to zero using the operators < or >=. This option does not warn for constant expressions.

This option is enabled by -Wextra.

-Wundef
Warn if an undefined identifier is evaluated in an #if directive.

-Wunreachable-code
Warn if the compiler detects that code will never be executed.

This option is intended to warn when the compiler detects that at least a whole line of source code will never be executed, because some condition is never satisfied or because it is after a procedure that never returns.

It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently-unreachable code.

For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.

This option is not made part of -Wall because in a debugging version of a program there is often substantial code which checks correct functioning of the program and is, hopefully, unreachable because the program does work. Another common use of unreachable code is to provide behavior which is selectable at compile-time.

-Wwrite-strings
When compiling C, give string constants the type const char[length] so that copying the address of one into a non-const char * pointer will get a warning; when compiling C++, warn about the deprecated conversion from string constants to char *. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using const in declarations and prototypes. Otherwise, it will just be a nuisance; this is why -Wall does not request these warnings.
3.4.8 Debugging

GCC include options that are used to debug your program or GCC itself.

-dumpmachine
Print the name of the compiler’s target machine and exit.

-dumpspecs
Print the compiler’s built-in specs and exit. (This is used when GCC itself is being built.) See Section 3.8.

-dumpversion
Print the compiler version (for example, 3.0) and exit.

-fdump-class-hierarchy[-option...]  (C++ only)
Dump a representation of each class’s hierarchy and virtual function table layout to a file named file.class, where file is the source file name. The parameter optionally specifies the details of the dump as described for the -fdump-tree option.

-fdump-translation-unit[-option...]
Dump a representation of the tree structure for the entire translation unit to a file named file.tu, where file is the source file name. The parameter optionally specifies the details of the dump as described for the -fdump-tree option.

-fdump-tree-switch[-option...]  (C++ only)
Dump the intermediate language tree to a file at the specified stage of processing. The file name is created by appending a switch-specific suffix to the source file name.

The switch parameter has the following possible values:

original
Dump tree to a file named file.original before any tree-based optimization.

optimized
Dump tree to a file named file.optimized after all tree-based optimization.

inlined
Dump tree to a file named file.inlined after function inlining.

The optional -option parameter is used to control the details of the dump. It consists of a list of one or more dump control option values separated by ‘-’ characters.

The option parameter has the following possible values:

address
Print address of each node. Usually this is not meaningful, as it changes according to the environment and source file. Its primary use is for tying up a dump file with a debug environment.
slim
Inhibit dumping of members of a scope or body of a function merely because that
scope has been reached. Only dump such items when they are directly reachable
by some other path.
all
Enable all dump control options (including address and slim).

NOTE  Not all dump controls are applicable to all dump types; those that are not
meaningful are ignored.

-feliminate-dwarf2-dups
Compress DWARF2 debugging information by eliminating duplicated
information about each symbol. This option only makes sense when generating
DWARF2 debugging information with -gdwarf-2.

-feliminate-unused-debug-symbols
Produce debugging information in stabs format (if that is supported), for only
symbols that are actually used.

-feliminate-unused-debug-types
Normally, when producing DWARF2 output, GCC will emit debugging
information for all types declared in a compilation unit, regardless of whether or
not they are actually used in that compilation unit. Sometimes this is useful, such
as if, in the debugger, you want to cast a value to a type that is not actually used in
your program (but is declared). More often, however, this results in a significant
amount of wasted space. With this option, GCC will avoid producing debug
symbol output for types that are nowhere used in the source file being compiled.

-fmem-report
Makes the compiler print some statistics about permanent memory allocation
when it finishes.

-fprofile-arcs
Add code so that program flow arcs are instrumented. During execution the
program records how many times each branch and call is executed and how many
times it is taken or returns. When the compiled program exits it saves this data to a
file called auxname.gcd for each source file. The data may be used for profile-
directed optimizations (-fbranch-probabilities), or for test coverage
analysis (-ftest-coverage). Each object file’s auxname is generated from the
name of the output file, if explicitly specified and it is not the final executable,
otherwise it is the basename of the source file. In both cases any suffix is removed
(e.g. foo.gcd for input file dir/foo.c, or dir/foo.gcd for output file
specified as -o dir/foo.o).

Compile the source files with -fprofile-arcs plus optimization and code
generation options. For test coverage analysis, use the additional -ftest-
coverage option. You do not need to profile every source file in a program.

Link your object files with -lgcov or -fprofile-arcs (the latter implies the
former).
Run the program on a representative workload to generate the arc profile information. This may be repeated any number of times. You can run concurrent instances of your program, and provided that the file system supports locking, the data files will be correctly updated. Also fork calls are detected and correctly handled (double counting will not happen).

For profile-directed optimizations, compile the source files again with the same optimization and code generation options plus -fbranch-probabilities (see Section 3.4.9).

For test coverage analysis, use the GNU gcov utility to produce human readable information from the .gcno and .gcda files. For more information see the GNU gcov documentation.

With -fprofile-arcs, for each function of your program GCC creates a program flow graph, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

-frandom-seed=string
This option provides a seed that GCC uses when it would otherwise use random numbers. It is used to generate certain symbol names that have to be different in every compiled file. It is also used to place unique stamps in coverage data files and the object files that produce them. You can use the -frandom-seed option to produce reproducibly identical object files.

The string should be different for every file you compile.

-fsched-verbose=n
On targets that use instruction scheduling, this option controls the amount of debugging output the scheduler prints. This information is written to standard error, unless -dS or -dR is specified, in which case it is output to the usual dump listing file, .sched or .sched2 respectively. However for n greater than nine, the output is always printed to standard error.

For n greater than zero, -fsched-verbose outputs the same information as -dRS. For n greater than one, it also output basic block probabilities, detailed ready list information and unit/instruction info. For n greater than two, it includes RTL at abort point, control-flow and regions info. And for n over four, -fsched-verbose also includes dependence info.

-ftest-coverage
Produce a notes file that the GNU gcov utility can use to show program coverage. Each source file’s note file is called auxname.gcno. Refer to the -fprofile-arcs option above for a description of auxname and instructions on how to generate test coverage data. Coverage data will match the source files more closely, if you do not optimize.

-ftime-report
Makes the compiler print some statistics about the time consumed by each pass when it finishes.
-g\texttt{level}

Request debugging information for the listed options (which are defined in detail below), while also using \texttt{level} to specify how much information is displayed. No space appears between the option name and the \texttt{level} parameter. The possible level values are 1, 2, and 3. The default is 2.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don’t plan to debug. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use \texttt{-g3}.

Note that in order to avoid confusion between DWARF1 debug level 2, and DWARF2 \texttt{-gdwarf-2} does not accept a concatenated debug level. Instead use an additional \texttt{-g\texttt{level}} option to change the debug level for DWARF2.

-\texttt{g}

Produce debugging information in the DWARF2 format. The GDB debugger can work with this information.

\texttt{-g} can be used when code optimization is specified with \texttt{-O}. Note that the shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GCC is generated with the capability for more than one debugging format.

-\texttt{p}

Generate extra code to write profile information suitable for the analysis program \texttt{prof}. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-\texttt{pg}

Generate extra code to write profile information suitable for the analysis program \texttt{gprof}. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-\texttt{print\_file\_name=library}

Print the full absolute name of the library file \texttt{library} that would be used when linking, and exit. With this option, GCC does not compile or link anything; it just prints the file name.
-print-libgcc-file-name
Same as -print-file-name=libgcc.a.

This is useful when you use -nostdlib or -nodefaultlibs but do want to link with libgcc.a. For example:

hexagon-gcc -nostdlib files... 'hexagon-gcc -print-libgcc-file-name'

-print-multi-directory
Print the directory name corresponding to the multilib selected by any other switches present in the command line. This directory is supposed to exist in GCC_EXEC_PREFIX.

-print-multi-lib
Print the mapping from multilib directory names to compiler switches that enable them. The directory name is separated from the switches by ;, and each switch starts with an @ instead of the -, without spaces between multiple switches. This is supposed to ease shell-processing.

-print-multi-os-directory
Print the relative path that is appended to each element in the default set of multilib search paths (to create the set of complete search paths).

The result of this option may depend on other options.

-print-prog-name=program
Like -print-file-name, but searches for a program such as cpp.

-print-search-dirs
Print the name of the configured installation directory and a list of program and library directories GCC will search, and exit.

This is useful when GCC prints the error message installation problem, cannot exec cpp0: No such file or directory. To resolve this you either need to put cpp0 and the other compiler components where GCC expects to find them, or you can set the environment variable GCC_EXEC_PREFIX to the directory where you installed them. Don’t forget the trailing ’/’. See Section 3.9.

-Q
Display function names as they are compiled, and statistics for each compiler pass.

-save-temps
Store the usual “temporary” intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling foo.c with -c -save-temps would produce files foo.i and foo.s, as well as foo.o. This creates a preprocessed foo.i output file even though the compiler now normally uses an integrated preprocessor.
-time

Report the CPU time taken by each subprocess in the compilation sequence. For C source files, this is the compiler proper and assembler (plus the linker if linking is done). The output looks like this:

```bash
# cc1 0.12 0.01
# as 0.00 0.01
```

The first number on each line is the “user time,” that is time spent executing the program itself. The second number is “system time,” time spent executing operating system routines on behalf of the program. Both numbers are in seconds.

### 3.4.9 Optimization

These options control various sorts of code optimizations performed by GCC.

Without any optimization option, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Enabling optimization options makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Using the `--unit-at-a-time` flag will allow the compiler to consider information gained from later functions in the file when compiling a function. Compiling multiple files at once to a single output file (and using `--unit-at-a-time`) will allow the compiler to use information gained from all of the files when compiling each of them.

-0

-01

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -0, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-0 enables the following optimization flags:

- fdefer-pop
- fmerge-constants
- fthread-jumps
- floop-optimize
- fif-conversion
- fif-conversion2
- fdelayed-branch
- fguess-branch-probability
- fcprop-registers

-0 also enables `--omit-frame-pointer` on machines where doing so does not interfere with debugging.
-O2

Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed trade-off. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

-O2 enables all optimization flags specified by -O. It also enables the following optimization flags:

-foptimize-sibling-calls
-fstrength-reduce
-fcse-follow-jumps -fcse-skip-blocks
-frerun-cse-after-loop -frerun-loop-opt
-fgcse -fgcse-lm -fgcse-sm -fgcse-las
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fregmove
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fcaller-saves
-fpeephole2
-freorder-blocks -freorder-functions
-fstrict-aliasing
-funit-at-a-time
-falign-functions -falign-jumps
-falign-loops -falign-labels
-fcrossjumping

Please note the warning under -fgcse about invoking -O2 on programs that use computed GOTO statements.

-O3

Optimize yet more. -O3 enables all optimizations specified by -O2, and also enables -finline-functions, -fweb and -frename-registers.

-O0

Do not optimize. This is the default.

-Os

Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

-Os disables the following optimization flags:

-falign-functions -falign-jumps -falign-loops
-falign-labels -freorder-blocks -fprefetch-loop-arrays

NOTE If you specify multiple -O options (with or without level numbers) only the last such option is effective.
### 3.4.10 Specific optimizations

The following options control specific code optimizations. Most of them are either enabled implicitly by the -O options, or are related to ones implicitly enabled. These options are explicitly specified only in the rare cases where it is worthwhile to “fine-tune” the optimizations.

**NOTE** Not all optimizations are controlled directly by a command option. Only optimizations that have an option are listed below.

Math-related specific optimizations are described in Section 3.4.11.

- **-falign-functions**
  - -falign-functions=n
  Align the start of functions to the next power-of-two greater than n, skipping up to n bytes. For instance, -falign-functions=32 aligns functions to the next 32-byte boundary, but -falign-functions=24 would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less.
  - -fno-align-functions and -falign-functions=1 are equivalent and mean that functions will not be aligned.
  - Some assemblers only support this flag when n is a power of two; in that case, it is rounded up.
  - If n is not specified or is zero, use a machine-dependent default.
  - Enabled at levels -O2, -O3.

- **-falign-jumps**
  - -falign-jumps=n
  Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to n bytes like -falign-functions. In this case, no dummy operations need be executed.
  - -fno-align-jumps and -falign-jumps=1 are equivalent and mean that branch targets will not be aligned.
  - If n is not specified or is zero, use a machine-dependent default.
  - Enabled at levels -O2, -O3.
-falign-labels
-\( f\text{-align-labels}=n \)
Align all branch targets to a power-of-two boundary, skipping up to \( n \) bytes like -falign-functions. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code.

-fno-align-labels and -falign-labels=1 are equivalent and mean that labels will not be aligned.

If -falign-loops or -falign-jumps are applicable and are greater than this value, then their values are used instead.

If \( n \) is not specified or is zero, use a machine-dependent default which is very likely to be 1, meaning no alignment.

Enabled at levels -O2, -O3.

-\( f\text{-align-loops} \)
-\( f\text{-align-loops}=n \)
Align loops to a power-of-two boundary, skipping up to \( n \) bytes like -falign-functions. The hope is that the loop will be executed many times, which will make up for any execution of the dummy operations.

-fno-align-loops and -falign-loops=1 are equivalent and mean that loops will not be aligned.

If \( n \) is not specified or is zero, use a machine-dependent default.

Enabled at levels -O2, -O3.

-fno-branch-count-reg
Do not use “decrement and branch” instructions on a count register. Instead, generate an instruction sequence which decrements the register, compares it to zero, and branches based on the result.

The default setting is enabled when -fstrength-reduce is enabled.

-fbranch-probabilities
After running a program compiled with -fprofile-arcs (Section 3.4.8), you can compile it a second time using -fbranch-probabilities to improve optimizations based on the number of times each branch was taken. When the program compiled with -fprofile-arcs exits it saves arc execution counts to a file called sourcename.gcda for each source file. The information in this data file is highly dependent on the structure of the generated code, so you must use the same source code and the same optimization options for both compilations.

With -fbranch-probabilities, GCC puts a REG BR_PROB note on each JUMP_INSN and CALL_INSN. These can be used to improve optimization. Currently, they are only used in one place: in reorg.c, instead of guessing which path a branch is mostly to take, the REG BR_PROB values are used to exactly determine which path is taken more often.
-fbranch-target-load-optimize
Perform branch target register load optimization before prologue / epilogue threading. The use of target registers can typically be exposed only during reload, thus hoisting loads out of loops and doing inter-block scheduling needs a separate optimization pass.

-fbranch-target-load-optimize2
Perform branch target register load optimization after prologue / epilogue threading.

-fcaller-saves
Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is always enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

Enabled at levels -O2, -O3, -Os.

-fconserve-stack
Attempt to minimize stack usage. The compiler tries to use less stack space, even if it makes the program slower. Implicitly sets large-stack-frame to 100 and large-stack-frame-growth to 400 (see --param below for details).

-fno-cprop-registers
After register allocation and post-register allocation instruction splitting, the compiler performs a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.

Disabled at levels -O, -O2, -O3, -Os.

-fcrossjumping
Perform cross-jumping transformation. This transformation unifies equivalent code and saves code size. The resulting code may or may not perform better than without cross-jumping.

Enabled at levels -O, -O2, -O3, -Os.

-fcse-follow-jumps
In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an if statement with an else clause, CSE will follow the jump when the condition tested is false.

Enabled at levels -O2, -O3, -Os.

-fcse-skip-blocks
This is similar to -fcse-follow-jumps, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if.

Enabled at levels -O2, -O3, -Os.
**-fdata-sections**  
Place each data item into its own section in the output file. The name of the data item determines the section’s name in the output file.

For more information see Section 3.11.5.

**NOTE**  
Use this option only when there are significant benefits from doing so. Using it causes the assembler and linker to create larger object and executable files, and also be slower.

**-fno-default-inline**  
Do not make member functions inline by default merely because they are defined inside the class scope (C++ only). Otherwise, when you specify `-O`, member functions defined inside class scope are compiled inline by default; i.e., you don’t need to add `inline` in front of the member function name.

**-fno-defer-pop**  
Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

Disabled at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fdelayed-branch**  
If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fdelete-null-pointer-checks**  
Use global dataflow analysis to identify and eliminate useless checks for null pointers. The compiler assumes that dereferencing a null pointer would have halted the program. If a pointer is checked after it has already been dereferenced, it cannot be null.

In some environments, this assumption is not true, and programs can safely dereference null pointers. Use `-fno-delete-null-pointer-checks` to disable this optimization for programs which depend on that behavior.

Enabled at levels `-O2`, `-O3`, `-Os`.

**-fexpensive-optimizations**  
Perform a number of minor optimizations that are relatively expensive.

Enabled at levels `-O2`, `-O3`, `-Os`. 

-ffloat-store
Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have. Similarly for the x86 architecture. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use `-ffloat-store` for such programs, after modifying them to store all pertinent intermediate computations into variables.

Enabled at levels `-O2`, `-O3`, `-Os`.

-fforce-addr
Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as `-fforce-mem` may.

-fno-function-cse
Do not put function addresses in registers; make each instruction that calls a constant function contain the function’s address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

The default is `-ffunction-cse`.

-ffunction-sections
Place each function item into its own section in the output file. The name of the function item determines the section’s name in the output file.

For more information see Section 3.11.4.

NOTE Use this option only when there are significant benefits from doing so. It will cause the assembler and linker to be slower and to create larger object and executable files.

-fgcse
Perform a global common subexpression elimination (GCSE) pass. This pass also performs global constant and copy propagation.

Enabled at levels `-O2`, `-O3`, `-Os`.

NOTE When compiling a program using computed GOTO statements (a GCC extension), you may get faster code by specifying `-fno-gcse`.

-fgcse-las
When `-fgcse-las` is enabled, the GCSE pass eliminates redundant loads that come after stores to the same memory location (both partial and full redundancies).

Enabled by default when `-fgcse` is enabled.
**-fgcse-lm**

When `-fgcse-lm` is enabled, GCSE attempts to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.

Enabled by default when `-fgcse` is enabled.

**-fgcse-sm**

When `-fgcse-sm` is enabled, a store motion pass is run after GCSE. This pass attempts to move stores out of loops. When used with `-fgcse-lm`, loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.

Enabled by default when `-fgcse` is enabled.

**-fno-guess-branch-probability**

Do not guess branch probabilities using a randomized model.

Sometimes GCC will opt to use a randomized model to guess branch probabilities, when none are available from either profiling feedback (`-fprofile-arcs`) or `__builtin_expect`. This means that different runs of the compiler on the same program may produce different object code.

In a hard real-time system, people don’t want different runs of the compiler to produce code that has different behavior; minimizing non-determinism is of paramount import. This switch allows users to reduce non-determinism, possibly at the expense of inferior optimization.

The default is `-fguess-branch-probability` at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fif-conversion**

Attempt to transform conditional jumps into branchless equivalents. This includes use of conditional moves; `min`, `max`, `set flags`, and `abs` instructions; and some tricks doable using standard arithmetic. The use of conditional execution is controlled by `if-conversion2`.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fif-conversion2**

Use conditional execution to transform conditional jumps into branchless equivalents.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fno-inline**

Don’t expand any inline functions except those declared with the function attribute `always_inline` (Section 5.2.25).

Note that the same result can be achieved by disabling optimization.

This option is useful when generating a profile.
-finline-functions
Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.

Disabling this option affects the inlining of only those functions heuristically identified by the compiler; it has no effect on the inlining of any other functions.

Enabled at level -O3.

-findirect-inlining
Inline indirect calls discovered at compile time as a result of performing function inlining. This option is active only when -finline-functions is enabled.

Enabled at level -O2.

-finline-limit=n
By default GCC does not automatically inline all functions that are marked as inline (i.e., marked with the inline keyword or defined in a class definition in C++). This flag controls which of these functions are actually inlined.

n specifies the maximum size of a function that can be inlined, which is expressed as the maximum number of pseudo instructions in the function (not counting parameter handling). The default is 600. Increasing the value of n can result in more inlined code at the cost of compilation time and memory consumption.

Decreasing the value of n usually makes the compilation faster and less code will be inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining heavily such as those based on recursive templates with C++.

Inlining is actually controlled by a number of parameters which can be specified individually with the option --param name=value. The -finline-limit=n option sets some of these parameters as follows:

max-inline-insns-single
is set to n/2.

max-inline-insns-auto
is set to n/2.

min-inline-insns
is set to 130 or n/4, whichever is smaller.

max-inline-insns-rtl
is set to n.

See below for a description of the individual parameters that control inlining.

A pseudo instruction represents only an abstract unit of measurement for a function’s size. It does not represent a count of assembly instructions, and its exact meaning may vary between releases.

-fkeep-inline-functions
Even if all calls to a given function are integrated, and the function is declared static, nevertheless output a separate run-time callable version of the function. This switch does not affect extern inline functions.
-fkeep-static-consts
  Emit variables declared static const when optimization isn’t turned on, even if
  the variables aren’t referenced.

  GCC enables this option by default. If you want to force the compiler to check if
  the variable was referenced, regardless of whether or not optimization is turned
  on, use the -fno-keep-static-consts option.

-floop-optimize
  Perform loop optimizations: move constant expressions out of loops, simplify exit
  test conditions and optionally do strength-reduction and loop unrolling as well.

  Enabled at levels -O, -O2, -O3, -Os.

-fmerge-all-constants
  Attempt to merge identical constants and identical variables.

  This option implies -fmerge-constants. In addition to -fmerge-constants
  this considers e.g. even constant initialized arrays or initialized constant variables
  with integral or floating point types. Languages like C or C++ require each non-
  automatic variable to have distinct location, so using this option will result in non-
  conforming behavior.

-fmerge-constants
  Attempt to merge identical constants (string constants and floating point
  constants) across compilation units.

  This option is the default for optimized compilation if the assembler and linker
  support it. Use -fno-merge-constants to inhibit this behavior.

  Enabled at levels -O, -O2, -O3, -Os.

-fmodulo-sched
  Perform swing modulo scheduling immediately before the first scheduling pass.
  This pass examines innermost loops and reorders their instructions by overlapping
  different iterations.

-fold-unroll-all-loops
  Unroll all loops, even if their number of iterations is uncertain when the loop is
  entered. This is done using the old loop unroller whose loop recognition is based
  on notes from the compiler front end. This usually makes programs run more
  slowly. -fold-unroll-all-loops implies the same options as -fold-unroll-
  loops.

-fold-unroll-loops
  Unroll loops whose number of iterations can be determined at compile time or
  upon entry to the loop, using the old loop unroller whose loop recognition is based
  on notes from the compiler front end. -fold-unroll-loops implies both
  -fstrength-reduce and -frerun-cse-after-loop. This option makes code
  larger, and may or may not make it run faster.
-fomit-frame-pointer
Don’t keep the frame pointer in a register for functions that don’t need one. This eliminates the need to generate instructions that save, set up, and restore frame pointers during function calls. Note that this optimization is performed only on leaf functions.

Enabled at levels -O, -O2, -O3, -Os.

-foptimize-register-move
Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. This is especially helpful on machines with two-operand instructions.

Same as -fregmove.

Enabled at levels -O2, -O3, -Os.

-foptimize-sibling-calls
Optimize sibling and tail-recursive calls.

Enabled at levels -O2, -O3, -Os.

-fpeel-loops
Peels the loops for that there is enough information that they do not roll much (from profile feedback). It also enables complete loop peeling (i.e. the complete removal of loops with small constant number of iterations).

-fnpeephole
-fnpeephole2
Disable any machine-specific peephole optimizations. The difference between -fnpeephole and -fnpeephole2 is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.

-fpeephole is enabled by default. -fpeephole2 enabled at levels -O2, -O3, -Os.

-fprefetch-loop-arrays
If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

Disabled at level -Os.

-fprofile-generate
Enable options usually used for instrumenting applications to produce a profile useful for later recompilation with profile feedback based optimization. You must use -fprofile-generate both when compiling and when linking your program.

The following options are enabled: -fprofile-arcs, -fprofile-values, -fvpt.

-fprofile-use
Enable profile feedback directed optimizations, and optimizations generally profitable only with profile feedback available.

The following options are enabled: -fbranch-probabilities, -fvpt, -funroll-loops, -fpeel-loops, -ftracer.
-fprofile-values
   If combined with -fprofile-arcs, it adds code so that some data about values
   of expressions in the program is gathered.

   With -fbranch-probabilities, it reads back the data gathered from profiling
   values of expressions and adds REG_VALUE_PROFILE notes to instructions for
   their later usage in optimizations.

-fregmove
   Same as -foptimize-register-move.

-frename-registers
   Attempt to avoid false dependencies in scheduled code by making use of registers
   left over after register allocation. This optimization will most benefit processors
   with lots of registers. It can, however, make debugging impossible, since variables
   will no longer stay in a “home register”.

-freorder-blocks
   Reorder basic blocks in the compiled function in order to reduce number of taken
   branches and improve code locality.

   Enabled at levels -O2, -O3.

-freorder-functions
   Reorder functions to improve code locality. This is implemented by using the
   special subsections text.hot (for most frequently executed functions) and
   text.unlikely (for unlikely executed functions). Reordering is performed by
   the linker.

   Note that profile feedback must be available to make this option effective. See
   -fprofile-arcs for details.

   Enabled at levels -O2, -O3, -Os.

-frerun-cse-after-loop
   Re-run common subexpression elimination after loop optimizations has been
   performed.

   Enabled at levels -O2, -O3, -Os.

-frerun-loop-opt
   Run the loop optimizer twice.

   Enabled at levels -O2, -O3, -Os.

-fschedule-insns
   If supported for the target machine, attempt to reorder instructions to eliminate
   execution stalls due to required data being unavailable. This helps machines that
   have slow floating point or memory load instructions by allowing other
   instructions to be issued until the result of the load or floating point instruction is
   required.

   Enabled at levels -O2, -O3, -Os.
-fschedule-insns2
Similar to -fschedule-insns, but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Enabled at levels -O2, -O3, -Os.

-fno-sched-interblock
Don’t schedule instructions across basic blocks. This is normally enabled by default when scheduling before register allocation, i.e. with -fschedule-insns or at -O2 or higher.

-fno-sched-spec
Don’t allow speculative motion of non-load instructions. This is normally enabled by default when scheduling before register allocation, i.e. with -fschedule-insns or at -O2 or higher.

-fsched-spec-load
Allow speculative motion of some load instructions. This only makes sense when scheduling before register allocation, i.e. with -fschedule-insns or at -O2 or higher.

-fsched-spec-load-dangerous
Allow speculative motion of more load instructions. This only makes sense when scheduling before register allocation, i.e. with -fschedule-insns or at -O2 or higher.

-fsched-stalled-insns=n
Define how many instructions (if any) can be moved prematurely from the queue of stalled instructions into the ready list, during the second scheduling pass.

-fsched-stalled-insns-dep=n
Define how many instruction groups (cycles) will be examined for a dependency on a stalled instruction that is candidate for premature removal from the queue of stalled instructions. Has an effect only during the second scheduling pass, and only if -fsched-stalled-insns is used and its value is not zero.

-fsched2-use-superblocks
When scheduling after register allocation, do use superblock scheduling algorithm. Superblock scheduling allows motion across basic block boundaries resulting on faster schedules. This option is experimental, as not all machine descriptions used by GCC model the CPU closely enough to avoid unreliable results from the algorithm.

This only makes sense when scheduling after register allocation, i.e. with -fschedule-insns2 or at -O2 or higher.
-fsched2-use-traces
Use -fsched2-use-superblocks algorithm when scheduling after register allocation and additionally perform code duplication in order to increase the size of superblocks using tracer pass. See -ftracer for details on trace formation.

This mode should produce faster but significantly longer programs. Also without -fbranch-probabilities the traces constructed may not match the reality and hurt the performance. This only makes sense when scheduling after register allocation, i.e. with -fschedule-insn2 or at -O2 or higher.

-fstrength-reduce
Perform the optimizations of loop strength reduction and elimination of iteration variables.

Enabled at levels -O2, -O3, -Os.

-fstack-protector
Generate extra code to check certain functions for buffer overflows (such as those caused by stack-smashing attacks). Checking is performed by adding a guard variable to functions identified as having vulnerable objects (including calls to alloca, or buffers larger than 8 bytes). The guard variable is initialized when a function is entered, and checked when the function exits. If a guard check fails, an error message is printed and the program exits. For more information see the parameter ssp_buffer_size.

-fstack-protector-all
Equivalent to -fstack-protector except that all functions have check code added (and not just the ones with vulnerable objects).

-fstrict-aliasing
Allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C (and C++), this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an unsigned int can alias an int, but not a void* or a double. A character type may alias any other type.

Pay special attention to code like this:

```c
union a_union {
    int i;
    double d;
};

int f() {
    a_union t;
    t.d = 3.0;
    return t.i;
}
```
The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with `-fstrict-aliasing`, type-punning is allowed, provided the memory is accessed through the union type. So, the code above will work as expected. However, this code might not:

```c
int f() {
    a_union t;
    int* ip;
    t.d = 3.0;
    ip = &t.i;
    return *ip;
}
```

Enabled at levels `-O2`, `-O3`, `-Os`.

**-fthread-jumps**
Perform optimizations which check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

**-fno-toplevel-reorder**
Do not reorder top-level variables, functions, and `asm` statements – instead, output them in the same order that they appear in the input file. Note that this option does not remove any unreferenced static variables.

This option is intended to support existing code which relies on a particular ordering. For new code, it is recommended to use attributes.

Enabled at level `-O0`.

**-ftracer**
Perform tail duplication to enlarge superblock size. This transformation simplifies the control flow of the function, enabling other optimizations to do a better job.

**-ftree-switch-conversion**
Convert simple initializations in `switch` statements into initializations from a scalar array. Enabled by default at levels `-O2` and higher.

**-ftree-builtin-call-dce**
Perform conditional dead code elimination (DCE) on calls to built-in functions that may set `errno`, but otherwise are free of side-effects. Enabled by default at `-O2` and higher, if `-Os` is not also specified.

**-ftree-vrp**
Perform value range propagation on trees. This optimization is similar to the constant propagation pass, but propagates value ranges instead of values. This allows the optimizer to remove unnecessary range checks such as array bound checks and null pointer checks. This flag is enabled by default at `-O2` and higher.

Null pointer check elimination is performed only if `-fdelete-null-pointer-checks` is enabled.

**-funit-at-a-time**
Parse the whole compilation unit before starting to produce code. This allows some extra optimizations to take place but consumes more memory.
-funroll-all-loops
Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. -funroll-all-loops implies the same options as -funroll-loops.

-funroll-loops
Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. -funroll-loops implies -frerun-cse-after-loop. It also enables complete loop peeling (i.e. complete removal of loops with small constant number of iterations). This option makes code larger, and may or may not make it run faster.

-funswitch-loops
Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

-fvpt
If combined with -fprofile-arcs, it instructs the compiler to add code to gather information about the values of expressions.

With -fbranch-probabilities, it reads back the data gathered and actually performs the optimizations based on them. Currently the optimizations include specialization of division operations using knowledge about the value of the denominator.

-fweb
Constructs webs as commonly used for register allocation purposes and assign each web individual pseudo register. This allows the register allocation pass to operate on pseudo registers directly, but also strengthens several other optimization passes, such as CSE, loop optimizer and trivial dead code remover. It can however make debugging impossible, since variables will no longer stay in a “home register”.

Enabled at levels -O3.

-fno-zero-initialized-in-bss
If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section. E.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

The default is -fzero-initialized-in-bss.

--param name=value
In some places, GCC uses various constants to control the amount of optimization that is done. For example, GCC will not inline functions that contain more that a certain number of instructions. You can control some of these constants on the command-line using the --param option.

The names of specific parameters, and the meaning of the values, are tied to the internals of the compiler, and are subject to change without notice in future releases.

In each case, the value is an integer. The allowable choices for name are specified in the following list:
**align-threshold**  
Fraction of the maximum frequency of executions for a basic block, in order for it to be aligned by the compiler.

**align-loop-iterations**  
Minimum number of iterations for a loop, in order for it to be aligned by the compiler.

**ggc-min-expand**  
GCC uses a garbage collector to manage its own memory allocation. This parameter specifies the minimum percentage by which the garbage collector’s heap should be allowed to expand between collections. Tuning this may improve compilation speed; it has no effect on code generation.

The default is $30\% + 70\% \times (\text{RAM}/1\text{GB})$ with an upper bound of 100% when RAM $\geq 1\text{GB}$. If `getrlimit` is available, the notion of “RAM” is the smallest of actual RAM, RLIMIT_RSS, RLIMIT_DATA and RLIMIT_AS. If GCC is not able to calculate RAM on a particular platform, the lower bound of 30% is used.

Setting this parameter and `ggc-min-heapsize` to zero causes a full collection to occur at every opportunity. This is extremely slow, but can be useful for debugging.

**ggc-min-heapsize**  
Minimum size of the garbage collector’s heap before it begins bothering to collect garbage. The first collection occurs after the heap expands by `ggc-min-expand` percent beyond `ggc-min-heapsize`. Again, tuning this may improve compilation speed, and has no effect on code generation.

The default is `RAM/8`, with a lower bound of 4096 (four megabytes) and an upper bound of 131072 (128 megabytes). If `getrlimit` is available, the notion of “RAM” is the smallest of actual RAM, RLIMIT_RSS, RLIMIT_DATA and RLIMIT_AS. If GCC is not able to calculate RAM on a particular platform, the lower bound is used.

Setting this parameter to a very large value effectively disables garbage collection. Setting both it and `ggc-min-expand` to zero causes a full garbage collection to occur at every opportunity.

**hot-bb-count-fraction**  
Fraction of the maximum count of repetitions of a basic block in a program, in order for it to be considered hot.

**hot-bb-frequency-fraction**  
Fraction of the maximum frequency of executions of a basic block in a program, in order for it to be considered hot.

**inline-call-cost**  
Cost of call instruction relative to simple arithmetic operations (which have a cost of 1). Increasing the cost value effectively increases the amount of inlining for code having a large abstraction penalty (i.e., many functions that just pass the arguments to other functions), and decreases inlining for code with a low abstraction penalty. The default value is 12.
**inline-unit-growth**  
Maximum overall growth of a compilation unit that can be caused by inlining. The default value is 30 (which limits the growth to 1.3 times the original size).

**integer-share-limit**  
Maximum value of a shared integer constant. Small integer constants can use shared data structures, reducing compiler memory usage and compile time. The default value is 256.

**ipcp-unit-growth**  
Maximum overall growth of a compilation unit that can be caused by interprocedural constant propagation. The default value is 10 (which limits the growth to 1.1 times the original size).

**large-function-growth**  
Specify (as percent value) the maximal growth of large function caused by inlining. The default value is 100 which limits large function growth to 2.0 times the original size.

**large-function-insns**  
Specify limit for very large functions. For functions exceeding the specified limit, inlining is constrained by --param large-function-growth. This parameter is used primarily to avoid extreme compilation times. The default value is 2700.

**large-stack-frame**  
Specify limit for large stack frames. During inlining the algorithm attempts to not exceed the specified limit. The default value is 256 bytes.

**large-stack-frame-growth**  
Specify (as percent value) the maximal growth of large stack frames caused by inlining. The default value is 1000 (which limits large stack frame growth to 11 times the original size).

**large-unit-insns**  
Specify limit for large translation units. Growth caused by inlining of units beyond the specified limit is limited by --param inline-unit-growth. For small units this may be too tight. However, for very large units consisting of small inlineable functions, the overall unit growth limit is necessary to avoid exponential explosion of code size. Thus for smaller units the size is increased to --param large-unit-insns before applying --param inline-unit-growth. The default value is 10000

**max-average-unrolled-insns**  
The maximum number of instructions biased by probabilities of their execution that a loop should have if that loop is unrolled, and if the loop is unrolled, it determines how many times the loop code is unrolled.

**max-completely-peeled-insns**  
The maximum number of instructions of a completely peeled loop.

**max-completely-peel-times**  
The maximum number of iterations of a loop to be suitable for complete peeling.
**max-inline-insns-auto**
When `-finline-functions` is specified, many functions that would otherwise not be considered for inlining get considered for inlining. To those functions, specify a different, more restrictive limit than is imposed on functions declared inline. The default value is 100.

**max-inline-insns-recursive**
**max-inline-insns-recursive-auto**
Specify the maximum number of instructions that an out-of-line copy of a self-recursive inline function can grow into as a result of recursive inlining.

**max-inline-insns-rtl**
For languages that use the RTL inliner (which happens after tree inlining), set the maximum allowable size (counted in RTL instructions) for the RTL inliner with this parameter. The default value is 600.

**max-inline-insns-single**
Several parameters control the compiler’s tree inliner. This value sets the maximum number of instructions in a single function that the tree inliner will consider for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++). The default value is 500.

**max-inline-recursive-depth**
**max-inline-recursive-depth-auto**
Specify maximum recursion depth used by recursive inlining.

For functions declared inline, this parameter is active. For functions not declared inline, recursive inlining happens only when `-finline-functions` and parameter `max-inline-recursive-depth-auto` are enabled. The default value is 8.

**max-peeled-insns**
The maximum number of instructions that a loop should have if that loop is peeled, and if the loop is peeled, it determines how many times the loop code is peeled.

**max-peel-times**
The maximum number of peelings of a single loop.

**max-predicted-iterations**
The maximum number of loop iterations predicted statically. This is useful in cases where a given function contains a single loop with known bounds and other loops with unknown bounds. The known number of iterations can be predicted correctly, while in the unknown loops the average number of iterations is about 10. This causes the loop without bounds to appear artificially cold relative to the other one.

**max-unrolled-insns**
The maximum number of instructions that a loop should contain if that loop is unrolled. If the loop is unrolled, this value determines how many times the loop code is unrolled.

**max-unroll-times**
The maximum number of unrollings of a single loop.
**max-unswitch-insns**
The maximum number of instructions in an unswitched loop.

**max-unswitch-level**
The maximum number of branches unswitched in a single loop.

**min-crossjump-insns**
The minimum number of instructions that must be matched at the end of two blocks before cross-jumping is performed on them. This value is ignored when all instructions in the block being cross-jumped from are matched. The default value is 5.

**min-inline-recursive-probability**
Recursive inlining is useful only for functions having deep recursion in average, and can be harmful for functions having little recursion depth by increasing the prologue size or the complexity of the function body to other optimizers.

When profile feedback is available (see `-fprofile-generate`) the actual recursion depth can be estimated from the probability that the function will recurse via a given call expression. This parameter limits inlining only to call expressions whose probability exceeds the specified threshold (expressed as a percentage value). The default value is 10.

**min-spec-prob**
The minimum probability (expressed as a percentage value) of reaching a source block for interblock speculative scheduling. The default value is 40.

**min-vect-loop-bound**
The minimum number of iterations below which a loop is not vectorized when `-ftree-vectorize` is used. To enable vectorization the number of iterations after vectorization must be greater than the value specified by this option. The default value is 0.

**predictable-branch-cost-outcome**
When a branch is predicted to be taken with probability lower than the specified threshold (expressed as a percentage value), it is considered well-predictable. The default value is 10.

**reorder-blocks-duplicate**
**reorder-blocks-duplicate-feedback**
Used by the basic-block reordering pass to determine whether to use an unconditional branch or duplicate the code from its destination. Code is duplicated when its estimated size is smaller than the specified value multiplied by the estimated size of an unconditional jump in the hot spots of the program. `reorder-block-duplicate-feedback` is used only when profile feedback is available, and may be set to higher values than `reorder-block-duplicate` because information about the hot spots is more accurate.

**sched-spec-prob-cutoff**
Minimal probability of speculation success (expressed as percentage value), so that speculative instructions are scheduled. The default value is 40.

**sched-mem-true-dep-cost**
Minimal interval (in processor cycles) between a store and a load operation targeting the same memory location. The default value is 1.
ssp-buffer-size
The minimum buffer/array size that triggers the generation of buffer overflow checking when -fstack-protector is used. The default value is 8.

struct-reorg-cold-struct-ratio
The threshold ratio (expressed as a percentage value) between a structure frequency and the frequency of the hottest structure in the program. This parameter is enabled by -fipa-struct-reorg. If the ratio (structure frequency calculated by profiling) / (hottest structure frequency in program) is less than the specified value, structure reorganization is not applied to this structure. The default value is 10.

switch-conversion-max-branch-ratio
Switch initialization conversion will refuse to create arrays that are bigger than switch-conversion-max-branch-ratio times the number of branches in the switch.

use-canonical-types
Whether the compiler should use the “canonical” type system. By default, this should always be 1, which uses a more efficient internal mechanism for comparing types in C++. However, if bugs in the canonical type system are causing compilation failures, set this value to 0 to disable canonical types.

vect-max-version-for-alignment-checks
The maximum number of runtime checks that can be performed when doing loop versioning for alignment in the vectorizer. For more information see --ftree-vect-loop-version.

vect-max-version-for-alias-checks
The maximum number of runtime checks that can be performed when performing loop versioning for alias in the vectorizer. For more information see --ftree-vect-loop-version.

3.4.11 Math optimizations

The following options control compiler behavior regarding floating point arithmetic. These options trade off between speed and floating point accuracy.

NOTE Enabling these options can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO rules/specifications for math functions. Therefore, all of these options must be enabled explicitly; and they are never enabled by any of the -O options (Section 3.4.9).

-fassociative-math
Allow the re-association of operands in a series of floating-point operations. It may also reorder floating-point comparisons, and so cannot be used when ordered comparisons are required.

This option requires both -fno-signed-zeros and -fno-trapping-math to be enabled. It is not intended to be used with -frounding-math.

The default is -fno-associative-math.
NOTE
Operand reordering can change the sign of zero, as well as ignore NaNs and inhibit or create underflow or overflow. It thus cannot be used on code that relies on rounding behavior (for example, \((x + 2^{52}) - 2^{52}\)).

-`fcx-limited-range`
Eliminate the need for a range reduction step when performing complex division. No checking is performed to test whether the result of a complex multiplication or division is NaN + I*NaN.

This option controls the default setting of the ISO C99 CX_LIMITED_RANGE pragma.

The default is `-fno-cx-limited-range`.

-`ffast-math`
Enable the following options: `fcx-limited-range`, `ffinite-math-only`, `fno-math-errno`, `fno-rounding-math`, `fno-signaling-nans`, and `funsafe-math-optimizations`.

This option causes the preprocessor macro `__FAST_MATH__` to be defined.

-`ffinite-math-only`
Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-Infs.

The default is `-fno-finite-math-only`.

-`fno-math-errno`
Do not set ERRNO after calling math functions that are executed with a single instruction (e.g., `sqrt`).

A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.

The default is `-fmath-errno`.

-`freciprocal-math`
Allow the reciprocal of a value to be used instead of dividing by the value, in cases where doing this enables code optimizations. For example, replacing \(x / y\) with \(x * (1/y)\) can result in more optimized code if the expression \((1/y)\) is subject to common subexpression elimination.

This option decreases precision and increases the number of flops operating on the value.

The default is `-fno-reciprocal-math`. 
-frounding-math

Disable transformations and optimizations that assume default floating point rounding behavior. This is round-to-zero for all floating point to integer conversions, and round-to-nearest for all other arithmetic truncations. This option should be specified for programs that change the FP rounding mode dynamically, or that may be executed with a non-default rounding mode. This option disables constant folding of floating point expressions at compile-time (which may be affected by rounding mode) and arithmetic transformations that are unsafe in the presence of sign-dependent rounding modes.

The default is -fno-rounding-math.

NOTE This option is experimental and currently does not guarantee to disable all optimizations that are affected by rounding mode. Future versions of GCC may provide finer control of this setting using C99’s FENV_ACCESS pragma. This option will be used to specify the default state for FENV_ACCESS.

-fsignaling-nans

Compile code assuming that IEEE signaling NaNs may generate user-visible traps during floating-point operations. Setting this option disables optimizations that may change the number of exceptions visible with signaling NaNs. This option implies -ftrapping-math.

This option causes the preprocessor macro __SUPPORT_SNAN__ to be defined.

The default is -fno-signaling-nans.

NOTE This option is experimental and currently does not guarantee to disable all optimizations that affect signaling NaN behavior.

-fno-signed-zeros

Allow code optimizations for floating point arithmetic which ignore the signedness of zero. IEEE arithmetic specifies distinct +0.0 and -0.0 values which prohibit simplifying expressions such as x+0.0 or 0.0*x (even when -ffinite-math-only is enabled).

This option implies that the sign of a zero result is not significant.

The default is -fsigned-zeros.

-fsingle-precision-constant

Treat floating point constant as single precision constant instead of implicitly converting it to double precision constant.

-fno-trapping-math

Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation. This option implies -fno-signaling-nans. Setting this option may allow faster code if one relies on “non-stop” IEEE arithmetic, for example.

The default is -ftrapping-math.
-funsafe-math-optimizations
Allow code optimizations for floating-point arithmetic that (a) assume that the arguments and results are valid, and (b) may violate IEEE or ANSI standards.
When used at link time, this option may include libraries or startup files that change the default FPU control word or other similar optimizations.
This option enables -fno-signed-zeros, -fno-trapping-math, -fassociative-math, and -freciprocal-math.
The default is -fno-unsafe-math-optimizations.

3.4.12 Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.
If you use the -E option (Section 3.4.2), no translation is done except preprocessing. Some of these options make sense only together with -E because they cause the preprocessor output to be unsuitable for actual compilation.

-A pred=ans
Make an assertion with the predicate pred and answer ans.

NOTE This form is preferred to the older form -A pred(ans) (which is still supported) because it does not use shell special characters.

-A -pred=ans
Cancel an assertion with the predicate pred and answer ans.

-ansi
Same as -std=c89.

-C
Do not discard comments. All comments are passed through to the output file, except for comments in processed directives, which are deleted along with the directive.
You should be prepared for side effects when using -C; it causes the preprocessor to treat comments as tokens in their own right. For example, comments appearing at the start of what would be a directive line have the effect of turning that line into an ordinary source line, since the first token on the line is no longer a #.

-CC
Do not discard comments, including during macro expansion. This is like -C, except that comments contained within macros are also passed through to the output file where the macro is expanded.
In addition to the side-effects of the -C option, the -CC option causes all C++-style comments inside a macro to be converted to C-style comments. This is to prevent later use of that macro from inadvertently commenting out the remainder of the source line.
The -CC option is generally used to support lint comments.
-dCHARS

CHARS is a sequence of one or more of the characters listed below. No space can appear between the option name and characters. Any other characters specified are interpreted by the compiler proper, or are reserved for future versions of GCC, and so are silently ignored. If you specify any characters with conflicting behaviors the result is undefined.

M  Instead of the normal output, generate a list of #define directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor. Assuming you have no file foo.h, the command

    touch foo.h; cpp -dM foo.h

will show all the predefined macros.

D  Like M except in two respects: it does not include the predefined macros, and it outputs both the #define directives and the result of preprocessing. Both kinds of output go to the standard output file.

N  Like D, but emit only the macro names, not their expansions.

I  Output #include directives in addition to the result of preprocessing.

-D name
-D name=definition

Predefine name as macro with the specified definition. (The default definition value is 1.) The contents of definition are tokenized and processed as if they appeared in a #define directive. In particular, the definition will be truncated by embedded newline characters.

If you are invoking the preprocessor from a shell or shell-like program you may need to use the shell’s quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

If you wish to define a function-like macro on the command line, write its argument list with surrounding parentheses before the equals sign (if any). Parentheses are meaningful to most shells, so you will need to quote the option. With sh and csh, -D'name(args...)=definition' works.

-D and -U options are processed in the order they are given on the command line. All -imacros file and -include file options are processed after all -D and -U options.

-D and -U options must not be used to specify names that begin with an underscore character, as these names are reserved for use by the system.

-fdollars-in-identifiers

Accept $ in identifiers.
-fexec-charset=charset
   Set the execution character set, used for string and character constants. The
default is UTF-8. charset can be any encoding supported by the system’s iconv
library routine.

-finput-charset=charset
   Set the input character set, used for translation from the character set of the input
file to the source character set used by GCC. If the locale does not specify, or
GCC cannot get this information from the locale, the default is UTF-8. This can
be overridden by either the locale or this command line option. Currently the
command line option takes precedence if there’s a conflict. charset can be any
encoding supported by the system’s iconv library routine.

-fpch-deps
   When using precompiled headers (Section 3.10), this flag will cause the
dependency-output flags to also list the files from the precompiled header’s
dependencies. If not specified only the precompiled header would be listed and
not the files that were used to create it because those files are not consulted when
a precompiled header is used.

-fpreprocessed
   Indicate to the preprocessor that the input file has already been preprocessed. This
suppresses things like macro expansion, trigraph conversion, escaped newline
splicing, and processing of most directives. The preprocessor still recognizes and
removes comments, so that you can pass a file preprocessed with -C to the
compiler without problems. In this mode the integrated preprocessor is little more
than a tokenizer for the front ends.
   -fpreprocessed is implicit if the input file has one of the extensions .i, .ii or
   .mi. These are the extensions that GCC uses for preprocessed files created by
   -save-temps.

-fno-show-column
   Do not print column numbers in diagnostics. This may be necessary if diagnostics
are being scanned by a program that does not understand column numbers (such
as dejagnu).

-fstrict-overflow
-fno-strict-overflow
   Enforce strict language semantics for the following features:
   - Pointer arithmetic
   - Signed overflow
   This option is enabled at levels -O2, -O3, -Os. It can be disabled with -fno-
strict-overflow.

**Pointer arithmetic**

In C and C++ the strict semantics for pointer arithmetic is that when adding an
offset value to a pointer, the result is undefined if it does not produce a pointer to
the same object. This rule enables the compiler to assume that the expression
\( p + u > p \) is always true for the pointer \( p \) and unsigned integer \( u \). This is valid
only when pointer wraparound is undefined; otherwise, the expression would
return false if \( p + u \) overflows using two’s complement arithmetic.
Signed overflow

In C and C++ the strict semantics for signed overflow is that signed overflow does not exist. For example, consider the loop:

```c
for (i = 1; i > 0; i *= 2)
```

This code is presumably intended to loop until the index variable `i` overflows. If `-fstrict-overflow` is used, the compiler assumes that overflow does not occur, and transforms this into an infinite loop.

Using `-fstrict-overflow` enables various optimizations. For example, the compiler assumes that an expression such as `i + 10 > i` is always true for signed values. However, this is true only if signed overflow is undefined, because in two’s-complement arithmetic the expression yields false if `i + 10` overflows.

NOTE When `-fstrict-overflow` is enabled, any attempt to determine whether an operation on signed numbers will overflow must be coded carefully so as not to generate an overflow itself.

```
-fwrapv causes -fstrict-overflow and -fno-strict-overflow to be equivalent for integers, and also permits certain types of overflow.
```

```
-ftabstop=width
Set the distance between tab stops. This helps the preprocessor report correct column numbers in warnings or errors, even if tabs appear on the line. If the value is less than 1 or greater than 100, the option is ignored. The default is 8.
```

```
-fwide-exec-charset=charset
Set the wide execution character set, used for wide string and character constants. The default is UTF-32 or UTF-16, whichever corresponds to the width of wchar_t. As with -ftarget-charset, charset can be any encoding supported by the system’s `iconv` library routine; however, you will have problems with encodings that do not fit exactly in wchar_t.
```

```
-fworking-directory
Enable generation of line markers in the preprocessor output that will let the compiler know the current working directory at the time of preprocessing. When this option is enabled, the preprocessor will emit, after the initial line marker, a second line marker with the current working directory followed by two slashes. GCC will use this directory, when it’s present in the preprocessed input, as the directory emitted as the current working directory in some debugging information formats. This option is implicitly enabled if debugging information is enabled, but this can be inhibited with the negated form -fno-working-directory. If the -P flag is present in the command line, this option has no effect, since no #line directives are emitted whatsoever.
```

```
--help
Print out CPP’s version number at the beginning of execution, and report the final form of the include path. See also -v.
```
-H
  Print the name of each header file used, in addition to other normal activities.
  Each name is indented to show how deep in the #include stack it is. Precompiled
  header files are also printed, even if they are found to be invalid; an invalid
  precompiled header file is printed with . . .X and a valid one with . . .!

-I dir
  Add the directory dir to the list of directories to be searched for header files.
  Directories named by -I are searched before the standard system include
  directories. If the directory dir is a standard system include directory, the option
  is ignored to ensure that the default search order for system directories and the
  special treatment of system headers are not defeated.

-I-
  This option is deprecated: delete it and use -iquote for any -I directories before
  the deleted -I- option.

-idirafter dir
  Search dir for header files, but do it after all directories specified with -I and the
  standard system directories have been exhausted. dir is treated as a system include
  directory.

-imacros file
  Exactly like -include, except that any output produced by scanning file is
  thrown away. Macros it defines remain defined. This allows you to acquire all the
  macros from a header without also processing its declarations.

  All files specified by -imacros are processed before all files specified by
  -include.

-include file
  Process file as if #include "file" appeared as the first line of the primary
  source file. However, the first directory searched for file is the preprocessor’s
  working directory instead of the directory containing the main source file. If not
  found there, it is searched for in the remainder of the #include "..." search
  chain as normal.

  If multiple -include options are given, the files are included in the order they
  appear on the command line.

-iprefix prefix
  Specify prefix as the prefix for subsequent -iwithprefix options. If the prefix
  represents a directory, you should include the final /.

-isystem dir
  Search dir for header files, after all directories specified by -I but before the
  standard system directories. Mark it as a system directory, so that it gets the same
  special treatment as is applied to the standard system directories.

-iwithprefix dir
  Append dir to the prefix specified previously with -iprefix, and add the
  resulting directory to the include search path. -iwithprefixbefore puts it in the
  same place -I would; -iwithprefix puts it where -idirafter would.
-M

Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files, including those coming from -include or -imacros command line options.

Unless specified explicitly (with -MT or -MQ), the object file name consists of the basename of the source file with any suffix replaced with object file suffix. If there are many included files then the rule is split into several lines using \-newline. The rule has no commands.

This option does not suppress the preprocessor’s debug output, such as -dM. To avoid mixing such debug output with the dependency rules you should explicitly specify the dependency output file with -MF, or use an environment variable like DEPENDENCIES_OUTPUT (Section 3.9). Debug output will still be sent to the regular output stream as normal.

Passing -M to the driver implies -E, and suppresses warnings with an implicit -w.

-MD

-MD is equivalent to -M -MF file, except that -E is not implied. The driver determines file based on whether an -o option is given. If it is, the driver uses its argument but with a suffix of .d, otherwise it take the basename of the input file and applies a .d suffix.

If -MD is used in conjunction with -E, any -o switch is understood to specify the dependency output file (but see the -MF option below). However, if used without -E, each -o is understood to specify a target object file.

Since -E is not implied, -MD can be used to generate a dependency output file as a side-effect of the compilation process.

-MF file

When used with -M or -MM, specifies a file to write the dependencies to. If no -MF switch is given the preprocessor sends the rules to the same place it would have sent preprocessed output.

When used with the driver options -MD or -MMD, -MF overrides the default dependency output file.

-MG

In conjunction with an option such as -M requesting dependency generation, -MG assumes missing header files are generated files and adds them to the dependency list without raising an error. The dependency filename is taken directly from the #include directive without prepending any path. -MG also suppresses preprocessed output, as a missing header file renders this useless.

This feature is used in automatic updating of makefiles.
-MM
Like -M but do not mention header files that are found in system header directories, nor header files that are included, directly or indirectly, from such a header.

This implies that the choice of angle brackets or double quotes in an #include directive does not in itself determine whether that header will appear in -MM dependency output. This is a slight change in semantics from GCC versions 3.0 and earlier.

-MD
Same as -MD except mention only user header files, not system header files.

-MP
This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors make gives if you remove header files without updating the Makefile to match.

This is typical output:

test.o: test.c test.h

test.h:

-MQ target
Same as -MT, but it quotes any characters which are special to Make. -MQ
'$\$(objpfx)foo.o' gives:
$$(objpfx)foo.o: foo.c

The default target is automatically quoted, as if it were given with -MQ.

-MT target
Change the target of the rule emitted by dependency generation. By default CPP takes the name of the main input file, including any path, deletes any file suffix such as .c, and appends the platform’s usual object suffix. The result is the target.

An -MT option will set the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to -MT, or use multiple -MT options.

For example, -MT '$(objpfx)foo.o' might give:
$(objpfx)foo.o: foo.c

-nostdinc
Do not search the standard system directories for header files. Only the directories you have specified with -I options (and the directory of the current file, if appropriate) are searched.

-nostdinc++
Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building the C++ library.)
-o file
Write output to file. This is the same as specifying file as the second non-option argument to cpp. GCC has a different interpretation of a second non-option argument, so you must use -o to specify the output file.

-P
Inhibit generation of linemarkers in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code, and will be sent to a program which might be confused by the linemarkers.

-pedantic
Issue all the mandatory diagnostics listed in the C standard. Some of them are left out by default, since they trigger frequently on harmless code.

-pedantic-errors
Issue all the mandatory diagnostics, and make all mandatory diagnostics into errors. This includes mandatory diagnostics that GCC issues without -pedantic but treats as warnings.

-remap
Enable special code to work around file systems which only permit very short file names, such as MS-DOS.

-std=standard
Specify the standard to which the code should conform. Currently CPP knows about C and C++ standards; others may be added in the future.

standard may be one of:
iso9899:1990
c89
The ISO C standard from 1990. c89 is the customary shorthand for this version of the standard. See also -ansi.
iso9899:199409
iso9899:1999
c99
iso9899:199x
c9x
The revised ISO C standard published in December 1999. Before publication this was known as C9X.
gnu89
The 1990 C standard plus GNU extensions. This is the default.
gnu99
gnu9x
The 1999 C standard plus GNU extensions.
c++98
The 1998 ISO C++ standard plus amendments.
gnu++98
The same as -std=c++98 plus GNU extensions. This is the default for C++ code.
--target-help
Print text describing all the command line options instead of preprocessing anything.

-traditional-cpp
Try to imitate the behavior of old-fashioned C preprocessors, as opposed to ISO C preprocessors.

-trigraphs
Process trigraph sequences. These are three-character sequences, all starting with ??, that are defined by ISO C to stand for single characters. For example, ??/ stands for \, so '??/n' is a character constant for a newline. By default, GCC ignores trigraphs, but in standard-conforming modes it converts them. See the -std and -ansi options.

The nine trigraphs and their replacements are

<table>
<thead>
<tr>
<th>Trigraph:</th>
<th>??(  ??)  ??&lt;  ??&gt;  ??=  ??/  ??'  ??!  ??-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement:</td>
<td>[    ]    {    }    #    \    ^</td>
</tr>
</tbody>
</table>

-U name
Cancel any previous definition of name, either built in or provided with a -D option.

-undef
Do not predefined any system-specific or GCC-specific macros. The standard predefined macros remain defined.

-v
Verbose mode. Same as --help.

-version
--version
Print out the CPP version number:

✶ With one dash, proceed with preprocessing as normal.
✶ With two dashes, exit immediately.

-w
Suppress all warnings, including those which CPP issues by default.

-Wall
Enables all optional warnings which are desirable for normal code. At present this is -Wcomment, -Wtrigraphs, -Wmultichar and a warning about integer promotion causing a change of sign in #if expressions. Note that many of the preprocessor warnings are enabled by default and have no options to control them.

-Wcomment
-Wcomments
Warn whenever a comment-start sequence /* appears in a /* comment, or whenever a backslash-newline appears in a // comment. (Both forms have the same effect.)
-**-Wendif-labels**
Warn whenever an `#else` or an `#endif` are followed by text. This usually happens in code of the form:

```
#if FOO
...
#else FOO
...
#endif FOO
```

The second and third `FOO` should be in comments, but often are not in older programs. This warning is on by default.

-**-Werror**
Make all warnings into hard errors. Source code which triggers warnings will be rejected.

-**-Wimport**
Warn the first time `#import` is used.

-**-Wsystem-headers**
Issue warnings for code in system headers. These are normally unhelpful in finding bugs in your own code, therefore suppressed. If you are responsible for the system library, you may want to see them.

-**-Wtraditional**
Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and problematic constructs which should be avoided.

-**-Wtrigraphs**
Most trigraphs in comments cannot affect the meaning of the program. However, a trigraph that would form an escaped newline (??/ at the end of a line) can, by changing where the comment begins or ends. Therefore, only trigraphs that would form escaped newlines produce warnings inside a comment.

This option is implied by `-Wall`. If `-Wall` is not given, this option is still enabled unless trigraphs are enabled. To get trigraph conversion without warnings, but get the other `-Wall` warnings, use `-trigraphs -Wall -Wno-trigraphs`.

-**-Wundef**
Warn whenever an identifier which is not a macro is encountered in an `#if` directive, outside of `defined`. Such identifiers are replaced with zero.

-**-Wunused-macros**
Warn about macros defined in the main file that are unused. A macro is `used` if it is expanded or tested for existence at least once. The preprocessor will also warn if the macro has not been used at the time it is redefined or undefined.

Built-in macros, macros defined on the command line, and macros defined in include files are not warned about.
NOTE If a macro is actually used, but only used in skipped conditional blocks, then CPP will report it as unused. To avoid the warning in such a case, you might improve the scope of the macro’s definition by, for example, moving it into the first skipped block. Alternatively, you can provide a dummy use with something like:

```
#if defined the_macro_causing_the_warning
#endif
```

-x language
Specify the source language to be preprocessed.
The language parameter has the following possible values:
c
c++
asm

This option is not related to standards conformance or extensions; it merely selects which base syntax to expect. If this option is not specified, CPP deduces the language from the source file name extension (.c, .cc, or .s). Some other common extensions for C++ and assembly are also recognized. If CPP does not recognize the extension, it treats the file as C; this is the most generic mode.

NOTE Previous versions of CPP accepted a -lang option which selected both the language and the standards conformance level. This option has been removed because it conflicts with the -l option.

-Xpreprocessor option
Pass option as an option to the preprocessor. You can use this to supply system-specific preprocessor options which GCC does not know how to recognize.

If you want to pass an option that takes an argument, you must use
-Xpreprocessor twice: once for the option, and once for the argument.

See also -Wa (Section 3.4.2).

3.4.13 Assembly
Command options can be passed directly to the assembler.

-Xassembler option
Pass option as an option to the assembler. You can use this to supply system-specific assembler options which GCC does not know how to recognize.

If you want to pass an option that takes an argument, you must use -Xassembler twice: once for the option, and once for the argument.

See also -Wa (Section 3.4.2).
3.4.14 Linking

These options become relevant when the compiler links object files into an executable output file. They are meaningless if the compiler is not performing linking.

**NOTE** The options that start with `-no` do not have alternative positive forms.

`object_file_name`
A file name that does not end in a special recognized suffix is treated as specifying an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is performed, these object files are used as input to the linker.

`-c`
If this option is used then linking is not performed. In this case object file names should not be specified as arguments. See Section 3.8.

See also `-E` and `-S` below.

`-dynamic`
Direct the linker to use an available shared (.so) library instead of a static (.a) library. If the linker cannot find the shared version of a library, it uses the corresponding static library. By default the linker normally uses a static library instead of a shared library.

`-E`
If this option is used then linking is not performed. In this case object file names should not be specified as arguments. See Section 3.8.

`-l library`

Search the library named `library` when linking.

The linker searches and processes libraries and object files in the order they are specified on the command line. Thus, `foo.o -lz bar.o` searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `z`, those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named `liblibrary.a`. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with `-L`.

Normally the files found this way are library files: archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `-l` option and specifying a file name is that `-l` appends the prefix `lib` and suffix `.a` to the specified argument (`library`), and searches for the resulting file name in several directories.

**NOTE** The option form where the parameter is not separated from the option name is supported only for POSIX compliance. Its use is not recommended.
-nodefaultlibs
Do not use the standard system libraries when linking. Only the libraries you
specify will be passed to the linker. The standard startup files are used normally,
unless -nostartfiles is used. The compiler may generate calls to memcpy,
memset, and memcmp for System V (and ISO C) environments, or to bcopy and
bzero for BSD environments. These entries are usually resolved by entries in
libc. These entry points should be supplied through some other mechanism when
this option is specified.

-nostartfiles
Do not use the standard system startup files when linking. The standard system
libraries are used normally, unless -nostdlib or -nodefaultlibs is used.

-nostdlib
Do not use the standard system startup files or libraries when linking. No startup
files and only the libraries you specify will be passed to the linker. The compiler
may generate calls to memcpy, memset, and memcmp for System V (and ISO C)
environments, or to bcopy and bzero for BSD environments. These entries are
usually resolved by entries in libc. These entry points should be supplied through
some other mechanism when this option is specified.

One of the standard libraries bypassed by -nostdlib and -nodefaultlibs is
libgcc.a, a library of internal subroutines that GCC uses to overcome
shortcomings of particular machines, or special needs for some languages. In most
cases, you need libgcc.a even when you want to avoid other standard libraries.
In other words, when you specify -nostdlib or -nodefaultlibs you should
usually specify -lgcc as well. This ensures that you have no unresolved
references to internal GCC library subroutines. (For example, __main, used to
ensure that C++ constructors will be called.)

-pie
Produce a position-independent executable on targets that support it. For
predictable results you must also specify the same set of options that were used to
generate code (-fpie, -fPIE, or model sub-options) when you specify this
option.

-s
Remove all symbol table and relocation information from the executable.

-S
If this option is used then linking is not performed. In this case object file names
should not be specified as arguments. See Section 3.8.

See also -c and -E above.

-shared
Produce a shared object which can then be linked with other objects to form an
executable. Not all systems support this option. For predictable results you must
also specify the same set of options that were used to generate code (-fpic,
-fPIC, or model sub-options) when you specify this option.¹
-shared-libgcc

On systems that provide libgcc as a shared library, these options force the use of either the shared or static version respectively. If no shared version of libgcc was built when the compiler was configured, these options have no effect.

There are several situations in which an application should use the shared libgcc instead of the static version. The most common of these is when the application wishes to throw and catch exceptions across different shared libraries. In that case, each of the libraries as well as the application itself should use the shared libgcc. Therefore, the C++ compiler drivers automatically add -shared-libgcc whenever you build a shared library or a main executable, because C++ programs typically use exceptions, so this is the right thing to do.

If, instead, you use the compiler driver to create shared libraries, you may find that they will not always be linked with the shared libgcc. If GCC finds, at its configuration time, that you have a non-GNU linker or a GNU linker that does not support option --eh-frame-hdr, it will link the shared version of libgcc into shared libraries by default. Otherwise, it will take advantage of the linker and optimize away the linking with the shared version of libgcc, linking with the static version of libgcc by default. This allows exceptions to propagate through such shared libraries, without incurring relocation costs at library load time.

However, if a library or main executable is supposed to throw or catch exceptions, you must link it using the C++ compiler driver, as appropriate for the languages used in the program, or using the option -shared-libgcc, such that it is linked with the shared libgcc.

See also -static-libgcc.

-static

On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.

-static-libgcc

See -shared-libgcc.

-symbolic

Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option -Xlinker -z -Xlinker defs). Only a few systems support this option.

-u symbol

Pretend the symbol symbol is undefined, to force linking of library modules to define it. You can use -u multiple times with different symbols to force loading of additional library modules.

---

1 On some systems, hexagon-gcc -shared needs to build supplementary stub code for constructors to work. On multi-libbed systems, hexagon-gcc -shared must select the correct support libraries to link against. Failing to supply the correct flags may lead to subtle defects. Supposing them in cases where they are not necessary is innocuous.
**-Xlinker** option

Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GCC does not know how to recognize.

If you want to pass an option that takes an argument, you must use \texttt{-Xlinker} twice, once for the option and once for the argument. For example, to pass \texttt{-assert definitions}, you must write \texttt{-Xlinker -assert -Xlinker definitions}. It does not work to write \texttt{-Xlinker "-assert definitions"}, because this passes the entire string as a single argument, which is not what the linker expects.

See also \texttt{-Wl} (Section 3.4.2).

**-mG0lib**

Use the \texttt{-G 0} version of the standard system libraries when linking.

By default GCC uses the version of the standard system libraries that was compiled with \texttt{-G 8}.

This option is used in conjunction with the \texttt{-G} option (Section 3.4.18).

**-moslib=library**

Search the RTOS-specific library named \texttt{liblibrary.a} while linking an RTOS application.

Equivalent to:

- Specifying \texttt{-llibrary} (Section 3.4.14) before \texttt{-lc} in the linker options.
- Specifying \texttt{--start-group} and \texttt{--end-group} around \texttt{-llibrary}, \texttt{-lc}, and \texttt{-lgcc} in the linker options.

The default is standalone.

For more information on the linker options see the *Hexagon Binutils* document.

**NOTE** When compiling RTOS applications the library and include file search paths must be explicitly specified. For example:

\begin{verbatim}
  hexagon-gcc -Lrtos_path -Irtos_path -moslib=rtos_lib
\end{verbatim}
3.4.15 Directory search

These options specify directories to search for header files, for libraries, and for parts of the compiler.

-Bprefix

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms cpp, cc1, as and ld. It tries prefix as a prefix for each program it tries to run, both with and without machine/version/ (see Section 3.8).

For each subprogram to be run, the compiler driver first tries the -B prefix, if any. If that name is not found, or if -B was not specified, the driver tries the standard prefix, which is install-gnu/lib/gcc, where install is a configuration-specific path name. If the file name is not found, the unmodified program name is searched for using the directories specified in your PATH environment variable.

The compiler will check to see if the path provided by the -B refers to a directory, and if necessary it will add a directory separator character at the end of the path.

-B prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into -L options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into -isystem options for the preprocessor. In this case, the compiler appends include to the prefix.

The run-time support file libgcc.a can also be searched for using the -B prefix, if needed. If it is not found there, the standard prefix above is tried. If still not found, the file is not part of the link.

Another way to specify a prefix much like the -B prefix is to use the environment variable GCC_EXEC_PREFIX. See Section 3.9.

As a special kludge, if the path provided by -B is [dir/]stageN/, where N is a number in the range 0 to 9, then it will be replaced by [dir/]include. This is to help with boot-strapping the compiler.
-Idir
Add the directory dir to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. However, you should not use this option to add directories that contain vendor-supplied system header files (use -isystem for that). If you use more than one -I option, the directories are scanned in left-to-right order; the standard system directories come after.

If a standard system include directory, or a directory specified with -isystem, is also specified with -I, the -I option will be ignored. The directory will still be searched but as a system directory at its normal position in the system include chain. This ensures that GCC’s procedure to fix buggy system headers and the ordering for the include_next directive are not inadvertently changed. If you really need to change the search order for system directories, use the -nostdinc or -isystem options.

By default GCC searches the following directories for header files:

- ${INSTALL_DIR}/gnu/lib/gcc/target/version/include
- ${INSTALL_DIR}/gnu/target/include

... where target is hexagon-mt, and version is 3.4.6

NOTE mt is short for “multi-threaded”.

-I- Any directories you specify with -I options before the -I- option are searched only for the case of #include "file"; they are not searched for #include <file>.

If additional directories are specified with -I options after the -I-, these directories are searched for all #include directives. (Ordinarily all -I directories are used this way.)

In addition, -I- inhibits the use of the current directory (where the current input file came from) as the first search directory for #include "file". There is no way to override this effect of -I-. Specifying "-I." searches the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

-I- does not inhibit the use of the standard system directories for header files. Thus, -I- and -nostdinc are independent.

This option is deprecated – it is meant to be replaced by -iquote.

-Ldir
Add directory dir to the list of directories searched by the -l option (Section 3.4.14).

By default GCC searches the following directories for libraries:

- ${INSTALL_DIR}/gnu/lib/gcc/target/version
- ${INSTALL_DIR}/gnu/target/lib

... where target is hexagon-mt, and version is 3.4.6
-specs=file
Process the spec string file file after the compiler reads in the standard specs file, in order to override the defaults that the hexagon-cc driver program uses when determining what switches to pass to cc1, cc1plus, as, ld, etc. Multiple spec string files can be specified on the command line, and they are processed in order, from left to right.

For more information on spec string files, see Section 3.8.

### 3.4.16 Processor version

These machine-dependent options control the type of code generated by the compiler.

- **-march=archname**
- **-mcpu=archname**

Specify the target Hexagon processor architecture (i.e., the instruction set) of the output file. The possible values are hexagonv2, hexagonv3, or hexagonv4. The default is hexagonv2.

- **-mv2**
  Equivalent to: -march=hexagonv2

- **-mv3**
  Equivalent to: -march=hexagonv3

- **-mv4**
  Equivalent to: -march=hexagonv4

**NOTE** For more information on processor versions see the Hexagon V4 Programmer’s Reference Manual.

These options affect the definition of the processor-specific predefined symbols (Section 5.5.1).

### 3.4.17 Target machine

The usual way to run GCC is to simply run the executable file (e.g., hexagon-gcc). GCC also provides options that will switch to a different cross-compiler or version.

- **-b machine**

  The argument machine specifies the target machine for compilation.

  The value to use for machine is the same as was specified as the machine type when configuring GCC as a cross-compiler. For example, if a cross-compiler was configured with configure i386v, meaning to compile for an 80386 running System V, then you would specify -b i386v to run that cross compiler.
-V version

The argument version specifies which version of GCC to run. This is useful when multiple versions are installed. For example, version might be 2.0, meaning to run GCC version 2.0.

**NOTE**  The -b and -v options work by running the <machine>-hexagon-gcc-
<version> executable, so there's no real reason to use the options if you can just run the executable directly.

### 3.4.18 Code generation

These options control the interface conventions used in code generation.

- -fargument-alias
- -fargument-noalias
- -fargument-noalias-global

Specify the possible relationships among parameters and between parameters and global data:

- -fargument-alias specifies that arguments (parameters) may alias each other and may alias global storage.
- -fargument-noalias specifies that arguments do not alias each other, but may alias global storage.
- -fargument-noalias-global specifies that arguments do not alias each other and do not alias global storage.

Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.

- -fasynchronous-unwind-tables

Generate unwind table in DWARF2 format, if supported by target machine. The table is exact at each instruction boundary, so it can be used for stack unwinding from asynchronous events (such as debugger or garbage collector).

- -fcall-used-reg

Treat the register named reg as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register reg.

It is an error to used this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine’s execution model will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.
-fcall-saved-reg
Treat the register named reg as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register reg if they use it.

It is an error to use this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine’s execution model will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

-fno-common
In C, allocate even uninitialized global variables in the data section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without extern) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

-fno-exceptions
Disable exception handling. Do not generate the extra code needed to propagate exceptions. GCC by default generates frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution.

If you do not specify this option, GCC will by default enable exception handling for both C++ (which normally requires it) and C (which normally does not). You must not specify this option when compiling C++ code that performs exception handling, or when compiling C code that needs to interoperate with exception handlers written in C++.

-ffixed-reg
Treat the register named reg as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machinespecific and are defined in the REGISTER_NAMES macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

-fno-ident
Ignore the #ident directive.

-finhibit-size-directive
Don’t output a .size assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling crtstuff.c; you should not need to use it for anything else.
-finstrument-functions

Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site.

```c
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The first argument is the address of the start of the current function (which can be looked up in the list file symbol table).

A function can be assigned the attribute `no_instrument_function` to disable instrumentation. This feature can be used for the profiling functions listed above, for high-priority interrupt routines, or for any functions from which the profiling functions cannot safely be called (such as signal handlers, if the profiling routines generate output or allocate memory).

**NOTE** This option currently disables function inlining. This restriction is expected to be removed in future releases.

The function `__builtin_return_address` does not work beyond the current function, so the call site information may not be available to the profiling functions.

-fleading-underscore

This option and its counterpart, `-fno-leading-underscore`, forcibly change the way C symbols are represented in the object file. One use is to help link with legacy assembly code.

**NOTE** Code compiled with `fleading-underscore` is not binary-compatible with code compiled without this option. Use this option to conform to a non-default application binary interface.

-fnon-call-exceptions

Generate code that allows trapping instructions to throw exceptions. Note that this requires platform-specific runtime support that does not exist everywhere. Moreover, it only allows trapping instructions to throw exceptions, i.e. memory references or floating point instructions. It does not allow exceptions to be thrown from arbitrary signal handlers such as SIGALRM.
**-fpack-struct**

Pack all structure members together without holes. If a value is specified, pack the structure members according to the value, which represents the maximum alignment of the packed objects (and which must be a small power of two).

**NOTE** Any objects with alignment requirements larger than the specified value are output potentially unaligned at the next fitting location.

Code compiled with `-fpack-struct` is not binary-compatible with code compiled without this option. Additionally, it makes the code suboptimal. Use this option to conform to a non-default application binary interface.

**-fpcc-struct-return**

Return “short” `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between GCC-compiled files and files compiled with other compilers.

The precise convention for returning structures in memory depends on the target configuration macros.

Short structures and unions are those whose size and alignment match that of some integer type.

**NOTE** Code compiled with `-fpcc-struct-return` is not binary-compatible with code compiled with `-freg-struct-return`. Use this option to conform to a non-default application binary interface.

**-fpic**

Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of GCC; it is part of the operating system). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that `-fpic` does not work; in that case, recompile with `-fPIC` instead. (These maximums are 8k on the SPARC and 32k on the m68k and RS/6000. The 386 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines.

**-fPIC**

If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k and the SPARC.

Position-independent code requires special support, and therefore works only on certain machines.
-fpie
-ffPIE

These options are similar to -fpic and -fPIC, but generated position independent code can be only linked into executables. Usually these options are used when -pie GCC option will be used during linking.

-freg-struct-return
Return struct and union values in registers when possible. This is more efficient for small structures than -fpcc-struct-return.

If you specify neither -fpcc-struct-return nor -freg-struct-return, GCC defaults to whichever convention is standard for the target. If there is no standard convention, GCC defaults to -fpcc-struct-return, except on targets where GCC is the principal compiler. In those cases GCC can choose the standard, and it thus chooses the more efficient register-return alternative.

**NOTE** Code compiled with -freg-struct-return is not binary-compatible with code compiled with -fpcc-struct-return. Use this option to conform to a non-default application binary interface.

-fshared-data
Requests that the data and non-const variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

-fno-short-enums
Store all enumeration variables in 4 bytes of memory.

By default the Hexagon processor stores enumeration variables in the minimum number of bytes necessary to store the enumeration values (Section 4.10).

**NOTE** Code compiled with -fno-short-enums is not binary-compatible with code compiled without this option. Use this option to conform to a non-default application binary interface.

-fshort-double
Use the same size for double as for float.

**NOTE** Code compiled with -fshort-double is not binary-compatible with code compiled without this option. Use this option to conform to a non-default application binary interface.
**-fshort-wchar**

Override the underlying type for `wchar_t` to be `short unsigned int` instead of the default for the target. This option is useful for building programs to run under WINE.

**NOTE**  Code compiled with `-fshort-wchar` is not binary-compatible with code compiled without this option. Use this option to conform to a non-default application binary interface.

**-fstack-check**

Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

Note that this switch does not actually cause checking to be done; the operating system must do that. The switch causes generation of code to ensure that the operating system sees the stack being extended.

**-fstack-limit-register=reg**

**-fstack-limit-symbol=sym**

**-fno-stack-limit**

Generate code to ensure that the stack does not grow beyond a certain value, either the value of a register or the address of a symbol. If the stack would grow beyond the value, a signal is raised. For most targets, the signal is raised before the stack overruns the boundary, so it is possible to catch the signal without taking special precautions.

For instance, if the stack starts at absolute address 0x80000000 and grows downwards, you can use the flags `-fstack-limit-symbol=__stack_limit` and `-Wl,--defsym,__stack_limit=0x7ffe0000` to enforce a stack limit of 128KB. Note that this may only work with the GNU linker.

**-ftls-model=model**

Alter the thread-local storage model to be used (). The `model` argument should be `global-dynamic`, `local-dynamic`, `initial-exec` or `local-exec`.

The default without `-fpic` is `initial-exec`; with `-fpic` the default is `global-dynamic`.

**-ftrapv**

Generate traps for signed overflow on addition, subtraction, and multiplication operations.

**-funwind-tables**

Similar to `-fexceptions` except that it will just generate any needed static data, but will not affect the generated code in any other way. You will normally not enable this option; instead, a language processor that needs this handling would enable it on your behalf.
-fverbose-asm

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

-fno-verbose-asm, the default setting, causes the extra information to be omitted and is useful when comparing two assembler files.

-fvisibility=[default|internal|hidden|protected]

Set the default ELF image symbol visibility. Unless overridden in the code, all symbols are marked with the specified visibility. Using this option offers the following advantages:

❖ Substantially decrease the link and load times for shared libraries
❖ Generate better-optimized code
❖ Provide near-perfect API export
❖ Prevent symbol clashes

The use of this option is therefore strongly recommended in any shared objects.

The commonly-used settings for this option are default and hidden: the default setting default indicates that a symbol is public (i.e., it can be linked to from outside the shared object).

To add visibility support to existing code, #pragma GCC visibility can be used to enclose the declarations where you want to set visibility (Section 5.2.32).

Note that -fvisibility affects C++ vague linkage entities: any exception class thrown between DSOs must be marked as default so the type_info nodes are unified between the DSOs.

For information on visibility techniques see http://gcc.gnu.org/wiki/Visibility.

-fwrapv

Instruct the compiler to assume that the signed arithmetic overflow of addition, subtraction, and multiplication operations wraps around using two’s complement representation.

For example, if the compiler generates an overflow while performing constant arithmetic the overflowed result can still be used, but only when -fwrapv is used.

This option enables some optimizations and disables others.

-G size

Place data objects that are less than or equal to the specified size in the small data section (.sdata/.sbss). The size is expressed in bytes. The default is 8.

This option is used to perform global data optimization (Section 3.11.2).

If you compile an application file with “-G 0”:

❖ Be sure to also specify -mG0lib (Section 3.4.14)
❖ Be sure all the other files in the application are also compiled with “-G 0”

NOTE An application must be compiled with a -G value less than or equal to the smallest -G value that any of its referenced libraries was compiled with.
-mno-dot-new
Disable the generation of instructions that use the assembly language predicate .new (including conditionals and speculative jumps).

-mno-falign
Disable the generation of .falign directives. For more information on this directive see the Hexagon Binutils document.

Enabled at optimization levels -O0, -Os.

-mno-hardware-loops
Disable the generation of hardware loop instructions (by generating software loops instead).

-mliteral-intrinsics
Generate all intrinsics as the corresponding Hexagon processor instruction. By default GCC expands some intrinsics to multiple-instruction operations to enable optimization.

-mnoliteral-pool
Do not store program literals in the Hexagon processor’s global data area.

This option is used to perform global data optimization (Section 3.11.2).

-mnoliteral-pool-addresses
Generate references to variable addresses as immediate operands instead of literals (thus reducing the number of literals in the Hexagon processor’s global data area).

This option is used to perform global data optimization (Section 3.11.2).

-mv1-mv2-uncached-data
Generate code with instruction packets limited to containing single memory accesses; in addition, map all memcpy calls to memcpy_v (which is implemented using only instruction packets that contain single memory accesses).

This option is used to generate code that accesses uncached memory on the V2 processor (which does not support dual memory accesses to uncached memory).

NOTE
This option is functionally equivalent to declaring all relevant data with the type qualifier volatile.

This option is not applicable to the V3 and V4 processors (which do support dual memory accesses to uncached memory). To achieve the same effect in V3 or V4, use volatile as noted above.
3.5 Screen messages

The compiler normally operates silently from the command line without generating any standard screen messages. However, if list options are specified the compiler may generate an assembly listing to the terminal (Section 3.7).

**NOTE** Compiler information (including the compiler version) can be displayed on the screen with the **--version** option (Section 3.4.1).

3.6 Warning and error messages

The compiler writes any warning and error messages to the standard error file (which is usually your terminal):

- Warnings report non-critical events and conditions which do not prevent the compiler from generating an output file.
- Errors report serious problems which prevent the compiler from generating an output file.

Because the compiler may invoke the assembler and linker to compile a program, the warning and error messages displayed during compilation may originate from the compiler, assembler, or linker.

The messages have the following general format:

```
filename: [nnn: [cc:]] message_text
```

- `filename` indicates the source file that contains the problem. This is usually the current input file.
- `nnn` indicates the line number in the source file where the problem occurred. The line number does not appear if `filename` is an object file.
- `cc` indicates the column number on the source line where the problem occurred.

**NOTE** If a message refers to a symbol with the prefix `__builtin_` (e.g., `__builtin_vcmpwgt`), this indicates a problem with compiling one of the intrinsics (`Q6_p_vcmpw_gt_PP` in this case). For more information on intrinsics see the *Hexagon V4 Programmer’s Reference Manual*.

The compiler includes many options which control the display of warning messages; for more information see Section 3.4.6 and Section 3.4.7.
3.7 List files

To create an assembly list file of the compiled program, pass the assembler list option -a to the assembler from the compiler command line. For more information see Section 3.4.13 and the Hexagon Binutils document.

3.8 Spec files

GCC is a driver program: it does its job by invoking a sequence of other programs to do the work of compiling, assembling and linking. GCC interprets its command-line parameters and uses these to deduce which programs it should invoke, and which command-line options it ought to place on their command lines. This behavior is controlled by spec strings. In most cases there is one spec string for each program that GCC can invoke, but a few programs have multiple spec strings to control their behavior. The spec strings built into GCC can be overridden by using the -specs= command-line switch to specify a spec file.

Spec files are plaintext files that are used to construct spec strings. They consist of a sequence of directives separated by blank lines. The type of directive is determined by the first non-whitespace character on the line and it can be one of the following:

%command

Issues a command to the spec file processor. The commands that can appear here are:

%include <file>
Search for file and insert its text at the current point in the specs file.

%include_noerr <file>
Just like %include, but do not generate an error message if the include file cannot be found.

%rename old_name new_name
Rename the spec string old_name to new_name.

* [spec_name]:

This tells the compiler to create, override or delete the named spec string. All lines after this directive up to the next directive or blank line are considered to be the text for the spec string. If this results in an empty string then the spec will be deleted. (Or, if the spec did not exist, then nothing will happen.) Otherwise, if the spec does not currently exist a new spec will be created. If the spec does exist then its contents will be overridden by the text of this directive, unless the first character of that text is the + character, in which case the text will be appended to the spec.
[suffix]:
Creates a new [suffix] spec pair. All lines after this directive and up to the next directive or blank line are considered to make up the spec string for the indicated suffix. When the compiler encounters an input file with the named suffix, it will processes the spec string in order to work out how to compile that file. For example:

```
.ZZ:
z-compile -input %i
```

This says that any input file whose name ends in .zz should be passed to the program z-compile, which should be invoked with the command-line switch `-input` and with the result of performing the `%i` substitution. (See below.)

As an alternative to providing a spec string, the text that follows a suffix directive can be one of the following:

@language
This says that the suffix is an alias for a known language. This is similar to using the `-x` command-line switch to GCC to specify a language explicitly. For example:

```
.ZZ:
@c++
```

Says that .ZZ files are, in fact, C++ source files.

#name
This causes an error messages saying:

```
nname compiler not installed on this system.
```

GCC already has an extensive list of suffixes built into it. This directive will add an entry to the end of the list of suffixes, but since the list is searched from the end backwards, it is effectively possible to override earlier entries using this technique.

GCC has the following spec strings built into it. Spec files can override these strings or create their own. Note that individual targets can also add their own spec strings to this list.

- **asm**
  Options to pass to the assembler
- **asm_final**
  Options to pass to the assembler post-processor
- **cpp**
  Options to pass to the C preprocessor
- **ccl**
  Options to pass to the C compiler
- **cclplus**
  Options to pass to the C++ compiler
- **endfile**
  Object files to include at the end of the link
- **link**
  Options to pass to the linker
Here is a small example of a spec file:

```
%rename lib old_lib

*lib:
    --start-group -lgcc -lc -leval1 --end-group %(old_lib)
```

This example renames the spec called `lib` to `old_lib` and then overrides the previous definition of `lib` with a new one. The new definition adds in some extra command-line options before including the text of the old definition.

Spec strings are a list of command-line options to be passed to their corresponding program. In addition, the spec strings can contain `%`-prefixed sequences to substitute variable text or to conditionally insert text into the command line. Using these constructs it is possible to generate quite complex command lines.

Below is a list of all defined `%`-sequences for spec strings. Note that spaces are not generated automatically around the results of expanding these sequences. Therefore you can concatenate them together or combine them with constant text in a single argument.

- `%%` Substitute one `%` into the program name or argument.
- `%i` Substitute the name of the input file being processed.
- `%b` Substitute the basename of the input file being processed. This is the substring up to (and not including) the last period and not including the directory.
- `%B` This is the same as `%b`, but include the file suffix (text after the last period).
- `%d` Marks the argument containing or following the `%d` as a temporary file name, so the file will be deleted if GCC exits successfully. Unlike `%g`, this contributes no text to the argument.
%g_suffix
Substitute a file name that has suffix suffix and is chosen once per compilation, and mark the argument in the same way as %d. To reduce exposure to denial-of-service attacks, the file name is now chosen in a way that is hard to predict even when previously chosen file names are known. For example, %g.s ... %g.o ... %g.s might turn into ccUVUAU.s ccXYAXZ12.o ccUVUAU.s suffix matches the regular expression [.A-Za-z]* or the special string %O, which is treated exactly as if %O had been preprocessed. Previously, %g was simply substituted with a file name chosen once per compilation, without regard to any appended suffix (which was therefore treated just like ordinary text), making such attacks more likely to succeed.

%u_suffix
Like %g, but generates a new temporary file name even if %u_suffix was already seen.

%U_suffix
Substitutes the last file name generated with %u_suffix, generating a new one if there is no such last file name. In the absence of any %u_suffix, this is just like %g_suffix, except they don’t share the same suffix space, so %g.s ... %u.s ... %g.s ... %u.s would involve the generation of two distinct file names, one for each %g.s and another for each %u.s. Previously, %u was simply substituted with a file name chosen for the previous %u, without regard to any appended suffix.

%j_suffix
Substitutes the name of the HOST_BIT_BUCKET, if any, and if it is writable, and if save-temps is off; otherwise, substitute the name of a temporary file, just like %u. This temporary file is not meant for communication between processes, but rather as a junk disposal mechanism.

%|_suffix
%m_suffix
Like %g, except if -pipe is in effect. In that case %| substitutes a single dash and %m substitutes nothing at all. These are the two most common ways to instruct a program that it should read from standard input or write to standard output. If you need something more elaborate you can use an %{pipe:X} construct: see for example f/lang-specs.h.

%.SUFFIX
Substitutes .SUFFIX for the suffixes of a matched switch’s args when it is subsequently output with %*. SUFFIX is terminated by the next space or %.

%w
Marks the argument containing or following the %w as the designated output file of this compilation. This puts the argument into the sequence of arguments that %o will substitute later.

%o
Substitutes the names of all the output files, with spaces automatically placed around them. You should write spaces around the %o as well or the results are undefined. %o is for use in the specs for running the linker. Input files whose names have no recognized suffix are not compiled at all, but they are included among the output files, so they will be linked.
%O
Substitutes the suffix for object files. Note that this is handled specially when it immediately follows %g, %u, or %U, because of the need for those to form complete file names. The handling is such that %o is treated exactly as if it had already been substituted, except that %g, %u, and %U do not currently support additional suffix characters following %o as they would following, for example, .o.

%p
Substitutes the standard macro predefinitions for the current target machine. Use this when running cpp.

%p
Like %p, but puts __ before and after the name of each predefined macro, except for macros that start with __ or with __L, where L is an uppercase letter. This is for ISO C.

%I
Substitute any of -iprefix (made from GCC_EXEC_PREFIX), -isysroot (made from TARGET_SYSTEM_ROOT), and -isystem (made from COMPILER_PATH and -B options) as necessary.

%s
Current argument is the name of a library or startup file of some sort. Search for that file in a standard list of directories and substitute the full name found.

%e str
Print str as an error message. str is terminated by a newline. Use this when inconsistent options are detected.

%(name)
Substitute the contents of spec string name at this point.

%(name)
Like %(...) but put __ around -D arguments.

%x{option}
Accumulate an option for %x.

%X
Output the accumulated linker options specified by -Wl or a %x spec string.

%Y
Output the accumulated assembler options specified by -Wa.

%Z
Output the accumulated preprocessor options specified by -Wp.

%a
Process the asm spec. This is used to compute the switches to be passed to the assembler.

%A
Process the asm_final spec. This is a spec string for passing switches to an assembler post-processor, if such a program is needed.

%l
Process the link spec. This is the spec for computing the command line passed to the linker. Typically it will make use of the %L %G %S %D and %E sequences.
%D

Dump out a -L option for each directory that GCC believes might contain startup files. If the target supports multilibs then the current multilib directory will be prepended to each of these paths.

%M

Output the multilib directory with directory separators replaced with _. If multilib directories are not set, or the multilib directory is . then this option emits nothing.

%L

Process the lib spec. This is a spec string for deciding which libraries should be included on the command line to the linker.

%G

Process the libgcc spec. This is a spec string for deciding which GCC support library should be included on the command line to the linker.

%S

Process the startfile spec. This is a spec for deciding which object files should be the first ones passed to the linker. Typically this might be a file named crt0.o.

%E

Process the endfile spec. This is a spec string that specifies the last object files that will be passed to the linker.

%C

Process the cpp spec. This is used to construct the arguments to be passed to the C preprocessor.

%c

Process the signed_char spec. This is intended to be used to tell cpp whether a char is signed. It typically has the definition:

%funsigned-char:-D__CHAR_UNSIGNED__

%1

Process the cc1 spec. This is used to construct the options to be passed to the actual C compiler (cc1).

%2

Process the cc1plus spec. This is used to construct the options to be passed to the actual C++ compiler (cc1plus).

%*

Substitute the variable part of a matched option. See below. Note that each comma in the substituted string is replaced by a single space.

%<S

Remove all occurrences of -S from the command line. Note: this command is position dependent. % commands in the spec string before this one will see -S, % commands in the spec string after this one will not.
%:function(args)
Call the named function function, passing it args. args is first processed as a
nested spec string, then split into an argument vector in the usual fashion. The
function returns a string which is processed as if it had appeared literally as part of
the current spec.

The following built-in spec functions are provided:

- if-exists
- if-exists-else

The if-exists spec function takes one argument, an absolute pathname to a file.
If the file exists, if-exists returns the pathname. Here is a small example of its
usage:

*startfile:
crt0%O%s %:if-exists(crti%O%s) crtbegin%O%s

The if-exists-else spec function is similar to the if-exists spec function,
except that it takes two arguments. The first argument is an absolute pathname to a
file. If the file exists, if-exists-else returns the pathname. If it does not exist,
it returns the second argument. This way, if-exists-else can be used to select
one file or another, based on the existence of the first. Here is a small example of
its usage:

*startfile:
crt0%O%s %:if-exists(crti%O%s) \\
%:if-exists-else(crtbeginT%O%s crtbegin%O%s)

%{S}
Substitutes the -S switch, if that switch was given to GCC. If that switch was not
specified, this substitutes nothing. Note that the leading dash is omitted when
specifying this option, and it is automatically inserted if the substitution is
performed. Thus the spec string %{foo} would match the command-line option
-foo and would output the command line option -foo.

%w{S}
Like %{S} but mark last argument supplied within as a file to be deleted on
failure.

%{S*}
Substitutes all the switches specified to GCC whose names start with -S, but
which also take an argument. This is used for switches like -o, -D, -I, etc. GCC
considers -o foo as being one switch whose names starts with o. %{o*} would
substitute this text, including the space. Thus two arguments would be generated.

%{S*&T*}
Like %{S*}, but preserve order of S and T options (the order of S and T in the spec
is not significant). There can be any number of ampersand-separated variables; for
each the wild card is optional. Useful for CPP as %{D*&U*&A*}.

%{S:x}
Substitutes x, if the -S switch was given to GCC.

%{!S:x}
Substitutes x, if the -S switch was not given to GCC.
%{S*:x}
Substitutes x if one or more switches whose names start with -s are specified to GCC. Normally x is substituted only once, no matter how many such switches appeared. However, if %* appears somewhere in x, then x will be substituted once for each matching switch, with the %* replaced by the part of that switch that matched the *.

%{.S:x}
Substitutes x, if processing a file with suffix S.

%{1.S:x}
Substitutes x, if not processing a file with suffix S.

%{S|P:x}
Substitutes x if either -S or -P was given to GCC. This may be combined with !, ., and * sequences as well, although they have a stronger binding than the |. If %* appears in x, all of the alternatives must be starred, and only the first matching alternative is substituted.

For example, a spec string like this:

```bash
%{.c:-foo} %{!.c:-bar} %{.c|d:-baz} %{!.c|d:-boggle}
```

will output the following command-line options from the following input command-line options:

```bash
fred.c -foo -baz
jim.d -bar -boggle
-d fred.c -foo -baz -boggle
-d jim.d -bar -baz -boggle
```

%{S;T;Y; .; :D}
If S was given to GCC, substitutes x; else if T was given to GCC, substitutes y; else substitutes d. There can be as many clauses as you need. This may be combined with ., !, |, and * as needed.

The conditional text x in a %{S:x} or similar construct may contain other nested % constructs or spaces, or even newlines. They are processed as usual, as described above. Trailing white space in x is ignored. White space may also appear anywhere on the left side of the colon in these constructs, except between . or * and the corresponding word.

The -O, -f, -m, and -W switches are handled specifically in these constructs. If another value of -O or the negated form of a -f, -m, or -W switch is found later in the command line, the earlier switch value is ignored, except with {s*} where s is just one letter, which passes all matching options.

The character | at the beginning of the predicate text is used to indicate that a command should be piped to the following command, but only if -pipe is specified.

It is built into GCC which switches take arguments and which do not. (You might think it would be useful to generalize this to allow each compiler’s spec to say which switches take arguments. But this cannot be done in a consistent fashion. GCC cannot even decide which input files have been specified without knowing which switches take arguments, and it must know which input files to compile in order to tell which compilers to run).
GCC also knows implicitly that arguments starting in `-l` are to be treated as compiler output files, and passed to the linker in their proper position among the other output files.

### 3.9 GCC environment variables

This section describes several environment variables which affect how GCC operates. Some of them work by specifying directories or prefixes to use when searching for various kinds of files. Others are used to specify other aspects of the compilation environment.

The environment variables `LANG` and `LC*` specify how GCC uses localization information to work with different national conventions. They can be set to any value supported by your computer system.

**NOTE** The compiler search directories can be specified using command options (Section 3.4.15). The options override the environment variables, which in turn override the compiler’s default configuration.

#### CFLAGS (C only)
Command options which are passed to the compiler driver (`hexagon-gcc`).

#### C_INCLUDE_PATH
See `CPATH`.

#### COMPILER_DEFAULTS_PATH
Pathname (or multiple pathnames separated by colons) specifying where the compiler searches for the compiler defaults file (`compiler.defaults`).

The specified pathnames override the compiler’s default path name. Therefore if the defaults file is not found on the specified paths, the compiler will not use any defaults file.

#### COMPILER_PATH
The value of `COMPILER_PATH` is a colon-separated list of directories, much like `PATH`. GCC tries the directories thus specified when searching for subprograms, if it can’t find the subprograms using `GCC_EXEC_PREFIX`.

#### CPATH
`C_INCLUDE_PATH`

#### CPLUS_INCLUDE_PATH
`OBJC_INCLUDE_PATH`

Each variable’s value is a list of directories separated by a special character, much like `PATH`, in which to look for header files. The special character, `PATH_SEPARATOR`, is target-dependent and determined at GCC build time. For Microsoft Windows-based targets it is a semicolon, and for almost all other targets it is a colon.

`CPATH` specifies a list of directories to be searched as if specified with `-I`, but after any paths given with `-I` options on the command line. This environment variable is used regardless of which language is being preprocessed.
The remaining environment variables apply only when preprocessing the particular language indicated. Each specifies a list of directories to be searched as if specified with -isystem, but after any paths given with -isystem options on the command line.

In all these variables, an empty element instructs the compiler to search its current working directory. Empty elements can appear at the beginning or end of a path. For instance, if the value of CPATH is :/special/include, that has the same effect as ”-I. -I/special/include”.

**CPLUS_INCLUDE_PATH**
See CPATH.

**CXXFLAGS (C++ only)**
Command options which are passed to the compiler driver (hexagon-g++).

**DEPENDENCIES_OUTPUT**
If this variable is set, its value specifies how to output dependencies for Make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.

The value of **DEPENDENCIES_OUTPUT** can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form file target, in which case the rules are written to file using target as the target name.

In other words, this environment variable is equivalent to combining the options -MM and -MF (see Section 3.4.12) along with an optional -MT option.

**GCC_EXEC_PREFIX**
If **GCC_EXEC_PREFIX** is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

If **GCC_EXEC_PREFIX** is not set, GCC will attempt to figure out an appropriate prefix to use based on the pathname it was invoked with.

If GCC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

The default value of **GCC_EXEC_PREFIX** is prefix/local/hexagon/gnu/bin/ where prefix is the value of prefix when you ran the configure script.

Other prefixes specified with -B take precedence over this prefix.

This prefix is also used for finding files such as crt0.o that are used for linking.

In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with /usr/local/hexagon/gnu/bin (more precisely, with the value of **GCC_INCLUDE_DIR**), GCC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with -Bfoo/, GCC will search foo/bar where it would normally search /usr/local/hexagon/gnu/bar. These alternate directories are searched first; the standard directories come next.
LANG
Specifications locale information for the compiler. One way in which this information is used is to determine the character set to be used when character literals, string literals and comments are parsed in C and C++. When the compiler is configured to allow multibyte characters, the following values for LANG are recognized:

- **C-JIS** Recognize JIS characters.
- **C-SJIS** Recognize SJIS characters.
- **C-EUCJP** Recognize EUCJP characters.

If LANG is not defined, or if it has some other value, then the compiler will use mblen and mbtowc as defined by the default locale to recognize and translate multibyte characters.

Some additional environments variables affect the behavior of the preprocessor.

**LC_CTYPE**
Specify character classification. GCC uses this information to determine the character boundaries in a string; it is needed for some multibyte encodings that contain quote and escape characters that would otherwise be interpreted as a string end or escape.

**LC_MESSAGES**
The LC_MESSAGES environment variable specifies the language to use in diagnostic messages.

**LC_ALL**
If the LC_ALL environment variable is set, it overrides the value of LC_CTYPE and LC_MESSAGES; otherwise, LC_CTYPE and LC_MESSAGES default to the value of the LANG environment variable. If none of these variables are set, GCC defaults to traditional C English behavior.

**LIBRARY_PATH**
The value of LIBRARY_PATH is a colon-separated list of directories, much like PATH. When configured as a native compiler, GCC tries the directories thus specified when searching for special linker files, if it can’t find them using GCC_EXEC_PREFIX. Linking using GCC also uses these directories when searching for ordinary libraries for the -l option (but directories specified with -L come first).

**OBJC_INCLUDE_PATH**
See CPATH.

**SUNPRO_DEPENDENCIES**
Equivalent to DEPENDENCIES_OUTPUT except that system header files are not ignored; therefore it implies -M rather than -MM. However, the dependence on the main input file is omitted. See Section 3.4.12.

**TMPDIR**
Specify directory to use for temporary files. GCC uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.
3.10 Using precompiled headers

Often large projects have many header files that are included in every source file. The time the compiler takes to process these header files over and over again can account for nearly all of the time required to build the project. To make builds faster, GCC allows users to “precompile” a header file; then, if builds can use the precompiled header file they will be much faster.

NOTE In certain cases GCC is known to crash when trying to use a precompiled header. If you have trouble with a precompiled header, you should remove the precompiled header and compile without it.

To create a precompiled header file, simply compile it as you would any other file, if necessary using the -x option to make the driver treat it as a C or C++ header file. You will probably want to use a tool like `make` to keep the precompiled header up-to-date when the headers it contains change.

A precompiled header file will be searched for when `#include` is seen in the compilation. As it searches for the included file, the compiler looks for a precompiled header in each directory just before it looks for the include file in that directory. The name searched for is the name specified in the `#include` with `.gch` appended. If the precompiled header file can’t be used, it is ignored.

For instance, if you have `#include "all.h"`, and you have `all.h.gch` in the same directory as `all.h`, then the precompiled header file will be used if possible, and the original header will be used otherwise.

Alternatively, you might decide to put the precompiled header file in a directory and use `-I` to ensure that directory is searched before (or instead of) the directory containing the original header. Then, if you want to check that the precompiled header file is always used, you can put a file of the same name as the original header in this directory containing an `#error` command.

This also works with `-include`. So yet another way to use precompiled headers, good for projects not designed with precompiled header files in mind, is to simply take most of the header files used by a project, include them from another header file, precompile that header file, and `-include` the precompiled header. If the header files have guards against multiple inclusion, they will be skipped because they’ve already been included (in the precompiled header).

If you need to precompile the same header file for different languages, targets, or compiler options, you can instead make a directory named like `all.h.gch`, and put each precompiled header in the directory. (It doesn’t matter what you call the files in the directory, every precompiled header in the directory will be considered.) The first precompiled header encountered in the directory that is valid for this compilation will be used; they’re searched in no particular order.

There are many other possibilities, limited only by your imagination, good sense, and the constraints of your build system.
A precompiled header file can be used only when these conditions apply:

- Only one precompiled header can be used in a particular compilation.
- A precompiled header can’t be used once the first C token is seen. You can have preprocessor directives before a precompiled header; you can even include a precompiled header from inside another header, so long as there are no C tokens before the `#include`.
- The precompiled header file must be produced for the same language as the current compilation. You can’t use a C precompiled header for a C++ compilation.
- The precompiled header file must be produced by the same compiler version and configuration as the current compilation is using. The easiest way to guarantee this is to use the same compiler binary for creating and using precompiled headers.
- Any macros defined before the precompiled header (including with `-D`) must either be defined in the same way as when the precompiled header was generated, or must not affect the precompiled header, which usually means that the they don’t appear in the precompiled header at all.
- Certain command-line options must be defined in the same way as when the precompiled header was generated. At present, it’s not clear which options are safe to change and which are not; the safest choice is to use exactly the same options when generating and using the precompiled header.

For all of these but the last, the compiler will automatically ignore the precompiled header if the conditions aren’t met. For the last item, some option changes will cause the precompiled header to be rejected, but not all incompatible option combinations have yet been found.

### 3.11 Using code optimization

GCC provides extensive support for code optimization. This section explains how to use code optimization to improve program performance. It covers the following topics:

- Basic optimization
- Global data optimization
- Function inlining
- Dead function elimination
- Unused data removal
- Fast math

**NOTE** This document does not describe how specific code optimizations work. For more information on optimizations see the compiler reference books listed in Section 1.8.
3.11.1 Basic optimization

GCC supports the following code optimizations:

- Alias analysis
- Loop unrolling
- Strength reduction
- Common subexpression elimination
- Instruction scheduling

GCC includes a large number of compiler options which control these and other code optimizations (Section 3.4.10). You can bypass these options by using only the \texttt{-O} options (Section 3.4.9) to control the degree of optimization performed by the compiler.

The \texttt{-O} options accept a numeric argument (known as the \textit{option level}) which specifies the degree of code optimization to be performed:

- \texttt{-O0}: Perform no optimization
- \texttt{-O1}: Perform basic optimizations. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
- \texttt{-O2}: Perform nearly all supported optimizations that do not involve a space-speed trade-off.
- \texttt{-O3}: Optimize yet more. \texttt{-O3} enables all optimizations specified by \texttt{-O2}, along with function inlining and optimizations which improve code quality but make debugging impossible.

In addition to the numeric option levels \texttt{-O} accepts the following arguments:

- \texttt{-Os}: Perform all \texttt{-O2} optimizations that do not increase code size, and also enable some additional space-saving optimizations

The \texttt{-O} options specify default settings for the individual optimizations. In most cases they generate good-quality code, but in some cases program performance can be further improved by setting specific optimizations to their non-default settings.

3.11.2 Global data optimization

The Hexagon processor instruction set supports global-pointer (GP) relative addressing, which is used to reduce the code size of global data accesses.

The software development tools support this address mode by placing small global and static variables in the predefined data sections \texttt{.sdata} (small data) and \texttt{.sbss} (small bss - for small uninitialized data), and then generating code which uses the more compact GP-relative address instructions to access the data stored in these sections.

\textbf{NOTE} For more information on sections see the \textit{Hexagon Binutils} document.
GP-relative addressing has the limitation of supporting only 16 bits of unsigned scaled offset from the GP register. Large applications may contain more small global and static data than can be addressed in 16 bits. When this occurs the linker generates a “relocation overflow” error. The programmer must then find a way to reduce the amount of global data assigned to the small data sections.

Three compiler options exist to reduce the amount of global data stored in the small data sections:

- **-mno-literal-pool-addresses**
- **-mno-literal-pool**
- **-G**

- **-mno-literal-pool-addresses** (Section 3.4.18) generates references to variable addresses as immediate operands instead of literals. The resulting code is slightly larger overall, but uses less space in the small data sections (because the variable-address-reference literals, which are normally placed in the small data sections, no longer exist). This option is particularly effective when an application contains numerous character strings used in `printf` calls.

- **-mno-literal-pool** (Section 3.4.18) causes all program literals to not be placed in the small data sections.

- **-G** (Section 3.4.18) places data objects in the small data sections only if they are less than or equal to the specified size. This option is used to limit the number of global data items placed in the small data sections; in particular, to limit them to the smaller data items that don’t use up the available address space so quickly.

**Optimization strategy**

When a large application generates a relocation error overflow, the following three-step procedure is recommended:

1. Compile the application with option `-mno-literal-pool-addresses` and see if it eliminates the problem.
2. If step 1 does not work, compile the application with `-mno-literal-pool` instead.
3. If step 2 does not work, compile the application with “-G 0” (which specifies that no data is to be placed in the small data sections).

**-G limitations**

The “-G 0” option should be tried last because it has the following restrictions:

- All the files in the application must be compiled with “-G 0”
- All the files in the application must be linked with the “-G 0” version of the standard system libraries (using the `-mG0lib` option: see Section 3.4.14)

**NOTE** Failure to follow these restrictions may result in a “relocation overflow” error generated while linking.
3.11.3 Function inlining

Function inlining involves replacing a program’s function calls with complete copies of the function code. This improves execution performance by eliminating the extra code required to perform the function call.

Function inlining in GCC is controlled by the following options (which are defined in Section 3.4.10):

- `-finline`
- `-finline-functions`
- `-finline-limit`

For more information on inline functions (including the inline function attributes) see Section 5.2.35.

**NOTE** Function inlining is primarily a speed optimization: its effect on code size is unpredictable and depends on several factors.

Overly aggressive inlining can increase the code size to the point where program performance degrades from register spilling or cache swapping.

3.11.4 Dead function elimination

Dead function elimination involves identifying and deleting any functions that are not used in a program (including unused library functions). This reduces the program code size and thus the amount of memory necessary to execute the program.

Dead function elimination in GCC is controlled by the option `-ffunction-sections` (Section 3.4.10) and the linker option `--gc-sections` (*Hexagon Binutils* document).

Using these options causes the compiler and linker to perform dead function elimination:

- `-ffunction-sections` causes the compiler to create separate sections for each function in the program.
- `--gc-sections` causes the linker to remove all unused sections from the program.

**NOTE** `-ffunction-sections` must be used together with `--gc-sections` to perform dead function elimination.

Because it is a linker option `--gc-sections` must be preceded by the `-Xlinker` option (Section 3.4.14) when used in a compiler command.

The following function attributes (Section 5.2.25) can be used to selectively control dead function elimination:

- `unused` – The function is eligible for dead function elimination
- `used` – The function is not eligible for dead function elimination
3.11.5 Unused data removal

Unused data removal involves identifying and deleting any data sections that are not used in a program. This reduces the program code size and thus the amount of memory necessary to execute the program.

Unused data removal in GCC is controlled by the option `-fdata-sections` (Section 3.4.10) and the linker option `--gc-sections` (Hexagon Binutils document).

Using these options causes the compiler and linker to perform unused data removal:

- `-fdata-sections` causes the compiler to create separate sections for each data item in the program.
- `--gc-sections` causes the linker to remove all unused sections from the program.

**NOTE** `-fdata-sections` must be used together with the linker option `--gc-sections` to perform unused data removal.

Because it is a linker option `--gc-sections` must be preceded by the `-Xlinker` option (Section 3.4.14) when used in a compiler command.

The following variable attributes (Section 5.2.27) can be used to selectively control unused data removal:

- `unused` – The variable is eligible for unused data removal

3.11.6 Fast math

The fast math options offer increased performance of mathematical operations at the expense of possible decreases in accuracy or compatibility with numeric standards.

For more information on these options see Section 3.4.11.

- `-ffast-math`
- `-ffinite-math-only`
- `-fno-math-errno`
- `-frounding-math`
- `-fsignaling-nans`
- `-fsingle-precision-constant`
- `-fno-trapping-math`
- `-funsafe-math-optimizations`
3.12 Processor-specific features

GCC supports the following features which are specific to the Hexagon processor:

- -mG0lib, -moslib (Section 3.4.14)
- -march, -mcpu, -mv2, -mv3 (Section 3.4.16)
- -G, -mno-dot-new, -mno-falign, -mno-hardware-loops, -mliteral-intrinsics, -mnoliteral-pool, -mnoliteral-pool-addresses, -mv1-mv2-uncached-data (Section 3.4.18)
4 Language Implementation

4.1 Overview

A conforming implementation of ISO C is required to document its choice of behavior in each of the areas that are designated “implementation defined.”

This chapter describes each implementation-defined area (including the relevant sections from the ISO/IEC 9899:1999 standard) and how the area is implemented in GCC:

- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating point
- Arrays and pointers
- Hints
- Structures, unions, enumerations, and bit-fields
- Qualifiers
- Declarators
- Statements
- Preprocessing directives
- Library functions
- Architecture
- Locale-specific behavior

NOTE For more information on the application binary interface (ABI) see the Hexagon Application Binary Interface Specification.
4.2 Translation

- *How a diagnostic is identified (C90 3.7, C99 3.10, C90 and C99 5.1.1.3).*
  GCC writes all diagnostic messages to *stderr*. For details see Section 3.6.

- *Whether each nonempty sequence of whitespace characters other than new-line is retained or replaced by one space character in translation phase 3 (C90 and C99 5.1.1.2).*
  In text output, each whitespace sequence is collapsed to a single space. For readability the first token on each non-directive line of output is preceded with enough spaces to appear in the same column as it did in the original source file.

4.3 Environment

The behaviors covered by this section mostly depend on the implementation of the C library, and are thus not defined by GCC itself.

- *The mapping between physical source file multibyte characters and the source character set in translation phase 1 (C90 and C99 5.1.1.2).*
  GCC requires its input to be ASCII or UTF-8.

4.4 Identifiers

- *Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (C99 6.4.2).*
  C++ and C99 allow universal character names, and C99 further permits implementation-defined characters. GCC currently does not permit universal character names.

- *The number of significant initial characters in an identifier (C90 6.1.2, C90 and C99 5.2.4.1, C99 6.4.2).*
  All characters are significant in both internal and external names.

- *Whether case distinctions are significant in an identifier with external linkage (C90 6.1.2).*
  C99 requires that case distinctions are always significant in identifiers with external linkage. In GCC all identifiers are case-sensitive.
### 4.5 Characters

- **The number of bits in a byte (C90 3.4, C99 3.6).**
  A byte is 8 bits.

- **The values of the members of the execution character set (C90 and C99 5.2.1).**
  The members of the execution character set are assigned their standard ASCII and UTF-8 values.

- **The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90 and C99 5.2.2).**
  Escape sequences (such as `\a`) are assigned their standard ASCII values.

- **The value of a char object into which has been stored any character other than a member of the basic execution character set (C90 6.2.5, C99 6.2.5).**
  The value is determined by the character encoding in the source character set.

- **Which of signed char or unsigned char has the same range, representation, and behavior as “plain” char (C90 6.1.2.5, C90 6.2.1.1, C90 6.2.5, C99 6.3.1.1).**
  unsigned char is the same as char.

- **The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 6.4.4.4, C90 and C99 5.1.1.2).**
  The mapping is identical between the source and execution character sets.

- **The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 6.4.4.4).**
  Multi-character character constants are evaluated one character at a time, shifting the previous value left by the number of bits per target character, and then OR-ing in the bit pattern of the new character truncated to the width of a target character. The final bit pattern is given type int, and is therefore signed, regardless of whether single characters are signed or not. If more characters exist in the constant than can fit in the target int, the compiler generates a warning message and the excess leading characters are ignored.
  For example, 'ab' is interpreted as “(int) ((unsigned char) 'a' * 256 + (unsigned char) 'b')”; while '\234a' is interpreted as “(int) ((unsigned char) '\234' * 256 + (unsigned char) 'a')”.

- **The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 6.4.4.4).**
  The value is derived from the numeric value of each character.

- **The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.4, C99 6.4.5).**
  The locale is specified by the LANG, LC_ALL, or LC_CTYPE environment variable.
The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5).

The locale is specified by the LANG, LC_ALL, or LC_CTYPE environment variable.

The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 6.4.5).

Each byte of a multibyte character (or unrepresented escape sequence) represents a character in the string, with value equal to the numeric value of the byte.

4.6 Integers

Any extended integer types that exist in the implementation (C99 6.2.5).

GCC does not support extended integer types.

Whether signed integer types are represented using sign and magnitude, two’s complement, or one’s complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 6.2.6.2).

GCC supports only two’s complement integer types, and all bit patterns are ordinary values.

The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1).

GCC does not support extended integer types.

The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 6.3.1.3).

For conversion to a type of width N, the value is reduced to modulo $2^N$ to be within range of the type; no signal is raised.

The results of some bitwise operations on signed integers (C90 6.3, C99 6.5).

Bitwise operators act on the representation of the value including both the sign and value bits, where the sign bit is considered immediately above the highest-value value bit. Signed “>>>” acts on negative numbers by sign extension.

The sign of the remainder on integer division (C90 6.3.5).

GCC follows the C99 requirement that the result of division is truncated towards zero.
4.7 Floating point

- The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (C90 and C99 5.2.4.2.2).

Floating point operations and math functions are supported in 32-bit, 64-bit, 80-bit, and 128-bit IEC 60559 floating-point formats, and have been tested by the library vendor (Dinkumware) for accuracy and conformance to optional real and complex floating-point requirements.

Floating point accuracy is controlled by the FLT_EVAL_METHOD macro defined in float.h.

- The rounding behaviors characterized by non-standard values of FLT_ROUNDS (C90 and C99 5.2.4.2.2).

GCC does not use such values.

- The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (C99 5.2.4.2.2).

GCC does not use such values.

- The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 6.3.1.4).

C99 Annex F is followed.

- The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, C99 6.3.1.5).

C99 Annex F is followed.

- How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 6.4.4.2).

C99 Annex F is followed.

- Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (C99 6.5).

Expressions are currently contracted only if the options `-funsafe-math-optimizations` or `-ffast-math` are used. This is subject to change.

- The default state for the FENV_ACCESS pragma (C99 7.6.1).

This pragma is not implemented, but the default is to “off” unless the option `-frounding-math` is used in which case it is “on”.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 7.6,C99 7.12).

This is dependent on the C library and is not defined by GCC itself.
■ **The default state for the FP_CONTRACT pragma (C99 7.12.2).**

This pragma is not implemented. Expressions are currently contracted only if the options `-funsafe-math-optimizations` or `-ffast-math` are used. This is subject to change.

■ **Whether the “inexact” floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9).**

This is dependent on the C library and is not defined by GCC itself.

■ **Whether the “underflow” (and “inexact”) floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9).**

This is dependent on the C library and is not defined by GCC itself.

### 4.8 Arrays and pointers

■ **The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3).**

A cast from pointer to integer discards the most-significant bits if the pointer representation is larger than the integer type, sign-extends\(^1\) if the pointer representation is smaller than the integer type, or otherwise leaves the bits unchanged.

A cast from integer to pointer discards the most-significant bits if the pointer representation is smaller than the integer type, extends according to the signedness of the integer type if the pointer representation is larger than the integer type, or otherwise leaves the bits unchanged.

When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer; otherwise, the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 6.5.6 and 6.5.8.

■ **The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6).**

The value is defined as: \((p2 - p1) * \text{sizeof(element
type)}\). The result type is `ptrdiff_t`, which on the Hexagon processor is defined as `long int`.

---

\(^1\) Future versions of GCC may zero-extend, or use a target-defined `ptr_extend` pattern. Do not rely on sign extension.
4.9 Hints

- The extent to which suggestions made by using the register storage-class specifier are effective (C90 6.5.1, C99 6.7.1).

  The register specifier has no effect on the code generated by GCC.

- The extent to which suggestions made by using the inline function specifier are effective (C99 6.7.4).

  GCC will not inline any functions if the -fno-inline or -O0 options are used. Otherwise, GCC may still be unable to inline a function for many reasons. The -Wunused option can be used to determine if and why a function has not been inlined.

4.10 Structures, unions, enumerations, and bit-fields

- A member of a union is accessed using a member of a different type (C90 6.3.2.3).

  The relevant bytes of the representation of the object are treated as an object of the type used for the access.

- Whether a “plain” int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (C90 6.5.2, C90 6.5.2.1, C99 6.7.2, C99 6.7.2.1).

  By default it is treated as unsigned int but this can be changed by the option -fsigned-bitfields.

- Allowable bit-field types other than _Bool, signed int, and unsigned int (C99 6.7.2.1).

  No other types are permitted in strictly-conforming mode.

- Whether a bit-field can straddle a storage-unit boundary (C90 6.5.2.1, C99 6.7.2.1).

  A bit-field can straddle a storage-unit boundary (which is a byte on the Hexagon processor).

- The order of allocation of bit-fields within a unit (C90 6.5.2.1, C99 6.7.2.1).

  Bit-fields are allocated in a storage unit starting with the unit’s least-significant bits (i.e., little-endian).

- The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 6.7.2.1).

  Each member is aligned to its natural byte boundary (i.e., 1, 2, 4, or 8 bytes). Members are allocated in the order they are declared in the structure. The alignment of the structure is the same as the alignment of its largest member. If necessary, padding is inserted between member or after the last member to ensure that the structure size is an integral multiple of the storage unit size.
The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2).

If an enumeration contains negative values, its type is the first of the types signed char, short, and int that can represent all the values.

If an enumeration contains no negative values, its type is the first of the types unsigned char, unsigned short and unsigned int that can represent all the values.

4.11 Qualifiers

What constitutes an access to an object that has volatile-qualified type (C90 6.5.3, C99 6.7.3).

Such an object is normally accessed by pointers and used for accessing hardware. In most expressions, it is intuitively obvious what read and write accesses are. For example:

```c
volatile int *dst = somevalue;
volatile int *src = someothervalue;
*dst = *src;
```

This reads the volatile object pointed to by src and stores the value into the volatile object pointed to by dst. There is no guarantee that these reads and writes are atomic, especially for objects larger than int.

However, if the volatile storage is not being modified, and the value of the volatile storage is not used, then the situation is less obvious. For example:

```c
volatile int *src = somevalue;
*src;
```

According to the C standard, this expression is an rvalue whose type is the unqualified version of its original type, i.e., int. Whether GCC interprets this as a read of the volatile object being pointed to, or only as a request to evaluate the expression for its side-effects, depends on this type.

If it is a scalar type, or an aggregate type whose only member object is of a scalar type, or a union type whose member objects are of scalar types, the expression is interpreted by GCC as a read of the volatile object; in the other cases, the expression is evaluated only for its side-effects.

NOTE An access to a volatile variable cannot be paralleled with accesses to any other variable. This programming restriction is used to enforce a Hexagon processor restriction that data stored in uncached memory cannot be accessed in parallel with any other memory data. For more information see Section 5.5.2.
4.12 Declarators

- The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4).

  GCC is limited only by available memory.

4.13 Statements

- The maximum number of case values in a switch statement (C90 6.6.4.2).

  GCC is limited only by available memory.

4.14 Preprocessing directives

- How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 6.4.7).

  Source file characters are mapped to their ASCII values.

- Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 6.10.1).

  A character constant has the same numeric value whether it is in a preprocessing directive or any other expression.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 6.10.1).

  Character constants can have negative values in this case.

- The places that are searched for an included "" delimited header, and how the places are specified or the header is identified (C90 6.8.2, C99 6.10.2).

  GCC first searches the path `<install>`\usr\local\hexagon\qc\include\c. If the file is not found, it additionally searches any paths specified with the `-I` command option (Section 3.4.12).

- How the named source file is searched for in an included "" delimited header (C90 6.8.2, C99 6.10.2).

  GCC first searches the current directory. If the file is not found, it additionally searches any paths specified with the `-I` command option (Section 3.4.12). If the file is still not found, it searches the path `<install>`\usr\local\hexagon\qc\include\c.

- The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (C90 6.8.2, C99 6.10.2).

  All tokens in a header name (including any white space) are incorporated into the pathname used to search for a header file.
The nesting limit for \#include processing (C90 6.8.2, C99 6.10.2).

GCC imposes a limit of 200 nested \#include directives.

Whether the \# operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (C99 6.10.3.2).

GCC does not do this.

The behavior on each recognized non-STDC \#pragma directive (C90 6.8.6, C99 6.10.6).

The GCC-specific pragmas are described in Section 5.4.4.

The definitions for \_DATE\_ and \_TIME\_ when respectively, the date and time of translation are not available (C99 6.8.8, C99 6.10.8).

If the date and time are not available, \_DATE\_ expands to "??? ?? ????" and \_TIME\_ expands to "???:??:??".

### 4.15 Library functions

The behaviors described in this section are dependent on the C library, and are not defined by GCC itself. For more information see the Hexagon C Library User Guide and Hexagon C++ Library User Guide.

The null pointer constant to which the macro NULL expands (C90 7.1.6, C99 7.17).

NULL expands to 0.

### 4.16 Architecture

Detailed information for this section is contained in the Hexagon Application Binary Interface Specification.

The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, C99 7.18.3).

The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (C99 6.2.6.1).

The byte order is little-endian.

The value of the result of the sizeof operator (C90 6.3.3.4, C99 6.5.3.4).

The number of bytes the type occupies in memory after following all rules regarding layout, alignment, and padding.
4.17 Locale-specific behavior

The behaviors described in this section are dependent on the C library, and are not defined by GCC itself.
5 Language Extensions

5.1 Overview

The GCC compilers support several language features which are not defined in ISO standard C or C++. These features are known as language extensions.

The compilers support four types of extensions:

- General extensions available in both C and C++ (Section 5.2)
- Extensions specific to C (Section 5.3)
- Extensions specific to C++ (Section 5.4)
- Extensions specific to the Hexagon processor (Section 5.5)

The command option -pedantic (Section 3.4.3) can be used to direct the compilers to generate warning messages if any language extensions are used.

To test for the availability of these features in conditional compilation, check for the predefined macro __GNUC__, which is always defined in the compilers.

**NOTE** Some features that are in the ISO C99 standard, but not in the C89 or C++ standards, are also accepted as extensions in C89 mode and in C++. 
5.2 General extensions

The general language extensions are available in both C and C++.

5.2.1 Alternate keywords

-ansi and the various -std options disable certain keywords. This causes trouble when you want to use GNU C extensions, or a general-purpose header file that should be usable by all programs, including ISO C programs. The keywords asm, typeof, and inline are not available in programs compiled with -ansi or -std (although inline can be used in a program compiled with -std=c99). The ISO C99 keyword restrict is only available when -std=gnu99 (which will eventually be the default) or -std=c99 (or the equivalent -std=iso9899:1999) is used.

The way to solve these problems is to put __ at the beginning and end of each problematical keyword. For example, use __asm__ instead of asm, and __inline__ instead of inline.

Other C compilers will not accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keyword. It looks like this:

```c
#ifndef __GNUC__
#define __asm__ asm
#endif
```

-pedantic and other options cause warnings for many GNU C extensions. You can prevent such warnings within one expression by writing __extension__ before the expression. __extension__ has no effect aside from this.

5.2.2 Dollar signs in identifier names

In GNU C you can use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers. However, dollar signs in identifiers are not supported on some target machines, typically because the target assembler does not allow them.

5.2.3 <ESC> characters in constants

You can use the sequence \e in a string or character constant to represent the ASCII character <ESC>.
5.2.4 Looser rules for escaped newlines

In an escaped newline the preprocessor allows whitespace to appear between the backslash character and the subsequent newline. Whitespace includes spaces, horizontal and vertical tabs, and form feeds. The preprocessor issues a warning, but treats it as a valid escaped newline and combines the two lines to form a single logical line. This works in comments and tokens as well as between tokens. Note that comments themselves are not treated as whitespace in this case.

5.2.5 Statements and declarations in expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```c
({ int y = foo (); int z;
  if (y > 0) z = y;
  else z = - y;
  z; })
```

This is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type `void`, and thus effectively no value.)

This feature is especially useful in making macro definitions “safe” (so that they evaluate each operand exactly once). For example, the “maximum” function is commonly defined as a macro in standard C as follows:

```c
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either `a` or `b` twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here taken as `int`), you can define the macro safely as follows:

```c
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit-field, or the initial value of a static variable.

If you don’t know the type of the operand, you can still do this, but you must use `typeof` (Section 5.2.8).
In G++, the result value of a statement expression undergoes array and function pointer decay, and is returned by value to the enclosing expression. For example:

```c++
A a;
({a;}).Foo ()
```

Given that `A` is a class, GCC constructs a temporary “`A`” object which holds the result of the statement expression, and which will be used to invoke `Foo`. Therefore the `this` pointer observed by `Foo` will not be the address of `a`.

Any temporaries created within a statement expression will be destroyed at the statement’s end. This makes statement expressions inside macros slightly different from function calls. In the latter case temporaries introduced during argument evaluation will be destroyed at the end of the statement that includes the function call. In the statement expression case they will be destroyed during the statement expression. For example:

```c++
#define macro(a)  ({__typeof__(a) b = (a); b + 3; })
template<typename T> T function(T a) { T b = a; return b + 3; }
void foo ()
{
    macro (X ());
    function (X ());
}
```

In this example the temporaries are destroyed in different places:

- In the `macro` case temporary `X` will be destroyed just after the initialization of `b`.
- In the `function` case temporary `X` will be destroyed when the function returns.

These considerations mean that it is probably a bad idea to use statement-expressions of this form in header files that are designed to work with C++. (Note that some versions of the GNU C Library contained header files using statement-expressions that lead precisely to this bug.)

### 5.2.6 Locally declared labels

GCC allows you to declare local labels in any nested block scope. A local label is just like an ordinary label, but you can only reference it (with a `goto` statement, or by taking its address) within the block in which it was declared.

Local label declarations can have the following forms:

```c++
__label__ label;
__label__ label1, label2, /* ... */;
```

The declaration must appear at the beginning of a block, before any ordinary declarations or statements. It defines the label `name`, but does not define the label itself. You must do this in the usual way, with `label:` within the statements of the statement expression.
Local labels are useful for complex macros. If a macro contains nested loops, a goto can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```c
#define SEARCH(value, array, target)              
do {                                              
    __label__ found;                                
    typeof (target) _SEARCH_target = (target);     
    typeof (*(array)) * _SEARCH_array = (array);  
    int i, j;                                       
    int value;                                      
    for (i = 0; i < max; i++)                       
        for (j = 0; j < max; j++)                    
            if (_SEARCH_array[i][j] == _SEARCH_target)  
                { (value) = i; goto found; }              
    (value) = -1;                                   
    found:                                          
} while (0)
```

This can also be written using a statement-expression:

```c
#define SEARCH(array, target)                     
({                                                
    __label__ found;                                
    typeof (target) _SEARCH_target = (target);     
    typeof (*(array)) * _SEARCH_array = (array);  
    int i, j;                                       
    int value;                                      
    for (i = 0; i < max; i++)                       
        for (j = 0; j < max; j++)                    
            if (_SEARCH_array[i][j] == _SEARCH_target)  
                { value = i; goto found; }             
    value = -1;                                     
    found:                                           
    value;                                          
})
```

Local label declarations also make the labels they declare visible to nested functions, if there are any. See Section 5.3.1 for details.
5.2.7 Labels as values

You can access the address of a label defined in the current function (or a containing
function) with the unary operator &&. The value has type void *. This value is a constant
and can be used wherever a constant of that type is valid. For example:

```c
void *ptr;
/* ... */
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed
goto statement goto *expr. For example:

```c
goto *ptr;
```

Any expression of type void * is allowed.

One way of using these constants is in initializing a static array that will serve as a jump
table:

```c
static void *array[] = { &&foo, &&bar, &&hack };`n
```

Then you can select a label with indexing, like this:

```c
goto *array[i];
```

Note that this does not check whether the subscript is in bounds: array indexing in C never
does that.

Such an array of label values serves a purpose much like that of the switch statement.
The switch statement is cleaner, so use that rather than an array unless the problem does
not fit a switch statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the
interpreter function can be stored in the threaded code for super-fast dispatching.

You may not use this mechanism to jump to code in a different function. If you do that,
totally unpredictable things will happen. The best way to avoid this is to store the label
address only in automatic variables and never pass it as an argument.

An alternate way to write the above example is:

```c
static const int array[] = { &&foo - &&foo, &&bar - &&foo, &&hack - &&foo };
goto *(&&foo + array[i]);
```

This is more friendly to code stored in shared libraries, as it reduces the number of
dynamic relocations that are needed, and by consequence allows the data to be read-only.
5.2.8 Referring to a type with typeof

Another way to refer to the type of an expression is with typeof. The syntax of using of this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.

There are two ways of writing the argument to typeof: with an expression or with a type. Here is an example of an expression:

```c
typeof (x[0] (1))
```

This assumes that \( x \) is an array of pointers to functions; the type described is that of the values of the functions.

Here is an example with a type name as the argument:

```c
typeof (int *)
```

In this case the type described is that of pointers to int.

If you are writing a header file that must work when included in ISO C programs, write __typeof__ instead of typeof. See Section 5.2.1.

A typeof-construct can be used anywhere a typedef name can be used. For example, you can use it in a declaration, in a cast, or inside sizeof or typeof.

typeof is often useful with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type and evaluates each of its arguments exactly once:

```c
#define max(a,b) \
  ({ typeof (a) _a = (a); \
     typeof (b) _b = (b); \
     _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for \( a \) and \( b \). Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

More examples of typeof:

- This declares \( y \) with the type of what \( x \) points to:
  ```c
typeof (*x) y;
  ```

- This declares \( y \) as an array of such values:
  ```c
typeof (*x) y[4];
  ```
This declares \( y \) as an array of pointers to characters:
\[
\text{typeof (typeof (char *)[4]) } y;
\]

It is equivalent to the following traditional C declaration:
\[
\text{char *y[4];}
\]

To see the meaning of the declaration using \textit{typeof} (and why it might be a useful way to write), rewrite it with these macros:
\[
\begin{align*}
\text{#define pointer(T) typeof(T *)} \\
\text{#define array(T, N) typeof(T [N])}
\end{align*}
\]

Now the declaration can be rewritten this way:
\[
\text{array (pointer (char), 4) } y;
\]

Here is the corresponding type:
\[
\text{array (pointer (char), 4)}
\]

### 5.2.9 Inquiring on alignment of types or variables

The keyword \texttt{__alignof__} allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like \texttt{sizeof}.

For example, if the target machine requires a \texttt{double} value to be aligned on an 8-byte boundary, then \texttt{__alignof__ (double)} is 8. This is true on many RISC machines. On more traditional machine designs, \texttt{__alignof__ (double)} is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd address. For these machines, \texttt{__alignof__} reports the \textit{recommended} alignment of a type.

If the operand of \texttt{__alignof__} is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified with GCC’s \texttt{__attribute__} extension (see Section 5.2.27). For example:

\[
\text{struct foo { int x; char y; } fool;}
\]

The value of \texttt{__alignof__ (fool.y)} is 1, even though its actual alignment is probably 2 or 4, the same as \texttt{__alignof__ (int)}.

It is an error to ask for the alignment of an incomplete type.
5.2.10 Conditionals with omitted operands

The middle operand in a conditional expression can be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

For example:

\[ x ? : y \]

If \( x \) is nonzero this evaluates to \( x \); otherwise, it evaluates to \( y \).

This example is equivalent to:

\[ x ? x : y \]

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

5.2.11 Double-word integers

ISO C99 supports data types for integers that are at least 64 bits wide, and as an extension GCC supports them in C89 mode and in C++. Simply write \( \text{long long int} \) for a signed integer, or \( \text{unsigned long long int} \) for an unsigned integer. To make an integer constant of type \( \text{long long int} \), add the suffix \( \text{LL} \) to the integer. To make an integer constant of type \( \text{unsigned long long int} \), add the suffix \( \text{ULL} \) to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bit-wise Boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports a fullword-to-doubleword widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GCC.

There may be pitfalls when you use \( \text{long long} \) types for function arguments, unless you declare function prototypes. If a function expects type \( \text{int} \) for its argument, and you pass a value of type \( \text{long long int} \), confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects \( \text{long long int} \) and you pass \( \text{int} \). The best way to avoid such problems is to use prototypes.
5.2.12 Complex numbers

ISO C99 supports complex floating data types, and as an extension GCC supports them in C89 mode and in C++, and supports complex integer data types which are not part of ISO C99. You can declare complex types using the keyword _Complex. As an extension, the older GNU keyword __complex__ is also supported.

For example, _Complex double x; declares x as a variable whose real part and imaginary part are both of type double. _Complex short int y; declares y to have real and imaginary parts of type short int; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix i or j (either one; they are equivalent). For example, 2.5fi has type _Complex float and 3i has type _Complex int. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant. This is a GNU extension; if you have an ISO C99 conforming C library (such as GNU libc), and want to construct complex constants of floating type, you should include <complex.h> and use the macros ı or _Complex_I instead.

To extract the real part of a complex-valued expression exp, write __real__ exp. Likewise, use __imag__ to extract the imaginary part. This is a GNU extension; for values of floating type, you should use the ISO C99 functions crealf, creal, creall, cimagf, cimag and cimagl, declared in <complex.h> and also provided as built-in functions by GCC.

The operator - performs complex conjugation when used on a value with a complex type. This is a GNU extension; for values of floating type, you should use the ISO C99 functions conjf, conj and conjl, declared in <complex.h> and also provided as built-in functions by GCC.

GCC can allocate complex automatic variables in a noncontiguous fashion; it’s even possible for the real part to be in a register while the imaginary part is on the stack (or vice versa). Only the DWARF2 debug info format can represent this, so use of DWARF2 is recommended. If you are using the stabs debug info format, GCC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable’s actual name is foo, the two fictitious variables are named foo$real and foo$imag. You can examine and set these two fictitious variables with your debugger.
5.2.13 Hex floats

ISO C99 supports floating-point numbers written not only in the usual decimal notation, such as \(1.55e1\), but also numbers such as \(0x1.fp3\) written in hexadecimal format. As a GNU extension, GCC supports this in C89 mode (except in some cases when strictly conforming) and in C++. In that format the \(0x\) hex introducer and the \(p\) or \(P\) exponent field are mandatory. The exponent is a decimal number that indicates the power of 2 by which the significant part will be multiplied. Thus \(0x1.fp3\) is \(1 \frac{15}{16}\), \(p3\) multiplies it by 8, and the value of \(0x1.fp3\) is the same as \(1.55e1\).

Unlike for floating-point numbers in the decimal notation the exponent is always required in the hexadecimal notation. Otherwise the compiler would not be able to resolve the ambiguity of, e.g., \(0x1.f\). This could mean \(1.0f\) or \(1.9375\) since \(f\) is also the extension for floating-point constants of type float.

5.2.14 Binary constants

Integer constants can be written in binary format. Binary constants consist of a sequence of 0 and 1 digits prefixed by \(0b\) or \(0B\).

Binary constants follow the same rules as octal and hexadecimal integer constants, enabling the use of suffixes such as \(L\) or \(UL\).

5.2.15 Arrays of length zero

Zero-length arrays are allowed in GNU C. They are useful as the last element of a structure which is actually a header for a variable-length object:

```c
struct line {
  int length;
  char contents[0];
};
```

```c
struct line *thisline = (struct line *) malloc (sizeof (struct line) + this_length);
thisline->length = this_length;
```

In ISO C90, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

In ISO C99, you would use a flexible array member, which is slightly different in syntax and semantics:

- Flexible array members are written as `contents[]` without the 0.
- Flexible array members have incomplete type, and so the `sizeof` operator may not be applied. As a quirk of the original implementation of zero-length arrays, `sizeof` evaluates to zero.
- Flexible array members may only appear as the last member of a `struct` that is otherwise non-empty.
A structure containing a flexible array member, or a union containing such a structure (possibly recursively), may not be a member of a structure or an element of an array. (However, these uses are permitted by GCC as extensions.)

Non-empty initialization of zero-length arrays is treated like any case where there are more initializer elements than the array holds, in that a suitable warning about “excess elements in array” is given, and the excess elements (all of them, in this case) are ignored.

Instead, GCC allows static initialization of flexible array members. This is equivalent to defining a new structure containing the original structure followed by an array of sufficient size to contain the data. For instance, in the following example \texttt{f1} is constructed as if it were declared like \texttt{f2}.

```c
struct f1 {
    int x; int y[];
} f1 = { 1, { 2, 3, 4 } };

struct f2 {
    struct f1 f1; int data[3];
} f2 = { { 1 }, { 2, 3, 4 } };
```

The convenience of this extension is that \texttt{f1} has the desired type, eliminating the need to consistently refer to \texttt{f2.f1}.

This has symmetry with normal static arrays, in that an array of unknown size is also written with [].

Of course, this extension only makes sense if the extra data comes at the end of a top-level object, as otherwise we would be overwriting data at subsequent offsets. To avoid undue complication and confusion with initialization of deeply nested arrays, we simply disallow any non-empty initialization except when the structure is the top-level object. For example:

```c
struct foo { int x; int y[]; };
struct bar { struct foo z; };

struct foo a = { 1, { 2, 3, 4 } };  // valid
struct bar b = { { 1, { 2, 3, 4 } } };  // invalid
struct bar c = { { 1, { } } };  // valid
struct foo d[1] = { { 1 { 2, 3, 4 } } };  // invalid
```
5.2.16 Variable-length arrays

Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C89 mode and in C++. (However, GCC’s implementation of variable-length arrays does not yet conform in detail to the ISO C99 standard.) These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```c
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
    char str[strlen (s1) + strlen (s2) + 1];
    strcpy (str, s1);
    strcat (str, s2);
    return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing function returns. The space for a variable-length array is deallocated as soon as the array name’s scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

You can also use variable-length arrays as arguments to functions:

```c
struct entry
tester (int len, char data[len][len])
{
    /* ... */
}
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with `sizeof`.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list (another GNU extension). For example:

```c
struct entry
tester (int len; char data[len][len], int len)
{
    /* ... */
}
```

The `int len` before the semicolon is a `parameter forward declaration`, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.
You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the “real” parameter declarations. Each forward declaration must match a “real” declaration in parameter name and data type. ISO C99 does not support parameter forward declarations.

### 5.2.17 Macros with a variable number of arguments

In the ISO C standard of 1999, a macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. For example:

```c
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

Here `...` is a variable argument. In the invocation of such a macro, it represents the zero or more tokens until the closing parenthesis that ends the invocation, including any commas. This set of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. See the CPP manual for more information.

GCC has long supported variable-argument macros, and used a different syntax that allowed you to give a name to the variable arguments just like any other argument. For example:

```c
#define debug(format, args...) fprintf (stderr, format, args)
```

This is equivalent to the ISO C example above, but arguably more readable and descriptive.

GNU CPP has two further variable-argument macro extensions, and permits them to be used with either of the above forms of macro definition.

In standard C, you are not allowed to leave the variable argument out entirely; but you are allowed to pass an empty argument. For example, this invocation is invalid in ISO C, because there is no comma after the string:

```c
debug ("A message")
```

GNU CPP permits you to completely omit the variable arguments in this way. In the above examples, the compiler would complain, though since the expansion of the macro still has the extra comma after the format string.

To help solve this problem, CPP behaves specially for variable arguments used with the token paste operator, `##`. For example:

```c
#define debug(format, ...) fprintf (stderr, format, ## __VA_ARGS__)
```

If the variable arguments in this example are omitted or empty, the `##` operator causes the preprocessor to remove the comma before it. If you do provide some variable arguments in your macro invocation, GNU CPP does not complain about the paste operation and instead places the variable arguments after the comma. Just like any other pasted macro argument, these arguments are not macro expanded.
5.2.18  Non-Lvalue arrays may have subscripts

In ISO C99, arrays that are not lvalues still decay to pointers, and may be subscripted, although they may not be modified or used after the next sequence point and the unary & operator may not be applied to them. As an extension, GCC allows such arrays to be subscripted in C89 mode, though otherwise they do not decay to pointers outside C99 mode. For example, the following code is valid in GNU C but not in C89:

```c
struct foo {int a[4];};
struct foo f();
bar (int index)
{
  return f().a[index];
}
```

5.2.19  Arithmetic on void- and function-pointers

In GNU C, addition and subtraction operations are supported on pointers to void and on pointers to functions. This is done by treating the size of a void or of a function as 1.

A consequence of this is that sizeof is also allowed on void and on function types, and returns 1.

The option -Wpointer-arith requests a warning if these extensions are used.

5.2.20  Compound literals

ISO C99 supports compound literals. A compound literal looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer; it is an lvalue. As an extension, GCC supports compound literals in C89 mode and in C++.

Usually, the specified type is a structure. Assume that struct foo and structure are declared as shown:

```c
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a struct foo with a compound literal:

```c
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```c
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```
You can also construct an array. If all the elements of the compound literal are (made up of) simple constant expressions, suitable for use in initializers of objects of static storage duration, then the compound literal can be coerced to a pointer to its first element and used in such an initializer, as shown here:

```c
char **foo = (char *[]) { "x", "y", "z" }; 
```

Compound literals for scalar types and union types are also allowed, but then the compound literal is equivalent to a cast.

As a GNU extension, GCC allows initialization of objects with static storage duration by compound literals (which is not possible in ISO C99, because the initializer is not a constant). It is handled as if the object was initialized only with the bracket enclosed list if compound literal’s and object types match. The initializer list of the compound literal must be constant. If the object being initialized has array type of unknown size, the size is determined by compound literal size:

```c
static struct foo x = (struct foo) {1, 'a', 'b'};
static int y[] = (int []) {1, 2, 3};
static int z[] = (int [3]) {1};
```

The above lines are equivalent to the following:

```c
static struct foo x = {1, 'a', 'b'};
static int y[] = {1, 2, 3};
static int z[] = {1, 0, 0};
```

### 5.2.21 Case ranges

You can specify a range of consecutive values in a single `case` label. For example:

```c
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from `low` to `high`, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```c
case 'A' ... 'Z':
```

**NOTE** The `...` symbol must be delimited by spaces; otherwise, when used with integer values it may be parsed incorrectly by the compiler. For example:

```c
case 1 ... 5:       // OK
case 1...5:         // WRONG - no spaces
```
5.2.22 Cast to a union type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with `union` tag or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See Section 5.2.20.)

The types that may be cast to the union type are those of the members of the union. For example:

```c
union foo { int i; double d; };  
int x;                     
double y;                 
```

In this case both `x` and `y` can be cast to type `union foo`.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```c
union foo u;                   
/* ... */                     
u = (union foo) x == u.i = x   
u = (union foo) y == u.d = y   
```

You can also use the union cast as a function argument:

```c
void hack (union foo);         
/* ... */                     
hack ((union foo) x);          
```

5.2.23 Unnamed struct/union fields within structs/unions

For compatibility with other compilers, GCC allows you to define a structure or union that contains, as fields, structures and unions without names. For example:

```c
struct {                     
  int a;                     
  union {                   
    int b;                   
    float c;                 
  };                        
  int d;                     
} foo;                       
```

In this example, the user would be able to access members of the unnamed union with code like `foo.b`. Note that only unnamed structs and unions are allowed, you may not have, for example, an unnamed `int`. 
You must never create such structures that cause ambiguous field definitions. For example:

```c
struct { // WRONG
  int a;
  struct {
    int a;
  };
} foo;
```

It is ambiguous which `a` is being referred to with `foo.a`. Such constructs are not supported and must be avoided. In the future, such constructs may be detected and treated as compilation errors.

### 5.2.24 Attributes

In GNU C you can declare functions, types, and variables with `attributes` that provide the compiler with various types of information about the declared item. For example:

```c
int old_func (void) __attribute__ ((deprecated));   // func
int count __attribute__ ((aligned (16))) = 0;       // var
struct symbol __attribute__ ((packed))              // type
{ ... };
```

Attributes are declared by specifying the keyword `__attribute__` after a standard item declaration. The keyword is followed by an `attribute list`, which consists of one or more comma-separated attribute names enclosed in double parentheses.

The attributes are defined in the following sections:

- Function attributes ([Section 5.2.25](#function-attributes))
- Type attributes ([Section 5.2.26](#type-attributes))
- Variable attributes ([Section 5.2.27](#variable-attributes))

The attribute syntax is described in detail in [Section 5.2.28](#attribute-syntax).
5.2.25 Function attributes

In GNU C you can declare functions with attributes that provide the compiler with various types of information about the declared function. For example:

```c
extern void foo (void) __attribute__((section ("ext")));
```

Among other things, attribute information enables the compiler to optimize function calls and check your code more carefully.

Function attributes are declared by specifying the keyword `__attribute__` after a function declaration. The keyword is followed by an attribute list, which consists of one or more comma-separated attribute names enclosed in double parentheses.

The following attributes are currently defined for functions: `noreturn`, `noinline`, `always_inline`, `pure`, `const`, `nothrow`, `format`, `format_arg`, `no_instrument_function`, `section`, `constructor`, `destructor`, `used`, `unused`, `deprecated`, `weak`, `malloc`, `alias`, `warn_unused_result` and `nonnull`.

Attributes can be declared with double underscore characters (i.e., `__`) delimiting the attribute name as well as the keyword `attribute`. For example:

```c
void fatal () __attribute__((__noreturn));
```

This enables you to declare attributes in header files without having to worry about a macro possibly having the same name as the attribute.

For a detailed description of the attribute syntax see Section 5.2.28.

NOTE Some attributes (such as `section`) can be applied to types and variables as well as functions.

**alias ("target")**

Cause the declaration to be emitted as an alias for another symbol, which must be specified. For example:

```c
void __f () { /* Do something. */; }
void f () __attribute__(((weak, alias ("__f"))));
```

This declares `f` to be a weak alias for `__f`. In C++, the mangled name for the target must be used.

NOTE The target symbol must be defined in the current translation unit.
alloc_size (size1 [, size2])
Inform the compiler that the function return value points to memory, with the memory size specified by one or two of the function’s parameters. The compiler uses this information to support __builtin_object_size.

The memory size is conventionally defined as either the value of the first memory size parameter, or the product of the two memory size parameters.

size1 and size2 specify the positions of the memory-size parameters in the function’s parameter list. The value 1 denotes the first parameter in the list.

For example, in the following declarations my_calloc is specified as having two memory size parameters, while my_realloc has only one:

void* my_calloc(size_t, size_t) __attribute__((alloc_size(1,2)))
void my_realloc(void*, size_t) __attribute__((alloc_size(2)))

always_inline
Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.

const
Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just a slightly more strict class than the pure attribute (listed below), since the function is not allowed to read global memory.

Note that a function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a non-const function usually must not be const. It does not make sense for a const function to return void.

constructor
Cause the function to be called automatically before execution enters main (). Similarly, the destructor attribute causes the function to be called automatically after main() has completed or exit() has been called.

Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

deprecated
Generate a warning if the function is used anywhere in the source file. This is useful when identifying functions that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated function, to enable users to easily find further information about why the function is deprecated, or what they should do instead. Note that the warnings occur only for uses.
For example:

```c
int old_fn () __attribute__((deprecated));
int old_fn ();
int (*fn_ptr)() = old_fn;
```

This results in a warning on line 3 but not on line 2.

The `deprecated` attribute can also be used for variables and types (see Section 5.2.27 and Section 5.2.26.)

**Destructor**

See constructor.

**dllexport**

On Microsoft Windows targets cause the compiler to provide a global pointer to a pointer in a DLL, so that it can be referenced with the `dllimport` attribute. The pointer name is formed by combining `_imp_` and the function or variable name.

Currently, the `dllexport` attribute is ignored for inlined functions, but export can be forced by using the `-fkeep-inline-functions` flag. The attribute is also ignored for undefined symbols.

When applied to C++ classes the attribute marks defined non-inlined member functions and static data members as exports. Static consts initialized in-class are not marked unless they are also defined out-of-class.

On Cygwin, mingw, and arm-pe targets, `__declspec(dllexport)` is recognized as a synonym for `__attribute__((dllexport))` for compatibility with other Microsoft Windows compilers.

Alternative methods for including the symbol in the DLL export table are to use a `.def` file with an `EXPORTS` section or, with GNU ld using the `--export-all` option.

**dllimport**

On Microsoft Windows targets cause the compiler to reference a function or variable via a global pointer to a pointer that is set up by the Microsoft Windows dynamic link library (DLL). The pointer name is formed by combining `_imp_` and the function or variable name. The attribute implies `extern` storage.

Currently, the attribute is ignored for inlined functions. If the attribute is applied to a symbol `definition`, an error is reported. If a symbol previously declared `dllimport` is later defined, the attribute is ignored in subsequent references and a warning is emitted. The attribute is also overridden by a subsequent declaration as `dllexport`.

When applied to C++ classes, the attribute marks non-inlined member functions and static data members as imports. However, the attribute is ignored for virtual methods to allow creation of vtables using thunks.

On Cygwin, mingw and arm-pe targets, `__declspec(dllimport)` is recognized as a synonym for `__attribute__((dllimport))` for compatibility with other Microsoft Windows compilers.
The use of the `dllimport` attribute on functions is not necessary, but provides a small performance benefit by eliminating a thunk in the DLL. The use of the `dllimport` attribute on imported variables was required on older versions of GNU ld, but can now be avoided by passing the `--enable-auto-import` switch to ld. As with functions, using the attribute for a variable eliminates a thunk in the DLL.

One drawback to using this attribute is that a pointer to a function or variable marked as `dllimport` cannot be used as a constant address. The attribute can be disabled for functions by setting the `-mnop-fun-dllimport` flag.

**format** (archetype, string-index, first-to-check)

Specify that a function takes `printf`, `scanf`, `strftime` or `strfmon` style arguments which should be type-checked against a format string. For example:

```c
extern int my_printf (void *my_object, const char *my_format, ...)
    __attribute__ ((format (printf, 2, 3)));
```

This causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter `archetype` determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime` or `strfmon`. (You can also use `__printf__`, `__scanf__`, `__strftime__` or `__strfmon__`.) The parameter `string-index` specifies which argument is the format string argument (starting from 1), while `first-to-check` is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency. For `strftime` formats, the third parameter is required to be zero. Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two, not one, when giving values for `string-index` and `first-to-check`.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions which take format strings as arguments, so that GCC can check the calls to these functions for errors. The compiler always (unless `-ffreestanding` is used) checks formats for the standard library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `strftime`, `vprintf`, `vfprintf` and `vscanf` whenever such warnings are requested (using `-Wformat`), so there is no need to modify the header file `stdio.h`. In C99 mode, the functions `snprintf`, `vsnprintf`, `vscanf`, `vfscanf` and `vsscanf` are also checked. Except in strictly conforming C standard modes, the X/Open function `strfmon` is also checked as are `printf_unlocked` and `fprintf_unlocked`. See Section 3.4.3.
**format_arg** *(string-index)*

Specify that a function takes a format string for a `printf`, `scanf`, `strftime` or `strfmon` style function and modifies it (for example, to translate it into another language), so the result can be passed to a `printf`, `scanf`, `strftime` or `strfmon` style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). For example:

```c
extern char *
my_dgettext (char *my_domain, const char *my_format)
__attribute__ ((format_arg (2)));
```

This causes the compiler to check the arguments in calls to a `printf`, `scanf`, `strftime` or `strfmon` type function, whose format string argument is a call to the `my_dgettext` function, for consistency with the format string argument `my_format`. If the `format_arg` attribute had not been specified, all the compiler can tell in such calls to format functions would be that the format string argument is not constant; this would generate a warning when `-Wformat-nonliteral` is used, but the calls can not be checked without the attribute.

The parameter *string-index* specifies which argument is the format string argument (starting from one). Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two.

The `format-arg` attribute allows you to identify your own functions which modify format strings, so that GCC can check the calls to `printf`, `scanf`, `strftime` or `strfmon` type function whose operands are a call to one of your own function. The compiler always treats `gettext`, `dgettext`, and `dcgettext` in this manner except when strict ISO C support is requested by `-ansi` or an appropriate `-std` option, or `-ffreestanding` is used. See Section 3.4.3.

**malloc**

Inform the compiler that a function can be treated as if any non-`NULL` pointer that it returns will not alias any other currently valid pointer. This often improves optimization. Standard functions with this property include `malloc` and `calloc`. `realloc`-like functions have this property as long as the old pointer is never referred to (including comparing it to the new pointer) after the function returns a non-`NULL` value.

**noinline**

Prevent a function from being considered for inlining.

**no_instrument_function**

If the command option `-finstrument-functions` is specified, profiling function calls will be generated at entry and exit of most user-compiled functions. Functions with this attribute will not be so instrumented.
nonnull (arg-index, ...)  
Specify that some function parameters should be non-null pointers. For example:

```c
extern void *
my_memcpy (void *dest, const void *src, size_t len)
__attribute__((nonnull (1, 2)));
```

This causes the compiler to check that, in calls to `my_memcpy`, arguments `dest` and `src` are non-null. If the compiler determines that a null pointer is passed in an argument slot marked as non-null, and the `-Wnonnull` option is enabled, a warning is issued. The compiler may also choose to make optimizations based on the knowledge that certain function arguments will not be null.

If no argument index list is given to the `nonnull` attribute, all pointer arguments are marked as non-null. To illustrate, the following declaration is equivalent to the previous example:

```c
extern void *
my_memcpy (void *dest, const void *src, size_t len)
__attribute__((nonnull));
```

noreturn

A few standard library functions, such as `abort` and `exit`, cannot return. GCC knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example:

```c
void fatal () __attribute__ ((noreturn));

void
fatal (/* ... */)
{
    /* ... */ /* Print error message. */ /* ... */
    exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

The `noreturn` keyword does not affect the exceptional path when that applies: a `noreturn`-marked function may still return to the caller by throwing an exception.

Do not assume that registers saved by the calling function are restored before calling the `noreturn` function.

A `noreturn` function cannot have a return type other than `void`.

The attribute `noreturn` is not implemented in GCC versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```c
typedef void voidfn ();

volatile voidfn fatal;
```
nothrow
Inform the compiler that a function cannot throw an exception. For example, most functions in the standard C library can be guaranteed not to throw an exception with the notable exceptions of `qsort` and `bsearch` that take function pointer arguments.

optimize
The `optimize` attribute is used to specify that a function is to be compiled with different optimization options than are specified on the command line. Arguments can either be numbers or strings. Numbers are assumed to be an optimization level. Strings that begin with `O` are assumed to be an optimization option, while other options are assumed to be used with a `-f` prefix. You can also use the `pragma GCC optimize` (Section 5.2.34) to set the optimization options that affect more than one function. This attribute can be used, for instance, to have frequently executed functions compiled with more aggressive optimization options that produce faster and larger code, while other functions can be called with less aggressive options.

pure
Many functions have no effects except the return value, and their return value depends only on the parameters and/or global variables. Such functions can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `pure`.

For example:

```c
int square (int) __attribute__ ((pure));
```

This specifies that the hypothetical function `square` is safe to call fewer times than the program says.

Some common examples of pure functions are `strlen` or `memcmp`. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as `feof` in a multithreading environment).

section ("section-name")
By default the compiler places the code it generates in the `.text` section of the output file. However, to support programs with nonstandard memory configurations, you may need to assign functions to another section. The `section` attribute specifies that a function is assigned to a particular section. For example:

```c
extern void foo (void) __attribute__ ((section ("ext")));
```

This assigns function `foo` to a section named `ext`.

GCC includes predefined sections which are used to assign functions to the Hexagon processor memories (Section 5.5.2). It also supports user-defined sections. For more information on sections see the `Hexagon Binutils` document.

**NOTE** To map the entire contents of a module to a particular section, consider using the facilities of the linker instead of the `section` attribute.
sentinel [(position)]

Ensure that the specified parameter in a variadic function call is explicitly set to NULL. The optional argument position specifies the position of the parameter in the parameter list, counting backwards from the end of the argument list. The default argument value is 0, which specifies the last parameter in the list.

For example:

```c
__attribute__((sentinel))
```

is equivalent to:

```c
__attribute__((sentinel(0)))
```

NULL is defined as the value zero with any pointer type. (If defined on your system as an integer type, NULL will require an explicit cast to be used in this attribute. GCC replaces file stddef.h with a copy which redefines NULL appropriately.)

**NOTE** Sentinels are valid only with variadic functions.

Warnings for missing or incorrect sentinels are enabled with the `-Wformat` option.

unused

Specify that the function is meant to be possibly unused. GCC will not produce a warning for this function.

used

Specify that code must be emitted for the function even if it appears that the function is not referenced. This is useful, for example, when the function is referenced only in inline assembly.

visibility ("visibility_type")

Cause the declaration to be emitted with the specified visibility (default, hidden, protected, or internal).

```c
void __attribute__((visibility ("protected")))
  f () { /* Do something. */; }
int i __attribute__((visibility ("hidden")));
```
See the ELF gABI for complete details. Here is a summary:

**warn_unused_result**

Cause a warning to be generated if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as realloc. For example:

```c
int fn () __attribute__((warn_unused_result));
int foo ()
{
    if (fn () < 0) return -1;
    fn ();
    return 0;
}
```

This results in a warning on line 5.

**default**

Default visibility is the normal case for ELF. This value is available for the visibility attribute to override other options that may change the assumed visibility of symbols.

**hidden**

Hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so no other module (executable or shared library) can reference it directly.

**protected**

Protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind to the local symbol. That is, the symbol cannot be overridden by another module.

**internal**

Internal visibility is like hidden visibility, but with additional processor specific semantics. GCC defines internal visibility to mean that the function is *never* called from another module. Note that hidden symbols, while they cannot be referenced directly by other modules, can be referenced indirectly via function pointers. By indicating that a symbol cannot be called from outside the module, GCC may for instance omit the load of a PIC register since it is known that the calling function loaded the correct value.
weak

Cause the declaration to be emitted as a weak symbol rather than a global symbol. A weak symbol is a symbol definition which can be overridden by a global symbol definition.

Weak symbols differ from global symbols in two ways. First, defining a weak symbol with the same name as a global symbol does not cause a link error; instead, the linker uses the global symbol definition and ignores the weak symbol. Second, when the linker searches library archive files (Section 3.4.14), it will not extract archive members in order to resolve any undefined weak symbols (as it does for global symbols). If the linker cannot find any definition for a weak symbol, it sets the symbol value to zero.

Weak symbols are primarily useful in defining library functions that can be overridden in user code, or for determining whether a specific object module is present in an image.

NOTE

A global symbol intended to override a weak symbol must be defined in one of the object files explicitly specified as linker input files.

Because weak symbols can evaluate to zero, calls to them may result in a link error if the target value exceeds the address range supported by the call instruction. Calling the symbol through a pointer eliminates this problem.

Weak symbols are intended for use in system software. Applications that use weak symbols are unreliable, because changes in the runtime environment may cause an otherwise working application to fail with an execution error.

5.2.26 Type attributes

In GNU C you can declare types with attributes that provide the compiler with various types of information about the declared type. For example:

```c
struct symbol __attribute__ ((packed)) {
    ... }
```

Type attributes are declared by specifying the keyword `__attribute__` after a type declaration. The keyword is followed by an attribute list, which consists of one or more comma-separated attribute names enclosed in double parentheses.

The following attributes are currently defined for types: aligned, packed, transparent_union, unused, deprecated and may_alias.

Attributes can be declared with double underscore characters (i.e., `__`) delimiting the attribute name as well as the keyword `attribute`. For example:

```c
typedef int T1 __attribute__((__deprecated__));
```

This enables you to declare attributes in header files without having to worry about a macro possibly having the same name as the attribute.
For a detailed description of the attribute syntax see Section 5.2.28.

**NOTE** Some attributes (such as section) can be applied to functions and variables as well as types.

- **aligned** *(alignment)*
  Specify a minimum alignment (in bytes) for variables of the specified type. For example:

  ```c
  struct S { short f[3]; } __attribute__((aligned (8)));
typedef int more_aligned_int __attribute__((aligned (8)));
  ```

  These declarations force the compiler to ensure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned at least on a 8-byte boundary. On a SPARC, having all variables of type `struct S` aligned to 8-byte boundaries allows the compiler to use the `ldd` and `std` (doubleword load and store) instructions when copying one variable of type `struct S` to another, thus improving run-time efficiency.

  Note that the alignment of any given `struct` or `union` type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. Thus you can effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

  As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can omit the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example:

  ```c
  struct S { short f[3]; } __attribute__((aligned (8)));
  ```

  Whenever you omit the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

  In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.
Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

deprecated

Generate a warning if the type is used anywhere in the source file. This is useful when identifying types that are expected to be removed in a future version of a program. If possible, the warning also includes the location of the declaration of the deprecated type, to enable users to easily find further information about why the type is deprecated, or what they should do instead. Note that the warnings only occur for uses and then only if the type is being applied to an identifier that itself is not being declared as deprecated. For example:

```c
typedef int T1 __attribute__((deprecated));
T1 x;
typedef T1 T2;
T2 y;
typedef T1 T3 __attribute__((deprecated));
T3 z __attribute__((deprecated));
```

This results in a warning on line 2 and 3 but not lines 4, 5, or 6. No warning is issued for line 4 because `T2` is not explicitly deprecated. Line 5 has no warning because `T3` is explicitly deprecated. Similarly for line 6.

The `deprecated` attribute can also be used for functions and variables (see Section 5.2.25 and Section 5.2.27).
may_alias

Accesses to objects with types with this attribute are not subjected to type-based alias analysis, but are instead assumed to be able to alias any other type of objects, just like the char type.

For example:

define short __attribute__((__may_alias__)) short_a;

int
main (void)
{
    int a = 0x12345678;
    short_a *b = (short_a *) &a;

    b[1] = 0;

    if (a == 0x12345678)
        abort();

    exit(0);
}

If you replaced short_a with short in the variable declaration, the above program would abort when compiled with -fstrict-aliasing, which is on by default at -O2 or above.

For more information on aliasing issues see option -fstrict-aliasing (Section 3.4.9).

packed

When attached to struct or union type definition, specify that each member of the structure or union is placed to minimize the memory required. When attached to an enum definition, it indicates that the smallest integral type should be used.

Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members. Specifying the -fshort-enums option on the command line is equivalent to specifying the packed attribute on all enum definitions.

In the following example struct my_packed_struct’s members are packed closely together, but the internal layout of its s member is not packed: to do that, struct my_unpacked_struct would also need to be packed:

struct my_unpacked_struct
{
    char c;
    int i;
};

struct my_packed_struct __attribute__((__packed__))
{
    char c;
    int i;
    struct my_unpacked_struct s;
};
NOTE  packed can be specified only on the definition of a enum, struct or union, and not on a typedef which does not also define the enumerated type, structure or union.

transparent_union
When attached to a union type definition, specify that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like const on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of the first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the wait function must accept either a value of type int * to comply with Posix, or a value of type union wait * to comply with 4.1 BSD. If the wait parameter were void *, then wait would accept both kinds of arguments; however, it would also accept any other pointer type, and this would make argument type checking less useful. Instead, <sys/wait.h> might define the interface as follows:

```
typedef union
{
    int *__ip;
    union wait *__up;
} wait_status_ptr_t __attribute__((__transparent_union__));

pid_t wait (wait_status_ptr_t);
```

This interface allows either int * or union wait * arguments to be passed, using the int * calling convention. The program can call wait with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, wait’s implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
    return waitpid (-1, p.__ip, 0);
}
```
unused
When attached to a type (including a union or a struct), specify that variables of that type are meant to appear possibly unused. GCC will not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

5.2.27 Variable attributes

In GNU C you can declare variables with attributes that provide the compiler with various types of information about the declared variable. For example:

```c
int count __attribute__ ((aligned (16))) = 0;
```

Variable attributes are declared by specifying the keyword __attribute__ after a variable declaration. The keyword is followed by an attribute list, which consists of one or more comma-separated attribute names enclosed in double parentheses.

The following attributes are currently defined for variables: aligned, cleanup, common, deprecated, dllexport, dllimport, mode, nocommon, packed, section, shared, tls_model, transparent_union, unused, vector_size, and weak.

Attributes can be declared with double underscore characters (i.e., __) delimiting the attribute name as well as the keyword attribute. For example:

```c
int foo __attribute__ ((__vector_size__ (16)));
```

This enables you to declare attributes in header files without having to worry about a macro possibly having the same name as the attribute.

For a detailed description of the attribute syntax see Section 5.2.28.

**NOTE** Some attributes (such as section) can be applied to functions and types as well as variables.

**aligned** (alignment)
Specify a minimum alignment for the variable or structure field, measured in bytes. For example:

```c
int x __attribute__ ((aligned (16))) = 0;
```

This causes the compiler to allocate the global variable x on a 16-byte boundary.

You can also specify the alignment of structure fields. For example, to create a double-word aligned int pair:

```c
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a double member that forces the union to be double-word aligned.
As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can omit the alignment factor and just inform the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example:

```c
short array[3] __attribute__((aligned));
```

Whenever you omit the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

**cleanup** (cleanup_function)

Call the specified function when the variable goes out of scope. This attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.

If `-fexceptions` is enabled, then `cleanup_function` will be run during the stack unwinding that happens during the processing of the exception. Note that the `cleanup` attribute does not allow the exception to be caught, only to perform an action. It is undefined what happens if `cleanup_function` does not return normally.

**common**

Specify that a variable be allocated in “common” storage. The `nocommon` attribute specifies the opposite: to allocate space for it directly.

These attributes override the default chosen by the `-fno-common` and `-fcommon` flags respectively.
deprecated
Generate a warning if the variable is used anywhere in the source file. This is useful when identifying variables that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated variable, to enable users to easily find further information about why the variable is deprecated, or what they should do instead.

Note that the warning only occurs for uses. For example:

```c
extern int old_var __attribute__ ((deprecated));
extern int old_var;
int new_fn () { return old_var; }
```

This results in a warning on line 3 but not on line 2.

The deprecated attribute can also be used for functions and types (see Section 5.2.25 and Section 5.2.26.)

dlexport
dlimport
These attributes are described in Section 5.2.25.

mode (mode)
Specify the data type for the declaration: whichever type corresponds to the mode mode. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of byte or _byte__ to indicate the mode corresponding to a one-byte integer, word or _word__ for the mode of a one-word integer, and pointer or __pointer__ for the mode used to represent pointers.

nocommon
See common.

packed
Specify that a variable or structure field should have the smallest possible alignment: one byte for a variable and one bit for a field, unless you specify a larger value with the aligned attribute.

In the following example the structure is packed, so field x immediately follows a:

```c
struct foo
{
    char a;
    int x[2] __attribute__ ((packed));
};
```
By default the compiler places the data it generates in the .data or .bss sections of the output file. However, to support programs with nonstandard memory configurations, you may need to assign variables to other sections. The section attribute specifies that a variable is assigned to a particular section. For example:

```c
struct duart a __attribute__((section("DUART_A"))) = { 0 };  
struct duart b __attribute__((section("DUART_B"))) = { 0 };  
char stack[10000] __attribute__((section("STACK"))) = { 0 };  
int init_data __attribute__((section("INITDATA"))) = 0;
```

```c
main()
{
    /* Initialize stack pointer */
    init_sp (stack + sizeof (stack));

    /* Initialize initialized data */
    memcpy (&init_data, &data, &edata - &data);

    /* Turn on the serial ports */
    init_duart (&a);
    init_duart (&b);
}
```

In this example DUART_A and DUART_B are user-defined sections which are used to perform memory-mapped access to a serial port interface.

The section attribute can be used only with initialized global variables. If the variable is not initialized, GCC will issue a warning and ignore the attribute.

This restriction exists because the linker requires each object to be defined once, with the exception being uninitialized variables which can tentatively be assigned to the common (.bss) section and multiply “defined”. You can force a variable to be initialized with the -fno-common option or nocommon attribute.

GCC includes predefined sections which are used to assign variables to the Hexagon processor memories (Section 5.5.2). It also supports user-defined sections. For more information on sections see the Hexagon Binutils document.

**shared**

On Microsoft Windows, in addition to putting variable definitions in a named section, the section can also be shared among all running copies of an executable or DLL. For example, the following program defines shared data by putting it in a named section shared and marking the section shareable:

```c
int foo __attribute__((section("shared"), shared)) = 0;
```

```c
int
main()
{
    /* Read and write foo. All running copies see the same value. */
    return 0;
}
```
NOTE Because of linking restrictions, the `shared` attribute can be paired with the `section` attribute only in a fully initialized global definition. For more information see the `section` attribute above.

The `shared` attribute is available only on the Windows platform.

tls_model ("tls_model")
Specify the thread-local storage model (see Section 5.2.44) of a particular __thread variable, overriding `-ftls-model=` command line switch on a per-variable basis. The `tls_model` argument should be one of `global-dynamic`, `local-dynamic`, `initial-exec` or `local-exec`.

transparent_union
When attached to a function parameter which is a union, specify that the corresponding argument may have the type of any union member, but the argument is passed as if its type were that of the first union member. For more details see See Section 5.2.26. You can also use this attribute on a `typedef` for a union data type; then it applies to all function parameters with that type.

unused
When attached to a variable, specify that that the variable is meant to be possibly unused. GCC will not produce a warning for this variable.

vector_size (bytes)
This attribute specifies the vector size for the variable, measured in bytes. For example:

```c
int foo __attribute__ ((vector_size (16)));
```

This causes the compiler to set the mode for `foo`, to be 16 bytes, divided into `int` sized units. Assuming a 32-bit int (a vector of 4 units of 4 bytes), the corresponding mode of `foo` will be V4SI.

This attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct. Aggregates with this attribute are invalid, even if they are of the same size as a corresponding scalar. For example:

```c
struct S { int a; };
struct S __attribute__ ((vector_size (16))) foo;
```

This is invalid even if the size of the structure is the same as the size of the `int`.

weak
This attribute is described in Section 5.2.25.
5.2.28Attribute syntax

This section provides detailed information on using attributes, including the attribute syntax and the C declarations that attributes can be used with. Section 5.2.25 through Section 5.2.27 describe the attributes defined for functions, types, and variables.

An attribute specifier has the following syntax:

```
__attribute__((attribute-list))
```

An attribute list is a possibly empty comma-separated sequence of attributes, where each attribute is one of the following:

- Empty. Empty attributes are ignored.
- A word (which may be an identifier such as unused, or a reserved word such as const).
- A word, followed by attribute parameters enclosed in single parentheses. These parameters can appear in one of the following forms:
  - An identifier. For example, mode attributes use this form.
  - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, format attributes use this form.
  - A possibly empty comma-separated list of expressions. For example, format_arg attributes use this form, with the list being a single integer constant expression, while alias attributes use this form with the list being a single string constant.

An attribute specifier list is a sequence of one or more attribute specifiers, not separated by any other tokens.

Label attributes

In GNU C an attribute specifier list may appear after the colon following a label, other than a case or default label. The only attribute applicable to a label is unused – this is intended for code generated by programs which contains labels that may be unused but which is compiled with -Wall. It would not normally be appropriate to use in human-written code, though it can be useful in cases where the code that jumps to the label is contained within an #ifdef conditional. GNU C++ does not permit such placement of attribute lists, as it is permissible for a declaration, which can begin with an attribute list, to be labelled in C++. Declarations cannot be labelled in C90 or C99, so the ambiguity does not arise there.
Type attributes

An attribute specifier list may appear as part of a `struct`, `union` or `enum` specifier. It may go either immediately after the `struct`, `union` or `enum` keyword, or after the closing brace. It is ignored if the content of the structure, union or enumerated type is not defined in the specifier in which the attribute specifier list is used: that is, in usages such as `struct __attribute__((foo)) bar` with no following opening brace. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type defined is not complete until after the attribute specifiers.

Declaration attributes

Otherwise, an attribute specifier appears as part of a declaration, counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration), or to a particular declarator within a declaration. Where an attribute specifier is applied to a parameter declared as a function or an array, it should apply to the function or array rather than the pointer to which the parameter is implicitly converted, but this is not yet correctly implemented.

Any list of specifiers and qualifiers at the start of a declaration may contain attribute specifiers, whether or not such a list may in that context contain storage class specifiers. (Some attributes, however, are essentially in the nature of storage class specifiers, and only make sense where storage class specifiers may be used; for example, `section`.) There is one necessary limitation to this syntax: the first old-style parameter declaration in a function definition cannot begin with an attribute specifier, because such an attribute applies to the function instead by syntax described below (which, however, is not yet implemented in this case). In some other cases, attribute specifiers are permitted by this grammar but not yet supported by the compiler. All attribute specifiers in this place relate to the declaration as a whole. In the obsolescent usage where a type of `int` is implied by the absence of type specifiers, such a list of specifiers and qualifiers may be an attribute specifier list with no other specifiers or qualifiers.

An attribute specifier list may appear immediately before a declarator (other than the first) in a comma-separated list of declarators in a declaration of more than one identifier using a single list of specifiers and qualifiers. Such attribute specifiers apply only to the identifier before whose declarator they appear. For example:

```
__attribute__((noreturn)) void d0 (void),
__attribute__((format(printf, 1, 2))) d1 (const char *, ...),
  d2 (void)
```

The `noreturn` attribute applies to all the functions declared; the `format` attribute only applies to `d1`. 

An attribute specifier list may appear immediately before the comma, =, or semicolon terminating the declaration of an identifier other than a function definition. At present, such attribute specifiers apply to the declared object or function, but in the future they may attach to the outermost adjacent declarator. In simple cases there is no difference; however, in examples such as the following a difference does exist:

```c
void (***(f)) (void) __attribute__((noreturn));
```

Here the `noreturn` attribute currently applies to `f`, which causes a warning because `f` is not a function; however, in the future it may apply to the function `***f`. The precise semantics of what attributes in such cases will apply to are not yet specified. Where an assembler name for an object or function is specified (Section 5.2.38), at present the attribute must follow the `asm` specification; in the future, attributes before the `asm` specification may apply to the adjacent declarator, and those after it to the declared object or function.

**NOTE** An attribute specifier list may in the future be permitted to appear after the declarator in a function definition (before any old-style parameter declarations or the function body).

Attribute specifiers may be mixed with type qualifiers appearing inside the `[]` of a parameter array declarator, in the C99 construct by which such qualifiers are applied to the pointer to which the array is implicitly converted. Such attribute specifiers apply to the pointer, not to the array, but currently this is not implemented and they are ignored.

An attribute specifier list may appear at the start of a nested declarator. At present, there are some limitations in this usage: the attributes correctly apply to the declarator, but for most individual attributes the semantics this implies are not implemented. When attribute specifiers follow the `*` of a pointer declarator, they may be mixed with any type qualifiers present. The following describes the formal semantics of this syntax. It will make the most sense if you are familiar with the formal specification of declarators in the ISO C standard.

Consider (as in C99 subclause 6.7.5 paragraph 4) a declaration `T D1`, where `T` contains declaration specifiers that specify a type `Type` (such as `int`) and `D1` is a declarator that contains an identifier `ident`. The type specified for `ident` for derived declarators whose type does not include an attribute specifier is as in the ISO C standard.

If `D1` has the form `( attribute-specifier-list D )`, and the declaration `T D` specifies the type “`derived-declarator-type-list Type`” for `ident`, then `T D1` specifies the type “`derived-declarator-type-list attribute-specifier-list Type`” for `ident`.

If `D1` has the form `* type-qualifier-and-attribute-specifier-list D`, and the declaration `T D` specifies the type “`derived-declarator-type-list Type`” for `ident`, then `T D1` specifies the type “`derived-declarator-type-list type-qualifier-and-attribute-specifier-list Type`” for `ident`.

For example:

```c
void (__attribute__((noreturn)) ***f) (void);
```

This specifies the type “pointer to pointer to pointer to pointer to non-returning function returning `void`"
For another example:

```c
char *__attribute__((aligned(8))) *f;
```

This specifies the type “pointer to 8-byte-aligned pointer to char”. Note again that this does not work with most attributes; for example, the usage of `aligned` and `noreturn` attributes given above is not yet supported.

For compatibility with existing code written for compiler versions that did not implement attributes on nested declarators, some laxity is allowed in the placing of attributes. If an attribute that only applies to types is applied to a declaration, it will be treated as applying to the type of that declaration. If an attribute that only applies to declarations is applied to the type of a declaration, it will be treated as applying to that declaration; and, for compatibility with code placing the attributes immediately before the identifier declared, such an attribute applied to a function return type will be treated as applying to the function type, and such an attribute applied to an array element type will be treated as applying to the array type. If an attribute that only applies to function types is applied to a pointer-to-function type, it will be treated as applying to the pointer target type; if such an attribute is applied to a function return type that is not a pointer-to-function type, it will be treated as applying to the function type.

**Attributes in C++**

Some details may vary for C++. Due to inconsistencies in the grammar for attributes, some forms described here may not be successfully parsed in all cases.

There are some problems with the semantics of attributes in C++. For example, there are no manglings for attributes, although they may affect code generation, so problems may arise when attributed types are used in conjunction with templates or overloading. Similarly, `typeid` does not distinguish between types with different attributes. Support for attributes in C++ may be restricted in the future to attributes on declarations only, but not on nested declarators.
5.2.29 Pragmas

GCC supports two types of pragmas:

- Compatibility pragmas to support code originally written for other compilers
- GCC-specific pragmas to provide user control over compiler features such as visibility, optimization, and diagnostic messages.

Pragmas are declared by specifying the keyword `#pragma` followed by the pragma name and optional parameters. GCC-specific pragmas include “gcc” as part of their pragma names.

The pragmas are defined in the following sections:

- Structure-packing pragmas (Section 5.2.30)
- Diagnostic pragmas (Section 5.2.31)
- Visibility pragmas (Section 5.2.32)
- Push/pop macro pragmas (Section 5.2.33)
- Function-specific option pragmas (Section 5.2.34)

NOTE Pragmas affect the source code that follows the pragma declaration. They have no affect on the source code before the declaration.

It is generally recommended to use attributes (Section 5.2.24) instead of pragmas to assign properties to source code.

5.2.30 Structure-packing pragmas

For compatibility with Microsoft Windows compilers, GCC supports the following pragmas for changing the alignment of members of structures (excluding zero-width bitfields), unions, and classes.

`#pragma pack(n)`
Pack fields with the specified maximum alignment. The specified alignment value must be a small power of two. See the `-fpack-struct` option.

`#pragma pack()`
Reset the current alignment to the value in effect when compilation began. See the `-fpack-struct` option.

`#pragma pack(push[,n])`
Push the current alignment value on an internal stack, and optionally set the current alignment to the specified value.

`#pragma pack(pop)`
Pop the current alignment from the internal stack

NOTE The non-stack pack pragmas have no effect on the internal stack used to store alignment values.
5.2.31 Diagnostic pragmas

GCC provides pragmas which enable users to selectively enable or disable certain types of diagnostic messages, and change the kind of the diagnostic.

For example, it may be useful for a project to require that all source files compile with the -Werror option, but also allow certain files to generate specific types of warnings. Or, a project may selectively enable diagnostics and treat them as errors, depending on which preprocessor macros are defined.

```c
#pragma GCC diagnostic kind option
```

Change the category of the specified warning diagnostic.

- `option` is a double-quoted string containing the name of the command option (Section 3.4) that generates the diagnostic. (The warning diagnostics are normally controlled by the `-W...` options.)

- `kind` can have the following values:
  - `error`
    - Treat the specified diagnostic as an error.
  - `warning`
    - Treat the specified diagnostic as a warning (even if `-Werror` is enabled).
  - `ignored`
    - Ignore the specified diagnostic.

Example:

```c
#pragma GCC diagnostic warning "-Wformat"
```

Not all warning diagnostics can be changed. To determine which diagnostics can be changed (and which command options control them), use the command option `-fdiagnostics-show-option`.

The pragma overrides any command option settings.

The pragma must be declared in the program source code before any function or data definition. If placed anywhere else, the result is undefined.

If the pragma appears multiple times in the source code with the same `option` value specified, the setting in the last instance is the one that remains in effect.

**NOTE** This pragma is intended not as a general purpose replacement for command options, but rather for implementing strict controls over project policies.
#pragma message string

Print the specified string as a compiler message during compilation. The message is informational only, and is neither a compiler error nor a warning.

Example:

```c
#pragma message "Compiling " __FILE__ " ...
```

The specified string can be parenthesized, and is printed with location information (i.e., file name and line number). For example, the following statements:

```c
#define DO_PRAGMA(x) _Pragma (#x)
#define TODO(x) DO_PRAGMA(message ("TODO - " #x))

TODO(remember to fix this)
```

generate the following compilation message:

`/tmp/file.c:4: note: #pragma message: TODO - Remember to fix this'.

5.2.32 Visibility pragmas

```c
#pragma GCC visibility push(visibility)
#pragma GCC visibility pop
```

Set the visibility for multiple declarations without having to provide separate visibility attributes (Section 5.2.24) for each declaration.

Each declaration is treated as if the GCC visibility attribute were specified for that declaration.

**NOTE** In C++, these pragmas affect only namespace-scope declarations; class members and template specializations are not affected. To override the visibility of a specific member or instantiation, use an attribute.
5.2.33  Push/pop macro pragmas

For compatibility with Microsoft Windows compilers, GCC supports the following pragmas for temporarily redefining macros.

#pragma push_macro("macro_name")
   Push the definition of the specified macro on a dedicated internal stack.
   Each macro has its own internal stack which is independent of the stacks used for all other macros.

#pragma pop_macro("macro_name")
   Pop the definition of the specified macro from its internal stack.
   If the stack is empty, the macro definition remains unchanged.
For example:

#define X 1
#pragma push_macro("X")
#undef X
#define X -1
#pragma pop_macro("X")
int x [X];

5.2.34  Function-specific option pragmas

#pragma GCC optimize ("string"...)
   Set the global optimization options for multiple function declarations without having to provide separate optimization attributes (Section 5.2.24) for each declaration.
   The parameter can specify one or more strings. The parenthesis around the options is optional. Each function is declared as if the GCC optimize attribute were specified for that function.

#pragma GCC push_options
#pragma GCC pop_options
   Maintain a stack of the current target and optimization options. These pragmas are intended for cases (such as include files) where it is useful to temporarily redefine the optimization settings established using #pragma GCC optimize.

#pragma GCC reset_options
   Clear the global optimization option settings established by #pragma GCC optimize and reset them to the values specified on the command line.
5.2.35 Inline functions

Declaring a function `inline` causes GCC to replace each call to the function with a complete copy of the function code. This improves execution performance by eliminating the extra code required to perform a function call. In addition, if any of the actual argument values are constants, the compiler may be able to simplify the code so not all of the inline function code needs to be included.

The effect of function inlining on code size is less predictable; depending on the particular case, the object code may become larger or smaller.

**NOTE** Substantial differences exist between how inline functions are defined in the ISO C99 standard and how they are currently implemented in GCC. Since GCC will eventually implement the ISO C99 semantics for inline functions, it is best to only use `static inline` for now to ensure compatibility. (The existing semantics will remain available when option `-std=gnu89` is specified, but eventually the default will be `-std=gnu99` which will implement the C99 semantics, though it does not yet do so.)

**Specifying inline functions**

To declare a function inline, specify the `inline` keyword in its declaration. For example:

```c
inline int inc (int *a)
{
    (*a)++;
}
```

**NOTE** If you are writing a header file to be included in ISO C programs, use the keyword `__inline__` instead of `inline` (Section 5.2.1).

Unlike C++, the `inline` keyword in C does not affect the function linkage.

As an alternative to using the `inline` keyword, you can automatically inline all functions that are smaller than a certain size by using the command option `-finline-functions`. The function size is specified with the command option `-fparam max-inline-insn-auto`. For more information on these options see Section 3.4.9.

To ensure that an inline function is always inlined regardless of the compiler settings, specify the function attribute `always_inline` (Section 5.2.25). For example:

```c
/* function prototype */
inline int inc (int *a) __attribute__((always_inline));
```

**NOTE** GCC automatically inlines member functions defined in the class body of C++ programs, even if the functions are not explicitly declared `inline`. 
Enabling function inlining

Function inlining is an optimization; therefore, the compiler normally performs it only when optimization is enabled (Section 3.4.9). By default all function calls in a program are implemented normally (using call instructions).

Table 5-1 lists what functions are inlined at each optimization level.

**Table 5-1  Enabling function inlining**

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Functions Inlined</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>All inline functions marked with function attribute “always_inline” (Section 5.2.25). No other inlining occurs regardless of any command option settings.</td>
</tr>
<tr>
<td>-O1 and up</td>
<td>All inline functions. All calls to C++ member functions defined in the scope of their class (except if option -fno-default-inline (Section 3.4.4) is used).</td>
</tr>
<tr>
<td>-O2 and up</td>
<td>All static functions that are called only once (even if the functions are not specified as inline).</td>
</tr>
<tr>
<td>-O3</td>
<td>All functions specified by option -finline-functions (even if the functions are not specified as inline).</td>
</tr>
</tbody>
</table>

**NOTE** The command option -fno-inline-functions disables only those functions heuristically inlined by -finline-functions. To disable the inlining of all inline functions (except those declared with attribute always_inline), use the option -fno-inline. For more information on these options see Section 3.4.9.

To prevent a function from being inlined, declare it with the function attribute noinline (Section 5.2.25).

Inline function restrictions

Not all functions can be inlined. A function cannot be inlined if it uses any of the following language features:

- Variadic functions (which accept variable arguments)
- Automatic storage with variable size (alloca)
- Variable-sized data types (Section 5.2.16)
- Computed goto’s (Section 5.2.7)
- Nonlocal goto’s
- Nested functions (Section 5.3.1)

If you use the command option -Winline (Section 3.4.7), the compiler will generate a warning message whenever a function marked inline cannot be inlined. The message includes a description of why the function cannot be inlined.
Static and external inline functions

If a function is declared both inline and static, and if all calls to the function are inlined and the function’s address is never used, then the function’s own assembly code is never referenced. In this case GCC does not output assembly code for the function unless you specify the command option -fkeep-inline-functions (Section 3.4.9).

Some function calls cannot be inlined; in particular, recursive calls and any calls that precede the function’s definition. If a non-inlined call exists (or if the program references the function address), the function is always compiled to assembly code.

When an inline function is not declared static, the compiler must assume that calls to it from other source files may exist. Since a global symbol can be defined only once in a program, the function must not be defined in the other source files, so any calls in those files cannot be inlined. Therefore, a non-static inline function is always compiled on its own in the usual manner.

When a function is declared both inline and extern, the definition is used only for inlining. In no case is the function compiled on its own, not even if its address is explicitly referenced in the program. Such an address becomes an external reference, as if the function had only been declared and not defined.

NOTE This combination of inline and extern has almost the effect of a macro. The proper way to use such functions is to put in the header file a version of the function definition which includes both of these keywords, and then put in the library file a copy of the function definition which lacks the keywords. The definition in the header file will inline most calls to the function. If any other uses of the function remain, they will refer to the copy in the library file.

5.2.36 Inline assembly language

GCC supports inline assembly language using the asm statement. For more information see Chapter 6.

5.2.37 Constructing function calls

Using the three built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

NOTE These functions may interact badly with some sophisticated features or other extensions of the language. Therefore they are recommended for use only in very simple functions which serve as mere forwarders of their arguments.
void * __builtin_apply_args ()
    Return pointer to data describing how to perform a call with the same arguments
    as were passed to the current function.

    This function stores the following items—argument pointer register, structure
    value address, and all registers that might be used to pass arguments to a
    function—in a block of memory allocated on the stack. It returns the address of
    the block

void * __builtin_apply (void (*function)(),
    void *arguments, size_t size)
    Invoke function with a copy of the parameters described by arguments and size.
    The value of arguments should be the value returned by __builtin_apply_ 
    args. The argument size specifies the size of the stack argument data, in bytes.
    This function returns a pointer to data describing how to return whatever value
    was returned by function. The data is saved in a block of memory allocated on the
    stack.
    Computing the proper value for size is not always simple. This value is used by
    __builtin_apply to compute the amount of data that should be pushed on the
    stack and copied from the incoming argument area.

void __builtin_return (void *result)
    Return value described by result from the containing function. You should
    specify, for result, a value returned by __builtin_apply.

5.2.38 Controlling names used in assembly code

You can specify the name to be used in the assembly code for a C function or variable by
writing the asm (or __asm__) keyword after the declarator as follows:

    int foo asm ("myfoo") = 2;

This specifies that the name to be used for the variable foo in the assembler code should
be myfoo rather than the usual foo.

On systems where an underscore is normally prepended to the name of a C function or
variable, this feature allows you to define names for the linker that do not start with an
underscore.

It does not make sense to use this feature with a non-static local variable since such
variables do not have assembler names. GCC presently accepts such code with a warning,
but will probably be changed to issue an error, rather than a warning, in the future.
You cannot use `asm` in this way in a function definition; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```c
extern func () asm ("FUNC");

func (x, y)
    int x, y;

/* ... */
```

**NOTE** Do not specify names that conflict with any other assembler symbols.

### 5.2.39 Function names as strings

GCC provides three predefined variables which always contain the name of the current function as a string:

- `__func__`
- `__FUNCTION__`
- `__PRETTY_FUNCTION__`

`__func__` is part of the C99 standard.

The identifier `__func__` is implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration `static const char __func__[] = "function-name";` appeared, where `function-name` is the name of the lexically-enclosing function. This name is the unadorned name of the function.

`__FUNCTION__` is another name for `__func__`.

In C, `__PRETTY_FUNCTION__` is yet another name for `__func__`. However, in C++ it contains the type signature of the function as well as its bare name. For example:

```c
extern "C" {
    extern int printf (char *, ...);
}

class a {
    public:
        void sub (int i)
        {
            printf ("__FUNCTION__ = %s\n", __FUNCTION__);
            printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
        }
};
int main (void)
{
    a ax;
    ax.sub (0);
    return 0;
}
```
This program generates the following output:

```c
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = void a::sub(int)
```

**NOTE**  These identifiers are variables, not preprocessor macros.

### 5.2.40 Getting the return or frame address of a function

The following predefined functions can be used to obtain information on the callers of a function.

```c
void *__builtin_return_address(unsigned int level)
```

Return the return address of the current function, or of one of its callers.

The `level` argument is number of frames to scan up the call stack. A value of 0 yields the return address of the current function, a value of 1 yields the return address of the caller of the current function, and so forth.

When inlining the expected behavior is that the function will return the address of the function that will be returned to. To work around this behavior use the `noinline` function attribute.

The `level` argument must be a constant integer.

**NOTE**  GCC does not support nonzero `level` argument values for this function.

```c
void *__builtin_frame_address(unsigned int level)
```

Similar to `__builtin_return_address`, but returns the address of the function frame rather than the return address of the function.

Calling `__builtin_frame_address` with a value of 0 yields the frame address of the current function, a value of 1 yields the frame address of the caller of the current function, and so forth.

The frame is the area on the stack which holds local variables and saved registers. The frame address is normally the address of the first word pushed on to the stack by the function. However, the exact definition depends upon the processor and the calling convention. If the processor has a dedicated frame pointer register, and the function has a frame, then `__builtin_frame_address` will return the value of the frame pointer register.

When the top of the stack has been reached, this function will return 0 if the first frame pointer is properly initialized by the startup code.

**NOTE**  This function should only be used with a nonzero argument for debugging purposes.
5.2.41 Offsetof

The following predefined function can be used instead of the macro offsetof:

```c
size_t __builtin_offsetof (typename, offsetof_member_designator)
```

Return the type-relative byte offset of the specified type element.

`offsetof_member_designator` can have any of the following forms:

- `identifier`
- `offsetof_member_designator.identifier`
- `offsetof_member_designator[expression]`

This function is defined to be consistent with the following definition:

```c
#define offsetof(type, member)  __builtin_offsetof (type, member)
```

In C++, `type` may be dependent. In either case, `member` consists either of a single identifier or a sequence of member accesses and array references.

5.2.42 Object size checking

The following predefined functions can be used to implement a limited buffer overflow protection mechanism which can prevent some buffer overflow attacks:

```c
size_t __builtin_object_size (void * ptr, int type)
```

Return the number of bytes from `ptr` to the end of the object specified by `ptr` (if the value is known at compile time).

`type` is an integer value in the range 0-3:

- If the least significant bit (LSB) of the value is 0, the referenced object is treated as a whole variable.
- If the LSB is 1, the closest surrounding subobject is treated as the referenced object.

The second bit in a `type` value determines if the function returns the maximum or minimum of remaining bytes. If `ptr` can reference more than one object, and all of the objects are known at compile time, the returned value depends on the most significant bit (MSB) of `type`:

- If the MSB of the value is 0, the returned value is the maximum of the remaining byte counts in those objects.
- If the MSB is 1, the returned value is the minimum of the remaining byte counts in those objects.

If the compiler cannot determine which objects `ptr` references, this function returns `(size_t)-1` for `type` values 0 or 1, and `(size_t)0` for values 2 or 3.

This function never evaluates its arguments for side-effects. If side-effects exist, it returns `(size_t)-1` for `type` values 0 or 1, and `(size_t)0` for values 2 or 3.
These functions are similar to the respective C library functions, but include an additional last parameter which specifies the number of bytes remaining in the object that the `dest` parameter references, or `(size_t)-1` if the size is not known.

These functions are optimized into their respective base functions if the last argument is `(size_t)-1` or if it can be determined at compile time that the destination object cannot be overflowed. However, if it can be determined at compile time that the object will always be overflowed, the compiler generates a warning.

```c
int __builtin___sprintf_chk (char *s, int flag, size_t os,
                             const char *fmt, ...)
```

```c
int __builtin___snprintf_chk (char *s, size_t maxlen, int flag,
                               size_t os, const char *fmt, ...)
```

```c
int __builtin___vprintf_chk (char *s, int flag, size_t os,
                              const char *fmt, va_list ap)
```

```c
int __builtin___vsnprintf_chk (char *s, size_t maxlen, int flag,
                                size_t os, const char *fmt, va_list ap)
```

These functions are similar to the respective C library functions, but include an additional `flag` parameter which is passed unchanged to the base function, and which contains values specifying what additional security measures the function should perform.

As in the other built-in checking functions, `os` specifies the referenced object (of size `s`). Note, however, that if `os` is `(size_t)-1` then these functions are optimized to their base functions only if `flag` is 0; otherwise, the checking function is called with `os` set to `(size_t)-1`.

```c
__builtin__printf_chk
__builtin__vprintf_chk
__builtin__fprintf_chk
__builtin__vfprintf_chk
```

These functions are similar to the respective C library functions, but include an additional `flag` parameter preceding the format string parameter. As with the other checking functions, the compiler attempts to optimize these functions to their base functions; otherwise, the checking function is called with the `flag` parameter passed to it.
5.2.43 Other built-in functions provided by GCC

GCC provides a large number of built-in functions other than the ones mentioned above. Some of these are for internal use in the processing of exceptions or variable-length argument lists and will not be documented here because they may change from time to time; we do not recommend general use of these functions.

The remaining functions are provided for optimization purposes.

GCC includes built-in versions of many of the functions in the standard C library. The versions prefixed with `__builtin__` will always be treated as having the same meaning as the C library function even if you specify the `-fno-builtin` option (Section 3.4.3). Many of these functions are only optimized in certain cases; if they are not optimized in a particular case, a call to the library function will be emitted.

Outside strict ISO C mode (-ansi, -std=c89 or -std=c99), the functions `__builtin_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `drem`, `dreml`, `exp`, `exp10`, `exp10l`, `ffs`, `ffsll`, `fscanf`, `fprintf_unlocked`, `fgets_unlocked`, `gamma`, `gammaf`, `gettext`, `index`, `j0f`, `j0l`, `j1f`, `j1l`, `jnf`, `jnl`, `jn`, `mempcpy`, `pow`, `pow10`, `pow10l`, `printf_unlocked`, `rint`, `scalbf`, `scalbl`, `scalb`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `strdup`, `strfmon`, `y0f`, `y0l`, `y1f`, `y1l`, `ynf`, `ynl` and `yn` may be handled as built-in functions. All these functions have corresponding versions prefixed with `__builtin__`, which may be used even in strict C89 mode.

The ISO C99 functions `__Exit`, `acosf`, `acoshf`, `acoshl`, `asinhf`, `asinhl`, `asinh`, `atanhf`, `atanhl`, `atanh`, `cabsf`, `cabsl`, `cabs`, `cacosf`, `cacosfl`, `cacos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhl`, `casinh`, `casinl`, `catanf`, `catanhf`, `catanh`, `catanh`, `catanl`, `catan`, `cbt`, `cbt`, `ccosf`, `ccoshf`, `ccoshl`, `ccos`, `ccosl`, `ccos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhl`, `casinh`, `casinl`, `catanf`, `catanhf`, `catanh`, `catanh`, `catanl`, `catan`, `cbt`, `cbt`, `ccosf`, `ccoshf`, `ccoshl`, `ccos`, `ccosl`, `ccos`, `cexpf`, `cexp`, `cimagf`, `cimagl`, `cimag`, `conjf`, `conj`, `conjl`, `conj`, `cypesignf`, `cypesignl`, `cotypesignf`, `cotypesignl`, `cwpf`, `cpwl`, `cwpow`, `cprojf`, `cprojl`, `cproj`, `crealf`, `creall`, `creal`, `csinf`, `csinfl`, `csinh`, `csinl`, `csin`, `csqrtf`, `csqrfl`, `csqrt`, `ctanf`, `ctanhf`, `ctanh`, `ctanl`, `ctan`, `erf`, `erfc`, `erfc`, `erff`, `erfl`, `erf`, `exp2f`, `exp2l`, `exp2`, `expmlf`, `expml`, `fdimf`, `fdiml`, `fdim`, `fmaf`, `fmal`, `fmaxf`, `fmaxl`, `fmax`, `fma`, `fminf`, `fminl`, `fmin`, `hypotf`, `hypotl`, `hypot`, `ilogbf`, `ilogbl`, `ilogb`, `imaxabs`, `lgammaf`, `lgamml`, `lgamma`, `llabs`, `llrintf`, `llrintl`, `llrint`, `llroundf`, `llroundl`, `llround`, `log1pf`, `log1pl`, `log1p`, `log2f`, `log2l`, `log2`, `logbf`, `logbl`, `lrintf`, `lrintl`, `lrint`, `lroundf`, `lroundl`, `lround`, `nearbyintf`, `nearbyintl`, `nearbyint`, `nextafterf`, `nextafter`, `nexttowardf`, `nexttoward`, `remainderf`, `remainderl`, `remainder`, `remquof`, `remquol`, `remquo`, `rintf`, `rintl`, `rint`, `roundf`, `roundl`, `round`, `scalblnf`, `scalblnl`, `scalbln`, `scalbnf`, `scalbnl`, `scalbn`, `snprintf`, `tgammaf`, `tgamma`, `truncf`, `trunc`, `vfscanf`, `vs scanf`, `vsnprintf` and `vsscanf` are handled as built-in functions except in strict ISO C90 mode (-ansi or -std=c89).
There are also built-in versions of the ISO C99 functions \texttt{acosf}, \texttt{acosl}, \texttt{asinf}, \texttt{asinl}, \texttt{atan2f}, \texttt{atan2l}, \texttt{atanf}, \texttt{atanl}, \texttt{ceilf}, \texttt{ceill}, \texttt{cosf}, \texttt{cosh}, \texttt{coshl}, \texttt{cosl}, \texttt{expf}, \texttt{expl}, \texttt{fabsf}, \texttt{fabsl}, \texttt{floorf}, \texttt{floorm}, \texttt{fmodf}, \texttt{fmodl}, \texttt{frexp}, \texttt{frexpl}, \texttt{ldexf}, \texttt{ldeql}, \texttt{log10f}, \texttt{log10l}, \texttt{logf}, \texttt{logl}, \texttt{modff}, \texttt{modfl}, \texttt{modf}, \texttt{powf}, \texttt{powl}, \texttt{sinf}, \texttt{sinhf}, \texttt{sinhl}, \texttt{sinl}, \texttt{sqrtf}, \texttt{sqrtl}, \texttt{tanf}, \texttt{tanhf}, \texttt{tanfl} and \texttt{tanl} that are recognized in any mode since ISO C90 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with \texttt{__builtin__}.

The ISO C90 functions \texttt{abort}, \texttt{abs}, \texttt{acos}, \texttt{asin}, \texttt{atan2}, \texttt{atan}, \texttt{calloc}, \texttt{ceil}, \texttt{cosh}, \texttt{cos}, \texttt{exit}, \texttt{exp}, \texttt{fabs}, \texttt{floor}, \texttt{fmod}, \texttt{fprintf}, \texttt{fputs}, \texttt{frexp}, \texttt{fscanf}, \texttt{labs}, \texttt{ldexp}, \texttt{log}, \texttt{malloc}, \texttt{memcmp}, \texttt{memcpy}, \texttt{memset}, \texttt{modf}, \texttt{pow}, \texttt{printf}, \texttt{putchar}, \texttt{puts}, \texttt{scanf}, \texttt{sin}, \texttt{snprintf}, \texttt{sprintf}, \texttt{sqrt}, \texttt{sqrtl}, \texttt{tan}, \texttt{tan}, \texttt{vfprintf}, \texttt{vprintf} and \texttt{vsprintf} are all recognized as built-in functions unless \texttt{-fno-builtin} is specified (or \texttt{-fno-builtin-function} is specified for an individual function). All of these functions have corresponding versions prefixed with \texttt{__builtin__}.

GCC provides built-in versions of the ISO C99 floating point comparison macros that avoid raising exceptions for unordered operands. They have the same names as the standard macros (\texttt{isgreater}, \texttt{isgreaterequal}, \texttt{isless}, \texttt{islessequal}, \texttt{islessgreater}, and \texttt{isunordered}), with \texttt{__builtin__} prefixed. We intend for a library implementor to be able to simply \#define each standard macro to its built-in equivalent.

\begin{verbatim}
int __builtin_types_compatible_p (type1, type2)
Determined whether two types are the same.
Returns 1 if the unqualified versions of the types \texttt{type1} and \texttt{type2} (which are types, not expressions) are compatible, 0 otherwise. The result of this built-in function can be used in integer constant expressions.
This function ignores top level qualifiers (e.g., \texttt{const}, \texttt{volatile}). For example, \texttt{int} is equivalent to \texttt{const int}.
The type \texttt{int[]} and \texttt{int[5]} are compatible. On the other hand, \texttt{int} and \texttt{char *} are not compatible, even if the size of their types, on the particular architecture are the same. Also, the amount of pointer indirection is taken into account when determining similarity. Consequently, \texttt{short *} is not similar to \texttt{short **}.
Furthermore, two types that are typedefed are considered compatible if their underlying types are compatible.
An \texttt{enum} type is not considered to be compatible with another \texttt{enum} type even if both are compatible with the same integer type; this is what the C standard specifies. For example, \texttt{enum \{foo, bar\}} is not similar to \texttt{enum \{hot, dog\}}.
\end{verbatim}
This function would typically be used in code whose execution varies depending on the arguments’ types. For example:

```c
#define foo(x)                                                     
({                                                           
    typeof (x) tmp;                                             
    if (__builtin_types_compatible_p (typeof (x), long double)) 
        tmp = foo_long_double (tmp);                           
    else if (__builtin_types_compatible_p (typeof (x), double)) 
        tmp = foo_double (tmp);                                
    else if (__builtin_types_compatible_p (typeof (x), float))  
        tmp = foo_float (tmp);                                 
    else                                                        
        abort ();                                               
    tmp;                                                        
})
```

**NOTE** This construct is only available for C.

```c
type __builtin_choose_expr (const_exp, exp1, exp2) 
Evaluate code depending on the value of a constant expression. This built-in function returns exp1 if `const_exp`, which is a constant expression that must be able to be determined at compile time, is nonzero. Otherwise it returns 0.

This function is analogous to the `?:` operator in C, except that the expression returned has its type unaltered by promotion rules. Also, the function does not evaluate the expression that was not chosen. For example, if `const_exp` evaluates to true, `exp2` is not evaluated even if it has side-effects.

This function can return an lvalue if the chosen argument is an lvalue.

If `exp1` is returned, the return type is the same as `exp1`’s type. Similarly, if `exp2` is returned, its return type is the same as `exp2`. For example:

```c
#define foo(x)                                                    
__builtin_choose_expr (                                         
   __builtin_types_compatible_p (typeof (x), double),            
   foo_double (x),                                               
   __builtin_choose_expr (                                       
   __builtin_types_compatible_p (typeof (x), float),           
   foo_float (x),                                              
/* The void expression results in a compile-time error when assigning the result to something. */ (void)0)
```

**NOTE** This construct is available only for C. Furthermore, the unused expression (`exp1` or `exp2` depending on the value of `const_exp`) may still generate syntax errors. This may change in future revisions.
**int __builtin_constant_p (exp)**

Determine if a value is known to be constant at compile-time, and hence that GCC can perform constant-folding on expressions involving that value.

The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. The return value 0 does not indicate that the value is not a constant, but merely that GCC cannot prove it is a constant with the specified value of the -O option.

You would typically use this function in an embedded application where memory was a critical resource. If you have some complex calculation, you may want it to be folded if it involves constants, but need to call a function if it does not. For example:

```c
#define Scale_Value(X)      \
   (__builtin_constant_p (X) \ 
    ? ((X) * SCALE + OFFSET) : Scale (X))
```

You can use this function in either a macro or an inline function. However, if you use it in an inlined function and pass an argument of the function as the argument to the built-in, GCC will never return 1 when you call the inline function with a string constant or compound literal (Section 5.2.20) and will not return 1 when you pass a constant numeric value to the inline function unless you specify the -O option.

You can also use __builtin_constant_p in initializers for static data. For example:

```c
static const int table[] = {
   __builtin_constant_p (EXPRESSION) ? (EXPRESSION) : -1,
   /* ... */
};
```

This is an acceptable initializer even if EXPRESSION is not a constant expression. GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

**long __builtin_expect (long exp, long c)**

Provide the compiler with branch prediction information.

In general, you should prefer to use actual profile feedback for this (-fprofile-arcs), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of exp, which should be an integral expression. The value of c must be a compile-time constant. The semantics of the built-in are that it is expected that exp == c. For example:

```c
if (__builtin_expect (x, 0))
   foo ();
```
This indicates that we do not expect to call \texttt{foo}, since we expect \texttt{x} to be zero. Since you are limited to integral expressions for \texttt{exp}, when testing pointer or floating-point values you should use constructions of the following form:

\begin{verbatim}
if (\_\texttt{builtin_expect} (ptr != NULL, 1))
    error ();
\end{verbatim}

\begin{verbatim}
void \_\texttt{builtin_prefetch} (\texttt{const void *} \texttt{addr} \left[\texttt{,rw,locality}\right])
\end{verbatim}

Minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to \texttt{\_\texttt{builtin_prefetch}} into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of \texttt{addr} is the address of the memory to prefetch. There are two optional arguments, \texttt{rw} and \texttt{locality}. The value of \texttt{rw} is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero (the default) indicates that the prefetch is preparing for a read. The value \texttt{locality} must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

\begin{verbatim}
for (i = 0; i < n; i++)
{
    a[i] = a[i] + b[i];
    \_\texttt{builtin_prefetch} (&a[i+j], 1, 1);
    \_\texttt{builtin_prefetch} (&b[i+j], 0, 1);
    /* ... */
}
\end{verbatim}

Data prefetch does not generate faults if \texttt{addr} is invalid, but the address expression itself must be valid. For example, a prefetch of \texttt{p->next} will not fault if \texttt{p->next} is not a valid address, but evaluation will fault if \texttt{p} is not a valid address.

\textbf{NOTE} If the target does not support data prefetch, the address expression is evaluated if it includes side effects; however, no other code is generated and GCC does not issue a warning.

\begin{verbatim}
double \_\texttt{builtin_huge_val} (void)
Returns a positive infinity, if supported by the floating-point format, else DBL_MAX. This function is suitable for implementing the ISO C macro \texttt{HUGE_VAL}.
\end{verbatim}

\begin{verbatim}
float \_\texttt{builtin_huge_valf} (void)
Similar to \_\texttt{builtin_huge_val}, except return type is float.
\end{verbatim}

\begin{verbatim}
long double \_\texttt{builtin_huge_vall} (void)
Similar to \_\texttt{builtin_huge_val}, except return type is long double.
\end{verbatim}
double __builtin_inf (void)
Similar to __builtin_huge_val, except a warning is generated if the target floating-point format does not support infinities. This function is suitable for implementing the ISO C99 macro INFINITY.

float __builtin_inff (void)
Similar to __builtin_inf, except return type is float.

long double __builtin_infl (void) Similar to __builtin_inf, except return type is long double.

double __builtin_nan (const char *str)
This is an implementation of the ISO C99 function nan.
Since ISO C99 defines this function in terms of strtod, which we do not implement, a description of the parsing is in order. The string is parsed as by strtol; that is, the base is recognized by leading 0 or 0x prefixes. The number parsed is placed in the significand such that the least significant bit of the number is at the least significant bit of the significand. The number is truncated to fit the significand field provided. The significand is forced to be a quiet NaN.

This function, if given a string literal, is evaluated early enough that it is considered a compile-time constant.

float __builtin_nanf (const char *str)
Similar to __builtin_nan, except return type is float.

long double __builtin_nanl (const char *str)
Similar to __builtin_nan, except return type is long double.

double __builtin_nans (const char *str)
Similar to __builtin_nan, except significand is forced to be a signaling NaN.

float __builtin_nansf (const char *str)
Similar to __builtin_nans, except return type is float.

long double __builtin_nansl (const char *str)
Similar to __builtin_nans, except return type is long double.

int __builtin_ffs (unsigned int x)
Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

int __builtin_clz (unsigned int x)
Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

int __builtin_ctz (unsigned int x)
Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

int __builtin_popcount (unsigned int x)
Returns the number of 1-bits in x.

int __builtin_parity (unsigned int x)
Returns the parity of x, i.e. the number of 1-bits in x modulo 2.

int __builtin_ffsl (unsigned long)
Similar to __builtin_ffs, except argument type is unsigned long.

int __builtin_clzl (unsigned long)
Similar to __builtin_clz, except argument type is unsigned long.
int __builtin_ctzl (unsigned long)
   Similar to __builtin_ctz, except argument type is unsigned long.

int __builtin_popcountl (unsigned long)
   Similar to __builtin_popcount, except argument type is unsigned long.

int __builtin_parityl (unsigned long)
   Similar to __builtin_parity, except argument type is unsigned long.

int __builtin_ffsll (unsigned long long)
   Similar to __builtin_ffs, except argument type is unsigned long long.

int __builtin_clzll (unsigned long long)
   Similar to __builtin_clz, except argument type is unsigned long long.

int __builtin_ctzll (unsigned long long)
   Similar to __builtin_ctz, except argument type is unsigned long long.

int __builtin_popcountll (unsigned long long)
   Similar to __builtin_popcount, except argument type is unsigned long long.

int __builtin_parityll (unsigned long long)
   Similar to __builtin_parity, except argument type is unsigned long long.

5.2.44 Thread-local storage

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. The run-time model GCC uses to implement this originates in the IA-64 processor-specific ABI, but has since been migrated to other processors as well. It requires significant support from the linker (ld), dynamic linker (ld.so), and system libraries (libc.so and libpthread.so), so it is not available everywhere.

At the user level, the extension is visible with a new storage class keyword: __thread. For example:

   __thread int i;
   extern __thread struct state s;
   static __thread char *p;

The __thread specifier may be used alone, with the extern or static specifiers, but with no other storage class specifier. When used with extern or static, __thread must appear immediately after the other storage class specifier.

The __thread specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data member.

When the address-of operator is applied to a thread-local variable, it is evaluated at runtime and returns the address of the current thread’s instance of that variable. An address so obtained may be used by any thread. When a thread terminates, any pointers to thread-local variables in that thread become invalid.

No static initialization may refer to the address of a thread-local variable.
In C++, if an initializer is present for a thread-local variable, it must be a *constant-expression*, as defined in 5.19.2 of the ANSI/ISO C++ standard.

### 5.2.44.1 ISO/IEC 9899:1999 edits for thread-local storage

The following are a set of changes to ISO/IEC 9899:1999 (aka C99) that document the exact semantics of the language extension.

- **5.1.2 Execution environments**
  
  Add new text after paragraph 1

  Within either execution environment, a *thread* is a flow of control within a program. It is implementation defined whether or not there may be more than one thread associated with a program. It is implementation defined how threads beyond the first are created, the name and type of the function called at thread startup, and how threads may be terminated. However, objects with thread storage duration shall be initialized before thread startup.

- **6.2.4 Storage durations of objects**

  Add new text before paragraph 3

  An object whose identifier is declared with the storage-class specifier *__thread* has *thread storage duration*. Its lifetime is the entire execution of the thread, and its stored value is initialized only once, prior to thread startup.

- **6.4.1 Keywords**

  Add *__thread*.

- **6.7.1 Storage-class specifiers**

  Add *__thread* to the list of storage class specifiers in paragraph 1.

  Change paragraph 2 to

  With the exception of *__thread*, at most one storage-class specifier may be given [...]. The *__thread* specifier may be used alone, or immediately following *extern* or *static*.

  Add new text after paragraph 6

  The declaration of an identifier for a variable that has block scope that specifies *__thread* shall also specify either *extern* or *static*.

  The *__thread* specifier shall be used only with variables.
5.2.44.2 ISO/IEC 14882:1998 edits for thread-local storage

The following are a set of changes to ISO/IEC 14882:1998 (aka C++98) that document the exact semantics of the language extension.

- **[intro.execution]**
  
  New text after paragraph 4

  A thread is a flow of control within the abstract machine. It is implementation-defined whether or not there may be more than one thread.

  New text after paragraph 7

  It is unspecified whether additional action must be taken to ensure when and whether side effects are visible to other threads.

- **[lex.key]**
  
  Add __thread.

- **[basic.start.main]**
  
  Add after paragraph 5

  The thread that begins execution at the main function is called the main thread. It is implementation-defined how functions beginning threads other than the main thread are designated or typed. A function so designated, as well as the main function, is called a thread startup function. It is implementation-defined what happens if a thread startup function returns, or what happens to other threads when any thread calls exit.

- **[basic.start.init]**
  
  Add after paragraph 4

  The storage for an object of thread storage duration shall be statically initialized before the first statement of the thread startup function. An object of thread storage duration shall not require dynamic initialization.

- **[basic.start.term]**
  
  Add after paragraph 3

  The type of an object with thread storage duration shall not have a non-trivial destructor, nor shall it be an array type whose elements (directly or indirectly) have non-trivial destructors.

- **[basic.stc]**
  
  Add “thread storage duration” to the list in paragraph 1.

  Change paragraph 2

  Thread, static, and automatic storage durations are associated with objects introduced by declarations [...].

  Add __thread to the list of specifiers in paragraph 3.
New section before [basic.stc.static]

The keyword __thread applied to a non-local object gives the object thread storage duration.

A local variable or class data member declared both static and __thread gives the variable or member thread storage duration.

Change paragraph 1

All objects which have neither thread storage duration, dynamic storage duration nor are local [...].

Add __thread to the list in paragraph 1.

Change paragraph 1

With the exception of __thread, at most one storage-class-specifier shall appear in a given decl-specifier-seq. The __thread specifier may be used alone, or immediately following the extern or static specifiers. [...] Add after paragraph 5

The __thread specifier can be applied only to the names of objects and to anonymous unions.

Add after paragraph 6

Non-static members shall not be __thread.

5.3 C-specific extensions

The C-specific language extensions are available only in the C compiler.

5.3.1 Nested functions

A nested function is a function defined inside another function. The nested function’s name is local to the block where it is defined. For example, here we define a nested function named square, and call it twice:

```c
foo (double a, double b) {
    double square (double z) { return z * z; }
    return square (a) + square (b);
}
```
The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, the following nested function uses an inherited variable named *offset*:

```c
bar (int *array, int offset, int size)
{
    int access (int *array, int index)
        { return array[index + offset]; }
    int i;
    /* ... */
    for (i = 0; i < size; i++)
        /* ... */ access (array, i) /* ... */
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function. For example:

```c
hack (int *array, int size)
{
    void store (int index, int value)
        { array[index] = value; }

    intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit.

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it’s not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (*Section 5.2.6*). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well.
For example:

```c
bar (int *array, int offset, int size)
{
    __label__ failure;
    int access (int *array, int index)
    {
        if (index > size)
            goto failure;
        return array[index + offset];
    }
    int i;
    /* ... */
    for (i = 0; i < size; i++)
        /* ... */ access (array, i) /* ... */
        /* ... */
    return 0;

    /* Control comes here from access
       if it detects an error. */
    failure:
        return -1;
}
```

A nested function always has internal linkage. Declaring one with `extern` is erroneous. If you need to declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations). For example:

```c
bar (int *array, int offset, int size)
{
    __label__ failure;
    auto int access (int *, int);
    /* ... */
    int access (int *array, int index)
    {
        if (index > size)
            goto failure;
        return array[index + offset];
    }
    /* ... */
}
```
### 5.3.2 Generalized lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them. All these extensions are deprecated.

A compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

\[
(a, b) += 5 \\
a, (b += 5)
\]

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

\[
&(a, b) \\
a, &b
\]

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. These two expressions are equivalent:

\[
(a ? b : c) = 5 \\
(a ? b = 5 : (c = 5))
\]

A cast is a valid lvalue if its operand is an lvalue. This extension is deprecated. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if \(a\) has type `char *`, these two expressions are equivalent:

\[
(int)a = 5 \\
(int)(a = (char *)(int)5)
\]

An arithmetic operation such as `+=` applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

\[
(int)a += 5 \\
(int)(a = (char *)(int) ((int)a + 5))
\]

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that `&(int)f` were permitted, where \(f\) has type `float`. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

\[
*&(int)f = 1;
\]

This is quite different from what `(int)f = 1` would do; it would convert 1 to floating point and store it. Rather than cause this inconsistency, we think it is better to prohibit use of `&` on a cast.

If you really do want an `int *` pointer with the address of \(f\), you can simply write `(int *)&f`. 
5.3.3 **Structures with no members**

GCC permits a C structure to have no members:

```c
struct empty {
};
```

The structure will have size zero.

**NOTE** In C++, empty structures are part of the language. G++ treats empty structures as if they had a single member of type `char`.

5.3.4 **Non-constant initializers**

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```c
foo (float f, float g)
{
    float beat_freqs[2] = { f-g, f+g };
    /* ... */
}
```

5.3.5 **Designated initializers**

Standard C89 requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In ISO C99 you can give the elements in any order, specifying the array indices or structure field names they apply to, and GNU C allows this as an extension in C89 mode as well.

To specify an array index, write `{index} =` before the element value. For example:

```c
```

This is equivalent to:

```c
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

An alternative obsolete syntax for this which GCC accepts is to write `{index} =` before the element value, with no `=`.

To initialize a range of elements to the same value, write `{first ... last} = value`. This is a GNU extension. For example:

```c
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```
If the value in it has side-effects, the side-effects will happen only once, not for each initialized field by the range initializer.

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with `.fieldname` = before the element value. For example, given the following structure definition the two subsequent initializations are equivalent:

```c
struct point { int x, y; }

struct point p = { .y = yvalue, .x = xvalue };
struct point p = { xvalue, yvalue };
```

Another form, which is obsolete and has the same meaning, is “fieldname:”. For example:

```c
struct point p = { y: yvalue, x: xvalue };
```

The `[index]` or `.fieldname` is known as a designator. You can also use a designator (or the obsolete colon syntax) when initializing a union, to specify which element of the union should be used. For example:

```c
union foo { int i; double d; }

union foo f = { .d = 4 };
```

This converts 4 to a double to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer i, since it is an integer. (See Section 5.2.22.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a designator applies to the next consecutive element of the array or structure. For example, the following two statements are equivalent:

```c
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example:

```c
int whitespace[256]
    = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
        ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

You can also write a series of `.fieldname` and `[index]` designators before an `=` to specify a nested subobject to initialize; the list is taken relative to the subobject corresponding to the closest surrounding brace pair. For example, with the `struct point` declaration above:

```c
struct point ptarray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

If the same field is initialized multiple times, it will be initialized with the value from the last initialization, overriding the previous initializations. If any such overridden initialization has side-effects, it is unspecified whether the side-effects happen or not. Currently, GCC will discard them and issue a warning.
5.3.6 **Mixed declarations and code**

ISO C99 and ISO C++ allow declarations and code to be freely mixed within compound statements. As an extension, GCC also allows this in C89 mode. For example, you can do:

```c
int i;
/* ... */
i++;
int j = i + 2;
```

Each identifier is visible from where it is declared until the end of the enclosing block.

5.3.7 **Prototypes and old-style function definitions**

GNU C extends ISO C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```c
/* Use prototypes unless the compiler is old-fashioned. */
#ifdef __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration. */
int isroot P((uid_t));

/* Old-style function definition. */
int isroot (x) /* ??? lossage here ??? */
uid_t x;
{ return x == 0;
}
```

Suppose the type `uid_t` happens to be `short`. ISO C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition’s argument is really an `int`, which does not match the prototype argument type of `short`.

This restriction of ISO C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the `uid_t` type is `short`, `int`, or `long`. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion.
Thus in GNU C the above example is equivalent to the following:

```c
int isroot (uid_t);

int
isroot (uid_t x)
{
    return x == 0;
}
```

### 5.3.8 C++ style comments

In GNU C you can use C++ style comments which start with `//` and continue until the end of the line. Many other C implementations allow such comments, and they are included in the 1999 C standard. However, C++ style comments are not recognized if you specify an `-std` option with a version of ISO C before C99, or `-ansi` (equivalent to `-std=c89`).

### 5.3.9 Incomplete enum types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can’t allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.
5.4 C++-specific extensions

The C++-specific language extensions are available only in the C++ compiler.

5.4.1 When is a volatile object accessed?

Both the C and C++ standard have the concept of volatile objects. These are normally accessed by pointers and used for accessing hardware. The standards encourage compilers to refrain from optimizations concerning accesses to volatile objects that it might perform on non-volatile objects:

- The C standard leaves it implementation-defined as to what constitutes a volatile access.
- The C++ standard omits to specify this, except to say that C++ should behave in a similar manner to C with respect to volatiles, where possible.

The minimum either standard specifies is that at a sequence point all previous accesses to volatile objects have stabilized and no subsequent accesses have occurred. Thus an implementation is free to reorder and combine volatile accesses which occur between sequence points, but cannot do so for accesses across a sequence point. The use of volatiles does not allow you to violate the restriction on updating objects multiple times within a sequence point.

In most expressions, it is intuitively obvious what is a read and what is a write. For example:

```c
volatile int *dst = somevalue;
volatile int *src = someothervalue;
*dst = *src;
```

This loads the volatile object pointed to by `src` and stores the value into the volatile object pointed to by `dst`. There is no guarantee that these reads and writes are atomic, especially for objects larger than `int`.

Less obvious expressions are where something which looks like an access is used in a void context. For example:

```c
volatile int *src = somevalue;
*src;
```

With C, such expressions are rvalues, and as rvalues cause a read of the object, GCC interprets this as a read of the volatile being pointed to. The C++ standard specifies that such expressions do not undergo lvalue to rvalue conversion, and that the type of the dereferenced object may be incomplete. The C++ standard does not specify explicitly that it is this lvalue to rvalue conversion which is responsible for causing an access. However, there is reason to believe that it is, because otherwise certain simple expressions become undefined. However, because it would surprise most programmers, G++ treats dereferencing a pointer to volatile object of complete type in a void context as a read of the object. When the object has incomplete type, G++ issues a warning. For example:

```c
struct S;
```
struct T { int m; };  
volatile S *ptr1 = somevalue;  
volatile T *ptr2 = somevalue;  
*ptr1;  
*ptr2;

In this example, a warning is issued for *ptr1, and *ptr2 causes a read of the object pointed to. If you wish to force an error on the first case, you must force a conversion to rvalue with, for instance a static cast, static_cast<S>(*ptr1).

When using a reference to volatile, G++ does not treat equivalent expressions as accesses to volatiles, but instead issues a warning that no volatile is accessed. The rationale for this is that otherwise it becomes difficult to determine where volatile access occur, and not possible to ignore the return value from functions returning volatile references. Again, if you wish to force a read, cast the reference to an rvalue.

5.4.2 Restricting pointer aliasing

G++ understands the C99 feature of restricted pointers, specified with the __restrict__, or __restrict type qualifier. Because you cannot compile C++ by specifying the -std=c99 language flag, restrict is not a keyword in C++.

In addition to allowing restricted pointers, you can specify restricted references, which indicate that the reference is not aliased in the local context. For example:

```c
void fn (int *__restrict__ rptr, int &__restrict__ rref)
{
    /* ... */
}
```

In the body of fn, rptr points to an unaliased integer and rref refers to a (different) unaliased integer.

You may also specify whether a member function’s this pointer is unaliased by using __restrict__ as a member function qualifier. For example:

```c
void T::fn () __restrict__
{
    /* ... */
}
```

Within the body of T::fn, this will have the effective definition T *__restrict__ const this. Notice that the interpretation of a __restrict__ member function qualifier is different to that of const or volatile qualifier, in that it is applied to the pointer rather than the object. This is consistent with other compilers which implement restricted pointers.

As with all outermost parameter qualifiers, __restrict__ is ignored in function definition matching. This means you only need to specify __restrict__ in a function definition, rather than in a function prototype as well.
5.4.3 Vague linkage

Several constructs are defined in C++ which require space in the object file but are not clearly tied to a single translation unit. We say that these constructs have “vague linkage”. Typically such constructs are emitted wherever they are needed, though sometimes we can be more clever.

5.4.3.1 Inline functions

Inline functions are typically defined in a header file which can be included in many different compilations. Hopefully they can usually be inlined, but sometimes an out-of-line copy is necessary, if the address of the function is taken or if inlining fails. In general, we emit an out-of-line copy in all translation units where one is needed. As an exception, we only emit inline virtual functions with the vtable, since it will always require a copy.

Local static variables and string constants used in an inline function are also considered to have vague linkage, since they must be shared between all inlined and out-of-line instances of the function.

5.4.3.2 VTables

C++ virtual functions are implemented in most compilers using a lookup table, known as a vtable. The vtable contains pointers to the virtual functions provided by a class, and each object of the class contains a pointer to its vtable (or vtables, in some multiple-inheritance situations). If the class declares any non-inline, non-pure virtual functions, the first one is chosen as the “key method” for the class, and the vtable is only emitted in the translation unit where the key method is defined.

**NOTE** If the chosen key method is later defined as inline, the vtable will still be emitted in every translation unit which defines it. Make sure that any inline virtuals are declared inline in the class body, even if they are not defined there.

5.4.3.3 type_info objects

C++ requires information about types to be written out in order to implement `dynamic_cast`, `typeid` and exception handling. For polymorphic classes (classes with virtual functions), the type_info object is written out along with the vtable so that `dynamic_cast` can determine the dynamic type of a class object at runtime. For all other types, we write out the type_info object when it is used: when applying `typeid` to an expression, throwing an object, or referring to a type in a catch clause or exception specification.

5.4.3.4 Template instantiations

Most everything in this section also applies to template instantiations, but there are other options as well. See Section 5.4.5.
When used with GNU ld version 2.8 or later on an ELF system such as GNU/Linux or Solaris 2, or on Microsoft Windows, duplicate copies of these constructs will be discarded at link time. This is known as COMDAT support.

On targets that don’t support COMDAT, but do support weak symbols, GCC will use them. This way one copy will override all the others, but the unused copies will still take up space in the executable.

For targets which do not support either COMDAT or weak symbols, most entities with vague linkage will be emitted as local symbols to avoid duplicate definition errors from the linker. This will not happen for local statics in inlines, however, as having multiple copies will almost certainly break things.

See Section 5.4.4 for another way to control placement of these constructs.

### 5.4.4 #pragma interface and implementation

#pragma interface and #pragma implementation provide the user with a way of explicitly directing the compiler to emit entities with vague linkage (and debugging information) in a particular translation unit.

#### NOTE
These #pragmas are not useful in most cases, because of COMDAT support and the “key method” heuristic mentioned in Section 5.4.3. Using them can actually cause your program to grow due to unnecessary out-of-line copies of inline functions. Currently the only benefit of these #pragmas is reduced duplication of debugging information, and that should be addressed soon on DWARF 2 targets with the use of COMDAT sections.

#pragma interface ["subdir/objects.h"]
Use this directive in header files that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing #pragma interface is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses #pragma implementation). Instead, the object files will contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to #pragma implementation.
#pragma implementation ["objects.h"]

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use #pragma interface. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use #pragma implementation with no argument, it applies to an include file with the same basename\(^1\) as your source file. For example, in `allclass.cc`, giving just #pragma implementation by itself is equivalent to #pragma implementation "allclass.h".

In versions of GNU C++ prior to 2.6.0 `allclass.h` was treated as an implementation file whenever you would include it from `allclass.cc` even if you never specified #pragma implementation. This was deemed to be more trouble than it was worth, however, and disabled.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use #include to include the header file; #pragma implementation only specifies how to use the file, it doesn’t actually include it.)

There is no way to split up the contents of a single header file into multiple implementation files.

#pragma implementation and #pragma interface also have an effect on function inlining.

If you define a class in a header file marked with #pragma interface, the effect on an inline function defined in that class is similar to an explicit extern declaration: the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as #pragma implementation, the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with -fno-implement-inlines. If any calls were not inlined, you will get linker errors.

---

\(^1\) A file’s *basename* is the name stripped of all leading path information and of trailing suffixes, such as .h or .C or .cc.
5.4.5  Where’s the template?

C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system. Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which are referred to as the Borland model and the Cfront model.

Borland model

Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; the compiler emits template instances in each translation unit that uses them, and the linker collapses them together. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. This disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all templates in the header file, since they must be seen to be instantiated.

Cfront model

The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. A more modern version of the repository works as follows: As individual object files are built, the compiler places any template definitions and instantiations encountered in the repository. At link time, the link wrapper adds in the objects in the repository and compiles any needed instances that were not previously emitted. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; for some code this can be just as transparent, but in practice it can been very difficult to build multiple programs in one directory and one program in multiple directories. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which should be compiled separately.

When used with GNU ld version 2.8 or later on an ELF system such as GNU/Linux or Solaris 2, or on Microsoft Windows, G++ supports the Borland model. On other systems, G++ implements neither automatic model.

A future version of G++ will support a hybrid model whereby the compiler will emit any instantiations for which the template definition is included in the compile, and store template definitions and instantiation context information into the object file for the rest. The link wrapper will extract that information as necessary and invoke the compiler to produce the remaining instantiations. The linker will then combine duplicate instantiations.
In the mean time, you have the following options for dealing with template instantiations:

1. Compile your template-using code with `-frepo`. The compiler will generate files with the extension `.rpo` listing all of the template instantiations used in the corresponding object files which can be instantiated there; the link wrapper, `collect2`, will then update the `.rpo` files to tell the compiler where to place those instantiations and rebuild any affected object files. The link-time overhead is negligible after the first pass, as the compiler will continue to place the instantiations in the same files.

   This is your best option for application code written for the Borland model, as it will just work. Code written for the Cfront model will need to be modified so that the template definitions are available at one or more points of instantiation; usually this is as simple as adding `#include <tmethods.cc>` to the end of each template header.

   For library code, if you want the library to provide all of the template instantiations it needs, just try to link all of its object files together; the link will fail, but cause the instantiations to be generated as a side effect. Be warned, however, that this may cause conflicts if multiple libraries try to provide the same instantiations. For greater control, use explicit instantiation as described in the next option.

2. Compile your code with `-fno-implicit-templates` to disable the implicit generation of template instances, and explicitly instantiate all the ones you use. This approach requires more knowledge of exactly which instances you need than do the others, but it’s less mysterious and allows greater control. You can scatter the explicit instantiations throughout your program, perhaps putting them in the translation units where the instances are used or the translation units that define the templates themselves; you can put all of the explicit instantiations you need into one big file; or you can create small files for each of the instances you need (as shown in the following example) and create a template instantiation library from them:

   ```
   #include "Foo.h"
   #include "Foo.cc"

   template class Foo<int>;
   template ostream& operator <<
   (ostream&, const Foo<int>&);
   ```

   If you are using Cfront-model code, you can probably get away with not using `-fno-implicit-templates` when compiling files that don’t `#include` the member template definitions.

   If you use one big file to do the instantiations, you may want to compile it without `-fno-implicit-templates` so you get all of the instances required by your explicit instantiations (but not by any other files) without having to specify them as well.
G++ has extended the template instantiation syntax given in the ISO standard to allow forward declaration of explicit instantiations (with `extern`), instantiation of the compiler support data for a template class (i.e. the vtable) without instantiating any of its members (with `inline`), and instantiation of only the static data members of a template class, without the support data or member functions (with `static`). For example:

```cpp
extern template int max (int, int);
inline template class Foo<int>;
static template class Foo<int>;
```

3. Do nothing. Pretend G++ does implement automatic instantiation management. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates it uses. In a large program, this can lead to an unacceptable amount of code duplication.

### 5.4.6 Extracting the function pointer from a bound pointer to member function

In C++, pointer to member functions (PMFs) are implemented using a wide pointer of sorts to handle all the possible call mechanisms; the PMF needs to store information about how to adjust the `this` pointer, and if the function pointed to is virtual, where to find the vtable, and where in the vtable to look for the member function. If you are using PMFs in an inner loop, you should really reconsider that decision. If that is not an option, you can extract the pointer to the function that would be called for a given object/PMF pair and call it directly inside the inner loop, to save a bit of time.

Note that you will still be paying the penalty for the call through a function pointer; on most modern architectures, such a call defeats the branch prediction features of the CPU. This is also true of normal virtual function calls.

The syntax for this extension is:

```cpp
extern A a;
extern int (A::*fp)();
typedef int (*fptr)(A *);
fptr p = (fptr)(a.*fp);
```

For PMF constants (i.e. expressions of the form `&Klasse::Member`), no object is needed to obtain the address of the function. They can be converted to function pointers directly:

```cpp
fptr p1 = (fptr)(&A::foo);
```

You must specify option `-Wno-pmf-conversions` to use this extension.
5.4.7 **C++-specific attributes**

Some attributes (Section 5.2.24) only make sense for C++ programs.

See Section 5.4.8 for an additional attribute.

**init_priority** *(priority)*

In standard C++, objects defined at namespace scope are guaranteed to be initialized in an order in strict accordance with that of their definitions *in a given translation unit*. No guarantee is made for initializations across translation units. However, GNU C++ allows users to control the order of initialization of objects defined at namespace scope with the `init_priority` attribute by specifying a relative `priority`, a constant integral expression currently bounded between 101 and 65535 inclusive. Lower numbers indicate a higher priority.

In the following example, A would normally be created before B, but the `init_priority` attribute has reversed that order:

```c
Some_Class A __attribute__((init_priority (2000)));
Some_Class B __attribute__((init_priority (543)));
```

Note that the particular values of `priority` do not matter; only their relative ordering.
visibility ("visibility_type")

Cause every member function of a class declaration to be emitted with the specified visibility (default, hidden, protected, or internal).

For example:

```c
class __attribute__((visibility("hidden"))) foo
{
  int foo1();
  int foo2();
}
```

See the ELF gABI for complete details. Here is a summary:

- **default**: Default visibility is the normal case for ELF. This value is available for the visibility attribute to override other options that may change the assumed visibility of symbols.

- **hidden**: Hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so no other module (executable or shared library) can reference it directly.

- **protected**: Protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind to the local symbol. That is, the symbol cannot be overridden by another module.

- **internal**: Internal visibility is like hidden visibility, but with additional processor specific semantics. GCC defines internal visibility to mean that the function is never called from another module. Note that hidden symbols, while they cannot be referenced directly by other modules, can be referenced indirectly via function pointers. By indicating that a symbol cannot be called from outside the module, GCC may for instance omit the load of a PIC register since it is known that the calling function loaded the correct value.

The syntax is similar to the `__declspec()` system used by Microsoft Windows compilers, enabling cross-platform projects to reuse their existing macro system for denoting imports and exports.

**NOTE** PLT indirection overhead can be eliminated by marking as hidden any internal classes that are never used outside a binary.
5.4.8 Strong using

**CAUTION** The semantics of this extension are not fully defined. Users should refrain from using this extension as its semantics may change subtly over time. It is possible that this extension will be removed in future versions of G++.

A using-directive with `__attribute__((strong))` is stronger than a normal using-directive in two ways:

- Templates from the used namespace can be specialized as though they were members of the using namespace.
- The using namespace is considered an associated namespace of all templates in the used namespace for purposes of argument-dependent name lookup.

This is useful for composing a namespace transparently from implementation namespaces. For example:

```c++
namespace std {
    namespace debug {
        template <class T> struct A { }
    }
    using namespace debug __attribute__((strong));
    template <> struct A<int> { }; // ok to specialize
    template <class T> void f (A<T>);
}

int main()
{
    f (std::A<float>());       // lookup finds std::f
    f (std::A<int>());
}
```

5.4.9 Deprecated features

In the past, the GNU C++ compiler was extended to experiment with new features, at a time when the C++ language was still evolving. Now that the C++ standard is complete, some of those features are superseded by superior alternatives. Using the old features might cause a warning in some cases that the feature will be dropped in the future. In other cases, the feature might be gone already.

While the list below is not exhaustive, it documents some of the options that are now deprecated:

- `-fexternal-templates`
- `-falt-external-templates`

These are two of the many ways for G++ to implement template instantiation. See Section 5.4.5. The C++ standard clearly defines how template definitions have to be organized across implementation units. G++ has an implicit instantiation mechanism that should work just fine for standard-conforming code.
-fstrict-prototype
-fno-strict-prototype

Previously it was possible to use an empty prototype parameter list to indicate an unspecified number of parameters (like C), rather than no parameters, as C++ demands. This feature has been removed, except where it is required for backwards compatibility. See Section 5.4.10.

The named return value extension has been deprecated, and is now removed from G++.

The use of initializer lists with new expressions has been deprecated, and is now removed from G++.

Floating and complex non-type template parameters have been deprecated, and are now removed from G++.

The implicit typename extension has been deprecated and is now removed from G++.

The use of default arguments in function pointers, function typedefs and and other places where they are not permitted by the standard is deprecated and will be removed from a future version of G++.

5.4.10 Backwards compatibility

Now that there is a definitive ISO standard C++, G++ has a specification to adhere to. The C++ language evolved over time, and features that used to be acceptable in previous drafts of the standard, such as the ARM® [Annotated C++ Reference Manual], are no longer accepted. In order to allow compilation of C++ written to such drafts, G++ contains some backwards compatibilities. All such backwards compatibility features are liable to disappear in future versions of G++. They should be considered deprecated. See Section 5.4.9.

For scope

If a variable is declared at for scope, it used to remain in scope until the end of the scope which contained the for statement (rather than just within the for scope). G++ retains this, but issues a warning, if such a variable is accessed outside the for scope.

Implicit C language

Old C system header files did not contain an extern "C" { . . . } scope to set the language. On such systems, all header files are implicitly scoped inside a C language scope. Also, an empty prototype () will be treated as an unspecified number of arguments, rather than no arguments, as C++ demands.
5.5 Processor-specific extensions

The Hexagon processor-specific language extensions support the following features:

- Predefined symbols
- Memory access
- Instruction access
- Vector access (32- and 64-bit; C and C++)
- Circular addressing
- Bit-reversed addressing

**NOTE** For more information on the processor features related to these extensions see the *Hexagon V4 Programmer's Reference Manual*.

5.5.1 Predefined symbols

GCC predefines a number of symbols to simplify the development of portable software. These symbols can be used in conditional statements to selectively include or exclude processor-specific sections of code. For example:

```c
#if __hexagon__
...              // Hexagon processor-specific code
#endif
```

Table 5-2 lists the symbols and how they are defined when compiling code for specific versions of the Hexagon processor (Section 3.4.16).

**Table 5-2** Version-specific predefined symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hexagon</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>HEXAGON_V2</strong></td>
<td>1</td>
<td>symbol not defined</td>
<td>symbol not defined</td>
</tr>
<tr>
<td><strong>HEXAGON_V3</strong></td>
<td>symbol not defined</td>
<td>1</td>
<td>symbol not defined</td>
</tr>
<tr>
<td><strong>HEXAGON_V4</strong></td>
<td>symbol not defined</td>
<td>symbol not defined</td>
<td>1</td>
</tr>
<tr>
<td><strong>HEXAGON_ARCH</strong></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**NOTE** When defined these symbols are preprocessor macros which evaluate to the values listed in Table 5-2.
5.5.2 Memory access

GCC includes predefined sections which are used to assign program functions or variables to the following processor-specific (or MSM-specific) memories:

- External bus interface (EBI)
- Tightly-coupled memory (TCM)
- Stacked memory interface (SMI)

In addition to specifying the memory, the predefined sections also specify the cache properties of the stored code/data.

Table 5-3 lists the predefined section names for the Hexagon processor memories.

Table 5-3 Hexagon processor memory sections

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ebi_code_cached</td>
<td>Code</td>
</tr>
<tr>
<td>.tcm_code_cached</td>
<td></td>
</tr>
<tr>
<td>.smi_code_cached</td>
<td></td>
</tr>
<tr>
<td>.ebi_data_cached</td>
<td>Data (cached)</td>
</tr>
<tr>
<td>.tcm_data_cached</td>
<td></td>
</tr>
<tr>
<td>.smi_data_cached</td>
<td></td>
</tr>
<tr>
<td>.ebi_data_cached_wt</td>
<td>Data (write-through cached)</td>
</tr>
<tr>
<td>.tcm_data_cached_wt</td>
<td></td>
</tr>
<tr>
<td>.smi_data_cached_wt</td>
<td></td>
</tr>
<tr>
<td>.ebi_data_uncached</td>
<td>Data (uncached)</td>
</tr>
<tr>
<td>.tcm_data_uncached</td>
<td></td>
</tr>
<tr>
<td>.smi_data_uncached</td>
<td></td>
</tr>
</tbody>
</table>

For example:

```c
int ainit[4] __attribute__((section(".ebi_data_cached_wt"))) = {1, 2, 3, 4};
void foo() __attribute__((section(".tcm_code_cached")));  
```

For more information on the Hexagon processor memories and memory cache see the *Hexagon V4 Programmer’s Reference Manual*.

**NOTE** All variables assigned to uncached memory must be declared as volatile. This programming restriction is used to enforce a Hexagon processor restriction that data stored in uncached memory cannot be accessed in parallel with any other memory data. For more information see Section 4.11.

Functions and variables assigned to the .text and .data sections (Section 5.2.25, Section 5.2.27) are automatically assigned to EBI memory. Program code and data cannot be stored in the same section.
5.5.3 Instruction access

To support efficient coding of the time-intensive sections of a program (without resorting to inline assembly language), GCC provides intrinsics which are used to directly access Hexagon processor instructions.

The instruction intrinsics are accessed by including the library header file `hexagon_protos.h`.

The following example shows how an instruction intrinsic is used to directly access the ALU64 instruction `Rdd=vminh(Rtt,Rss)`:

```c
#include <hexagon_protos.h>

int main()
{
    long long v1=0xFFFF0000FFFF0000;
    long long v2=0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions. For more information see the Instructions chapter of the `Hexagon V4 Programmer’s Reference Manual`.

5.5.4 Vector access (32-bit)

To support efficient coding of 32-bit vector operations, GCC provides intrinsics which are used to perform the following tasks:

- Read or write individual words, half-words, or bytes in 32-bit vectors
- Construct 32-bit vectors from a sequence of words, half-words, or bytes

The vector intrinsics are accessed by including the header file `hexagon_types.h`. 32-bit vectors are defined as variables of type `Q6Vect32`. 
The following example shows how the 32-bit vector intrinsics are used:

```c
#include <hexagon_types.h>

int main()
{
    Word16 w=1, x=3;
    Word32 t=1;
    unsigned char b;
    Q6Vect32 v, rslt;

    // construct 32-bit vector from two half-words
    v = Q6V32_CREATE_H(w, x);
    // read low-order byte of 32-bit vector
    b = Q6V32_GET_UB0(v);
    // write word as 32-bit vector
    // return modified vector
    // v remains unchanged
    rslt = Q6V32_PUT_W(v, t);
}
```

**Figure 5-1** shows how words, half-words, and bytes are accessed in a 32-bit vector.

**Figure 5-1**  Vector access (32-bit)
5.5.4.1 Construct vector

Construct a 32-bit vector from the specified values and return it as a result value.

Q6Vect32 Q6V32_CREATE_W(int w);
Q6Vect32 Q6V32_CREATE_H(short h1, short h0);
Q6Vect32 Q6V32_CREATE_B(char b3, char b2, char b1, char b0);

5.5.4.2 Extract word

Extract word (either signed or unsigned) from a 32-bit vector and return it as a result value.

int Q6V32_GET_W(Q6Vect32 v);
unsigned int Q6V32_GET_UW(Q6Vect32 v);

5.5.4.3 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 32-bit vector and return it as a result value.

short Q6V32_GET_H0(Q6Vect32 v);
short Q6V32_GET_H1(Q6Vect32 v);
unsigned short Q6V32_GET_UH0(Q6Vect32 v);
unsigned short Q6V32_GET_UH1(Q6Vect32 v);

5.5.4.4 Extract byte

Extract the specified byte (either signed or unsigned) from a 32-bit vector and return it as a result value.

signed char Q6V32_GET_B0(Q6Vect32 v);
signed char Q6V32_GET_B1(Q6Vect32 v);
signed char Q6V32_GET_B2(Q6Vect32 v);
signed char Q6V32_GET_B3(Q6Vect32 v);
unsigned char Q6V32_GET_UB0(Q6Vect32 v);
unsigned char Q6V32_GET_UB1(Q6Vect32 v);
unsigned char Q6V32_GET_UB2(Q6Vect32 v);
unsigned char Q6V32_GET_UB3(Q6Vect32 v);

5.5.4.5 Set word

Assign word value to a 32-bit vector and return the modified vector as a result value.

Q6Vect32 Q6V32_PUT_W(Q6Vect32 v, int val);
5.5.4.6 **Set half-word**

Assign a value to the specified half-word in a 32-bit vector and return the modified vector as a result value.

\[
\begin{align*}
\text{Q6Vect32} & \text{ Q6V32\_PUT\_H0(Q6Vect32 } v, \text{ short val);} \\
\text{Q6Vect32} & \text{ Q6V32\_PUT\_H1(Q6Vect32 } v, \text{ short val);}
\end{align*}
\]

5.5.4.7 **Set byte**

Assign a value to the specified byte in a 32-bit vector and return the modified vector as a result value.

\[
\begin{align*}
\text{Q6Vect32} & \text{ Q6V32\_PUT\_B0(Q6Vect32 } v, \text{ char val);} \\
\text{Q6Vect32} & \text{ Q6V32\_PUT\_B1(Q6Vect32 } v, \text{ char val);} \\
\text{Q6Vect32} & \text{ Q6V32\_PUT\_B2(Q6Vect32 } v, \text{ char val);} \\
\text{Q6Vect32} & \text{ Q6V32\_PUT\_B3(Q6Vect32 } v, \text{ char val);}
\end{align*}
\]

5.5.5 **Vector access (32-bit; C++)**

For C++ programs the vector access library also implements the 32-bit vector access operations (Section 5.5.4) as member functions of the class Q6Vect32C.

5.5.5.1 **Constructors**

Construct a 32-bit vector from the specified values.

\[
\begin{align*}
\text{Q6Vect32C:Q6Vect32C(int } w = 0); \\
\text{Q6Vect32C:Q6Vect32C(short h1, short h0);} \\
\text{Q6Vect32C:Q6Vect32C(signed char b3, signed char b2,} \\
\text{ signed char b1, signed char b0);} \\
\text{Q6Vect32C:Q6Vect32C(const Q6Vect32C & } v); \\
\text{Q6Vect32C:Q6Vect32C& operator = (const Q6Vect32C & } v);
\end{align*}
\]

5.5.5.2 **Extract word**

Extract word (either signed or unsigned) from a 32-bit vector and return it as a result value.

\[
\begin{align*}
\text{int Q6Vect32C:W(void);} \\
\text{unsigned int Q6Vect32C:UW(void);}
\end{align*}
\]
5.5.5.3 **Extract half-word**

Extract the specified half-word (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
short Q6Vect32C:H0(void);
short Q6Vect32C:H1(void);
unsigned short Q6Vect32C:UH0(void);
unsigned short Q6Vect32C:UH1(void);
```

5.5.5.4 **Extract byte**

Extract the specified byte (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
signed char Q6Vect32C:B0(void);
signed char Q6Vect32C:B1(void);
signed char Q6Vect32C:B2(void);
signed char Q6Vect32C:B3(void);
unsigned char Q6Vect32C:UB0(void);
unsigned char Q6Vect32C:UB1(void);
unsigned char Q6Vect32C:UB2(void);
unsigned char Q6Vect32C:UB3(void);
```

5.5.5.5 **Set word**

Assign word value to a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C:W(int w);
```

5.5.5.6 **Set half-word**

Assign a value to the specified half-word in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C:H0(short h);
Q6Vect32C Q6Vect32C:H1(short h);
```

5.5.5.7 **Set byte**

Assign a value to the specified byte in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C:B0(signed char b);
Q6Vect32C Q6Vect32C:B1(signed char b);
Q6Vect32C Q6Vect32C:B2(signed char b);
Q6Vect32C Q6Vect32C:B3(signed char b);
```
5.5.6 Vector access (64-bit)

To support efficient coding of 64-bit vector operations, GCC provides intrinsics which are used to perform the following tasks:

- Read or write individual double-words, words, half-words, or bytes in 64-bit vectors
- Construct 64-bit vectors from a sequence of double-words, words, half-words, or bytes

The vector intrinsics are accessed by including the header file `hexagon_types.h`.

64-bit vectors are defined as variables of type `Q6Vect64`.

The following example shows how the 64-bit vector intrinsics are used:

```c
#include <hexagon_types.h>

int main()
{
    Word16 w=1, x=3, y=2, z=1;
    Word32 t=1;
    unsigned char b;
    Q6Vect64 v, rslt;

    // construct 64-bit vector from four half-words
    v = Q6V64_CREATE_H(w, x, y, z);

    // read low-order byte of 64-bit vector
    b = Q6V64_GET_UB0(v);

    // write high-order word of 64-bit vector
    // return modified vector
    // v remains unchanged
    rslt = Q6V64_PUT_W1(v, t);
}
```
Figure 5-2 shows how double-words, words, half-words, and bytes are accessed in a vector.

![Figure 5-2 Vector access (64-bit)](image)

5.5.6.1 **Construct vector**

Construct a 64-bit vector from the specified values and return it as a result value.

```c
Q6Vect64 Q6V64_CREATE_D(long long d);
Q6Vect64 Q6V64_CREATE_W(int w1, int w0);
Q6Vect64 Q6V64_CREATE_H(short h3, short h2, short h1, short h0);
Q6Vect64 Q6V64_CREATE_B(char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0);
```

5.5.6.2 **Extract double-word**

Extract double-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```c
long long Q6V64_GET_D(Q6Vect64 v);
unsigned long long Q6V64_GET_UD(Q6Vect64 v);
```
### 5.5.6.3 Extract word

Extract the specified word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```c
int Q6V64_GET_W0(Q6Vect64 v);
int Q6V64_GET_W1(Q6Vect64 v);

unsigned int Q6V64_GET_UW0(Q6Vect64 v);
unsigned int Q6V64_GET_UW1(Q6Vect64 v);
```

### 5.5.6.4 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```c
short Q6V64_GET_H0(Q6Vect64 v);
short Q6V64_GET_H1(Q6Vect64 v);
short Q6V64_GET_H2(Q6Vect64 v);
short Q6V64_GET_H3(Q6Vect64 v);

unsigned short Q6V64_GET_UH0(Q6Vect64 v);
unsigned short Q6V64_GET_UH1(Q6Vect64 v);
unsigned short Q6V64_GET_UH2(Q6Vect64 v);
unsigned short Q6V64_GET_UH3(Q6Vect64 v);
```

### 5.5.6.5 Extract byte

Extract the specified byte (either signed or unsigned) from a 64-bit vector and return it as a result value.

```c
signed char Q6V64_GET_B0(Q6Vect64 v);
signed char Q6V64_GET_B1(Q6Vect64 v);
signed char Q6V64_GET_B2(Q6Vect64 v);
signed char Q6V64_GET_B3(Q6Vect64 v);
signed char Q6V64_GET_B4(Q6Vect64 v);
signed char Q6V64_GET_B5(Q6Vect64 v);
signed char Q6V64_GET_B6(Q6Vect64 v);
signed char Q6V64_GET_B7(Q6Vect64 v);

unsigned char Q6V64_GET_UB0(Q6Vect64 v);
unsigned char Q6V64_GET_UB1(Q6Vect64 v);
unsigned char Q6V64_GET_UB2(Q6Vect64 v);
unsigned char Q6V64_GET_UB3(Q6Vect64 v);
unsigned char Q6V64_GET_UB4(Q6Vect64 v);
unsigned char Q6V64_GET_UB5(Q6Vect64 v);
unsigned char Q6V64_GET_UB6(Q6Vect64 v);
unsigned char Q6V64_GET_UB7(Q6Vect64 v);
```
**5.5.6.6 Set double-word**

Assign double-word value to a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64 Q6V64_PUT_D(Q6Vect64 v, long long val);
```

**5.5.6.7 Set word**

Assign a value to the specified word in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64 Q6V64_PUT_W0(Q6Vect64 v, int val);
Q6Vect64 Q6V64_PUT_W1(Q6Vect64 v, int val);
```

**5.5.6.8 Set half-word**

Assign a value to the specified half-word in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64 Q6V64_PUT_H0(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H1(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H2(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H3(Q6Vect64 v, short val);
```

**5.5.6.9 Set byte**

Assign a value to the specified byte in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64 Q6V64_PUT_B0(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B1(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B2(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B3(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B4(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B5(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B6(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B7(Q6Vect64 v, char val);
```
5.5.7 Vector access (64-bit; C++)

For C++ programs the vector access library also implements the 64-bit vector access operations (Section 5.5.6) as member functions of the class Q6Vect64C.

5.5.7.1 Constructors

Construct a 64-bit vector from the specified values.

Q6Vect64C:Q6Vect64C(long long d = 0);
Q6Vect64C:Q6Vect64C(int w1, int w0);
Q6Vect64C:Q6Vect64C(short h3, short h2, short h1, short h0);
Q6Vect64C:Q6Vect64C(signed char b7, signed char b6, signed char b5, signed char b4, signed char b3, signed char b2, signed char b1, signed char b0);
Q6Vect64C:Q6Vect64C(const Q6Vect64C & v);
Q6Vect64C:Q6Vect64C& operator = (const Q6Vect64C & v);

5.5.7.2 Extract double-word

Extract double-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

long long Q6Vect64C:D(void);
unsigned long long Q6Vect64C:UD(void);

5.5.7.3 Extract word

Extract the specified word (either signed or unsigned) from a 64-bit vector and return it as a result value.

int Q6Vect64C:W0(void);
int Q6Vect64C:W1(void);
unsigned int Q6Vect64C:UW0(void);
unsigned int Q6Vect64C:UW1(void);

5.5.7.4 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

short Q6Vect64C:H0(void);
short Q6Vect64C:H1(void);
short Q6Vect64C:H2(void);
short Q6Vect64C:H3(void);
5.5.7.5 Extract byte

Extract the specified byte (either signed or unsigned) from a 64-bit vector and return it as a result value.

```c
signed char Q6Vect64C:B0(void);
signed char Q6Vect64C:B1(void);
signed char Q6Vect64C:B2(void);
signed char Q6Vect64C:B3(void);
signed char Q6Vect64C:B4(void);
signed char Q6Vect64C:B5(void);
signed char Q6Vect64C:B6(void);
signed char Q6Vect64C:B7(void);
unsigned char Q6Vect64C:UB0(void);
unsigned char Q6Vect64C:UB1(void);
unsigned char Q6Vect64C:UB2(void);
unsigned char Q6Vect64C:UB3(void);
unsigned char Q6Vect64C:UB4(void);
unsigned char Q6Vect64C:UB5(void);
unsigned char Q6Vect64C:UB6(void);
unsigned char Q6Vect64C:UB7(void);
```

5.5.7.6 Set double-word

Assign double-word value to a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64C Q6Vect64C:D(long long d);
```

5.5.7.7 Set word

Assign a value to the specified word in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64C Q6Vect64C:W0(int w);
Q6Vect64C Q6Vect64C:W1(int w);
```

5.5.7.8 Set half-word

Assign a value to the specified half-word in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64C Q6Vect64C:H0(short h);
Q6Vect64C Q6Vect64C:H1(short h);
Q6Vect64C Q6Vect64C:H2(short h);
Q6Vect64C Q6Vect64C:H3(short h);
```
5.5.7.9 **Set byte**

Assign a value to the specified byte in a 64-bit vector and return the modified vector as a result value.

```c
Q6Vect64C Q6Vect64C:B0(signed char b);
Q6Vect64C Q6Vect64C:B1(signed char b);
Q6Vect64C Q6Vect64C:B2(signed char b);
Q6Vect64C Q6Vect64C:B3(signed char b);
Q6Vect64C Q6Vect64C:B4(signed char b);
Q6Vect64C Q6Vect64C:B5(signed char b);
Q6Vect64C Q6Vect64C:B6(signed char b);
Q6Vect64C Q6Vect64C:B7(signed char b);
```

5.5.8 **Circular addressing**

**NOTE** The circular addressing intrinsics described in this section are deprecated. The Hexagon V4 processor architecture introduces a new form of circular addressing which will eventually supersede the circular addressing model described here.

To support data stream processing the Hexagon processor supports circular buffer addressing. Because it is difficult for compilers to generate efficient code for this address mode, the following intrinsics have been defined to efficiently support circular addressing:

- `Q6_circ_load_update_XX`
- `Q6_circ_store_update_XX`
- `Q6_circ_load_updateI_XX`
- `Q6_circ_store_updateI_XX`

These intrinsics perform both the circular buffer access (i.e., load or store) and the updating of the circular buffer pointer:

- The first two update the pointer with a *constant* increment/decrement value.
- The second two (which end with _I_XX) update the pointer with a *variable* increment/decrement value.

**NOTE** Variable increment/decrements support a larger value range than constant increment/decrements.

The _xx suffix specifies the type of data accessed by the intrinsic (byte, etc.).

The circular addressing intrinsics are accessed by including the library header file `hexagon_protos.h`.

For more information on the underlying circular address mode see the *Hexagon V3 Programmer’s Reference Manual*.

**Figure 5-3** shows an example of how the circular addressing intrinsics are used.
#include <stdio.h>
#include <hexagon_protos.h>

void circular(int input_array[], int output_array[], int element_load[],

    int *p0 = &input_array[20];
    int *p1 = output_array;
    int i;

    for (i = 0; i < 150; i++)
    {
        Q6_circ_load_update_W(element_load[i], p0, 1, 150, 8);
    }

    for (i = 0; i < 160; i++)
    {
        Q6_circ_store_update_W(i, p1, 1, 150, 8);
    }

#define N 150
int g_input_array[N] __attribute__((aligned(1024)));
int g_output_array[N] __attribute__((aligned(1024)));
//circular buffers should be global or static

int main()
{
    int i;
    int element_load[150];
    for (i = 0; i < N; i++)
    {
        g_input_array[i] = i;
    }

    circular(g_input_array, g_output_array, element_load);

    printf("\n-------element_load--------\n");
    for (i = 0; i < 150; i++)
    {
        printf("%d; ", element_load[i]);
        if((i%15)==14) printf("\n");
    }

    printf("\n-------g_output_array--------\n");
    for (i = 0; i < 150; i++)
    {
        printf("%d; ", g_output_array[i]);
        if((i%15)==14)printf("\n");
    }

    printf("\n-------end---------------\n");
}

Figure 5-3  Circular addressing program example
In the first for-loop of function `circular` (shown in Figure 5-3), pointer `p0` is used to load array `element_load` with the contents of circular buffer `input_array`.

`input_array` is initialized with the values 0 to 149, and 150 circular loads are performed on it; however, because the circular loads start from the 21st element in `input_array`, the values loaded into `element_load` initially range from 20 to 149, then (wrapping around to the front of `input_array`) continue from 0 to 19, as shown in Figure 5-4 below.

In the second for-loop of function `circular`, pointer `p1` is used to store the current value of the loop variable into circular buffer `output_array`.

The circular stores write 160 elements, which is 10 more than the circular buffer length. Because of this, the circular stores first write the values 0 to 149 to `output_array` (starting at index 0), then wrap around to the front of `output_array` and overwrite its first 10 elements (index 0 to 9) with the values 150 to 159, as shown in Figure 5-4 below.

Figure 5-4 shows the program output generated by the program in Figure 5-3.

```
--------element_load--------
20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34;
35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49;
50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62; 63; 64;
65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79;
80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94;
95; 96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109;
110; 111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124;
125; 126; 127; 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139;
140; 141; 142; 143; 144; 145; 146; 147; 148; 149; 0; 1; 2; 3; 4;
5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19;
--------g_output_array--------
150; 151; 152; 153; 154; 155; 156; 157; 158; 159; 10; 11; 12; 13; 14;
15; 16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29;
30; 31; 32; 33; 34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44;
45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59;
60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74;
75; 76; 77; 78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89;
90; 91; 92; 93; 94; 95; 96; 97; 98; 99; 100; 101; 102; 103; 104;
105; 106; 107; 108; 109; 110; 111; 112; 113; 114; 115; 116; 117; 118; 119;
120; 121; 122; 123; 124; 125; 126; 127; 128; 129; 130; 131; 132; 133; 134;
135; 136; 137; 138; 139; 140; 141; 142; 143; 144; 145; 146; 147; 148; 149;
--------end--------
```

**Figure 5-4**  Circular addressing program output
5.5.8.1 Circular load

Load value from circular buffer and update circular buffer pointer.

\[ \text{Q6\_circ\_load\_update\_XX}( \text{destination variable}, \text{buffer location pointer}, \text{increment/decrement value}, \text{buffer length}, \text{K1 value} ); \]

- The _XX suffix specifies the data type of the intrinsic (Section 5.5.8.4).
- The destination and buffer location pointer must be L-values.
- The increment/decrement value must be specified as a signed constant value, with one array element considered as unity (in conformance with ANSI C pointer increments). The value must be in the range -8 ... 7, and must have an absolute value less than the buffer length.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length can be between 3 and (128K-1) bytes.
- The K1 value determines the maximum possible buffer length (i.e., \(2^{K_1-1}\) elements, where \(2^{K_1-1} \leq \text{buffer length} \leq 2^{K_1-1}\)).

**NOTE** The intrinsic behavior is undefined if the specified K1 value is incompatible with the circular buffer length. For more information see the Hexagon V4 Programmer’s Reference Manual (PRM).

The K1 value differs slightly from the K value defined in the PRM:

- \(K_1 = K-1\) for 8-byte data types (e.g., long, long)
- \(K_1 = K\) for 4-byte data types (e.g., int)
- \(K_1 = K+1\) for 2-byte data types (e.g., short)
- \(K_1 = K+2\) for 1-byte data types (e.g., char)

**NOTE** If the circular load intrinsic is used only to increment the circular buffer pointer (i.e., a “dummy” load), the destination variable must be declared volatile to prevent the compiler from optimizing away the intrinsic call.

The circular intrinsics define increment/decrement values differently from the assembly-level circular instructions. For more information see the PRM.
Example

For an example of a circular load, consider the variable input_array, which is a circular buffer of 150 integer elements. Access from the first element (i.e., input_array[0]) to the 150th element (input_array[149]) is to be circular, with access initially starting from the 21st element (input_array[20]).

The pointer initialization statement would appear as follows:

```c
int *p0 = &input_array[20];
```

This C statement initializes the buffer location pointer (which will be used to perform circular access on the buffer) to the address of the 21st element of input_array.

The circular load statement would appear as follows:

```c
Q6_circ_load_update_W(x, p0, 1, 150, 8);
```

This C statement loads the data currently pointed by the pointer variable p0 into the integer variable x. The load intrinsic circularly post-increments the pointer variable by 1 (i.e., by one unit of the input_array base type (int)).

The circular buffer in the above example has the following properties:

- **buffer start address = input_array[0]**, because the current location that p0 points to (i.e., element input_array[20]) has element input_array[0] as the closest memory location that is aligned to 1024 bytes (the alignment requirement for circular buffer input_array; see Section 5.5.8.4), and that appears at or before it.

- **buffer length =150 integer words = 150 \times 4 \text{ bytes}**

If the increment/decrement value is a positive number a circular post-increment is performed. If it is a negative number a circular post-decrement is performed. If the circular pointer increment/decrement is not needed at the access point, then the increment/decrement value can be specified as 0:

```c
Q6_circ_load_update_W(x, p0, 0, 150, 8)
```

The buffer length can be a variable, but must have a known upper bound. In this case the K1 value is based on the upper bound of the buffer length.

For example, `Q6_circ_load_update_W(x, p0, 1, y, 8)` specifies that the buffer length variable y can specify a maximum length of 255 elements (since the K1 value is 8).

**NOTE**  The K1 value field is redundant if the buffer length is constant, but still must be assigned a valid K1 value.

Substituting “element_load[i]” for the parameter “x” makes the preceding example identical to part of the program example in Figure 5-3.
### 5.5.8.2 Circular store

Store value to circular buffer and update circular buffer pointer.

```c
Q6_circ_store_update_XX( <source expression>,
    <buffer location pointer>,
    <increment/decrement value>,
    <buffer length>,
    <K1 value> );
```

- The _XX suffix specifies the data type of the intrinsic (Section 5.5.8.4).
- The source expression specifies the value to be stored.
- The buffer location pointer must be an L-value.
- The increment/decrement value must be specified as a signed constant value, with one array element considered as unity (in conformance with ANSI C pointer increments). The value must be in the range -8 ... 7, and must have an absolute value less than the buffer length.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length can be between 3 and (128K-1) bytes.
- The K1 value determines the maximum possible buffer length (i.e., \(2^{K1-1}\) elements, where \(2^{K1-1} \leq\) buffer length \(\leq 2^{K1-1}\)).

**NOTE** The intrinsic behavior is undefined if the specified K1 value is incompatible with the circular buffer length. For more information see the Hexagon V4 Programmer’s Reference Manual (PRM).

The K1 value differs slightly from the K value defined in the PRM:

- \(K1 = K - 1\) for 8-byte data types (e.g., `long long`)
- \(K1 = K\) for 4-byte data types (e.g., `int`)
- \(K1 = K + 1\) for 2-byte data types (e.g., `short`)
- \(K1 = K + 2\) for 1-byte data types (e.g., `char`)

**NOTE** The circular intrinsics define increment/decrement values differently from the assembly-level circular instructions. For more information see the PRM.
**Example**

For an example of a circular store, consider the variable `output_array`, which is a circular buffer of 150 integer elements. Access from the first element (i.e., `output_array[0]`) to the 150th element (`output_array[149]`) is to be circular, with access initially starting from the first element (`output_array[0]`).

The pointer initialization statement would appear as follows:

```c
int *p1 = output_array[0];
```

This C statement initializes the buffer location pointer (that will be used to perform circular access on the buffer) to the address of the first element of `output_array`.

The circular store statement would appear as follows:

```c
Q6_circ_store_update_W(x, p1, 1, 150, 8);
```

This C statement loads the data currently pointed by the pointer variable `p1` into the integer variable `x`. The store intrinsic circularly post-increments the pointer variable by 1 (i.e., by one unit of the `output_array` base type (`int`)).

The circular buffer in the above example has the following properties:

- buffer start address = `&output_array[0]`, because the current location that `p1` points to (i.e., element `output_array[0]`) has element `output_array[0]` as the closest memory location that is aligned to 1024 bytes (the alignment requirement for circular buffer `output_array`; see Section 5.5.8.4), and that appears at or before it.
- buffer length = 150 integer words = 150 × 4 bytes

If the increment/decrement value is a positive number a circular post-increment is performed. If it is a negative number a circular post-decrement is performed.

If the circular pointer increment/decrement is not needed at the access point, then the increment/decrement value can be specified as 0:

```c
Q6_circ_store_update_W(x, p1, 0, 150, 8)
```

The buffer length can be a variable, but must have a known upper bound. In this case the K1 value is based on the upper bound of the buffer length.

For example, `Q6_circ_store_update_W(x, p1, 1, y, 8)` specifies that the buffer length variable `y` can specify a maximum length of 255 elements (since the K1 value is 8).

**NOTE** The K1 value field is redundant if the buffer length is constant, but still must be assigned a valid K1 value.

Substituting “i” for the parameter “x” makes the preceding example identical to part of the program example in Figure 5-3.
5.5.8.3 Circular addressing with variable increment

The Hexagon processor supports circular addressing with larger, variable increment values. To access this address mode the following intrinsics have been defined.

\[
\text{Q6\_circ\_load\_updateI\_XX( <destination variable>,} \\
\text{<buffer location pointer>,} \\
\text{<increment/decrement variable>,} \\
\text{<buffer length>,} \\
\text{<K1 value> );}
\]

\[
\text{Q6\_circ\_store\_updateI\_XX( <source expression>,} \\
\text{<buffer location pointer>,} \\
\text{<increment/decrement variable>,} \\
\text{<buffer length>,} \\
\text{<K1 value> );}
\]

These intrinsics are identical to the corresponding constant-increment circular addressing intrinsics (Section 5.5.8.1, Section 5.5.8.2) except for two differences:

■ The intrinsic names include the letter I to distinguish them from the other circular intrinsics. (I refers to the M-register field where the increment value is stored.)

■ The increment/decrement value must be specified using an integer variable (not a constant), with one array element considered as unity (in conformance with ANSI C pointer increments). The value must be in the range -1024 ... 1023, and must have an absolute value less than the buffer length.

NOTE This increment/decrement range is significantly larger than the range for the constant-increment circular load and store intrinsics.

The circular intrinsics define increment/decrement values differently from the assembly-level circular instructions. For more information see the PRM.
5.5.8.4 Data type specification

The Hexagon processor can perform memory accesses with different data types; see the *Hexagon V4 Programmer’s Reference Manual* for details.

The data type (size, signed/unsigned, and byte positions) is an important aspect in determining the correct assembly instruction for circular access. Therefore a data type suffix is appended to the name of the circular load/store intrinsics:

- The suffix B, H, UB, UBH, UH, W, or D is appended to the name of the circular load intrinsic.
- The suffix B, HL, HH, W, or D is appended to the name of the circular store intrinsic.

Table 5-4 and Table 5-5 list the names of the specific circular load/store intrinsics.

Table 5-4 Circular load intrinsics

<table>
<thead>
<tr>
<th>Intrinsic name</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6_circ_load_update_UB</td>
<td>8-bit, with zero extension to 32 bits</td>
</tr>
<tr>
<td>Q6_circ_load_update_B</td>
<td>8-bit, with sign extension to 32 bits</td>
</tr>
<tr>
<td>Q6_circ_load_update_UH</td>
<td>16-bit, with zero extension to 32 bits</td>
</tr>
<tr>
<td>Q6_circ_load_update_H</td>
<td>16-bit, with sign extension to 32 bits</td>
</tr>
<tr>
<td>Q6_circ_load_update_UBH</td>
<td>Currently unused</td>
</tr>
<tr>
<td>Q6_circ_load_update_W</td>
<td>32-bit, all bits loaded</td>
</tr>
<tr>
<td>Q6_circ_load_update_D</td>
<td>64-bit, all bits loaded</td>
</tr>
</tbody>
</table>

**NOTE** Two versions of the UBH data type exist: one returns a 32-bit result, the other a 64-bit result. This would normally present a problem for C; however, because the UBH types specify vector operations and circular/bit-reversed addressing cannot be vectorized, the UBH data type is ignored in this version.

Table 5-5 Circular store intrinsics

<table>
<thead>
<tr>
<th>Intrinsic name</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6_circ_store_update_B</td>
<td>8-bit</td>
</tr>
<tr>
<td>Q6_circ_store_update_HL</td>
<td>16-bit, store lower half (bits 15:0)</td>
</tr>
<tr>
<td>Q6_circ_store_update_HH</td>
<td>16-bit, store upper half (bits 31:16)</td>
</tr>
<tr>
<td>Q6_circ_store_update_W</td>
<td>32-bit</td>
</tr>
<tr>
<td>Q6_circ_store_update_D</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
5.5.8.5  Circular buffer alignment

To support circular addressing, circular buffers must be properly aligned in memory. Alignment is performed with the `aligned` attribute (Section 5.2.27). For example:

```c
int circ_buffer[150] __attribute__((aligned(1024)));
```

A circular buffer is properly aligned when its starting byte address is an integral multiple of the closest power-of-2 value larger than the buffer size (in bytes). For example, the circular buffer declared above is aligned to 1024 bytes because the buffer size is 600 bytes (150 integer words \(\times 4\) bytes), and 1024 is the next power of 2 above 600.

**NOTE**  The alignment can be expressed in terms of K values as \(2^{K+2}\). For more information see the Hexagon V4 Programmer’s Reference Manual.

The alignment can also be expressed in terms of \(K_1\) values (Section 5.5.8.1):
- \(2^{K_1+3}\) for 8-byte data types (e.g., `long long`)
- \(2^{K_1+2}\) for 4-byte data types (e.g., `integer`)
- \(2^{K_1+1}\) for 2-byte data types (e.g., `short`)
- \(2^{K_1}\) for 1-byte data types (e.g., `char`)

Circular buffers can be set up on portions (or “slices”) of an array as long as the buffers themselves are properly aligned. For example, consider the following array:

```c
int big_array[1600] __attribute__((aligned(1024)));
```

Given the alignment of `big_array`, it is possible to allocate circular buffers of length 150 integer words (600 bytes) at each location in `big_array` that has an alignment equal to 1024 bytes, as shown in Figure 5-5.

In this case, when multiple circular buffer start addresses potentially exist within an array, an arbitrary pointer variable used for circular buffer access will always wrap around to the most recently preceding circular buffer start address, as shown in Figure 5-5. Therefore circular buffers are feasible within an array only under the following conditions:

- The buffer start address is one of the possible buffer start addresses
- The buffer size (in bytes) is less than the buffer alignment value

**NOTE**  Since variables allocated in the stack should have an alignment of 8 bytes or less, in most cases circular buffers need to be declared either as global variables or (if declared locally within a function) as static.
Figure 5-5  Buffer alignment

Array of 1600 words i.e., 6400 bytes

Possible Circular Buffer (600 bytes)
Possible Circular Buffer (600 bytes)
Possible Circular Buffer (600 bytes)
Possible Circular Buffer (600 bytes)

Possible Buffer start address (aligned to 1024 bytes)
Pointer modulo wraps to address A if it was between A and B
1024 bytes

Possible Buffer start address (aligned to 1024 bytes)
Pointer modulo wraps to address B if it was between B and C
1024 bytes

Possible Buffer start address (aligned to 1024 bytes)
Pointer modulo wraps to address C if it was between C and D
1024 bytes

Possible Buffer start address (aligned to 1024 bytes)
1024 bytes
5.5.8.6 Special issues

Exceptional cases may arise if ordinary pointer arithmetic is performed on the C pointer variable used for circular accesses. Whenever ordinary pointer arithmetic is performed on the pointer variable, the pointer update no longer occurs in the modulo-wrap around fashion.

In case the pointer arithmetic (e.g., $p = p + 256$ (where $p$ is a pointer to `int`) on a buffer of 600 bytes, as shown in Figure 5-6 below) adds a value that makes the pointer point to a location outside the intended circular buffer, the subsequent circular load/store would access data from a new invalid circular buffer.

![Figure 5-6 Invalid circular buffer access](image)
5.5.9 Bit-reversed addressing

To support Viterbi encoding and fast Fourier transforms (FFT) the Hexagon processor supports bit-reversed addressing. Because it is difficult for compilers to generate efficient code for this address mode, the following intrinsics have been defined to efficiently support bit-reversed addressing:

- Q6_bitrev_load_update_XX
- Q6_bitrev_store_update_XX

These intrinsics perform both the bit-reversed buffer access (i.e., load or store) and the updating of the bit-reversed buffer pointer.

**NOTE** The _xx suffix specifies the type of data accessed by the intrinsic (byte, etc.).

The bit-reversed addressing intrinsics are accessed by including the library header file hexagon_protos.h.

For more information on the underlying bit-reversed address mode see the *Hexagon V4 Programmer’s Reference Manual*.

Figure 5-7 shows an example of how the intrinsics are used.

```c
#include <stdio.h>
#include <hexagon_protos.h>

void bitrev(int input_array[], int output_array[], int element_load[])
{
    int *p0 = input_array;
    int *p1 = output_array;
    int i;

    for (i = 0; i < 256; i++)
    {
        Q6_bitrev_load_update_W(element_load[i], p0, 8);
    }

    for (i = 0; i < 256; i++)
    {
        Q6_bitrev_store_update_W(element_load[i], p1, 8);
    }
}

#define N 256
int g_input_array[N] __attribute__((aligned(1<<16)));
int g_output_array[N] __attribute__((aligned(1<<16)));
//bit-reversed buffers should be global or static
```
```c
int main()
{
    int i;
    int element_load[256];
    for (i = 0; i < N; i++)
    {
        g_input_array[i] = i;
    }

    bitreverse(g_input_array, g_output_array, element_load);

    printf("\n-------------element_load-------------\n");
    for (i = 0; i < 256; i++)
    {
        printf("%d, ", element_load[i]);
        if((i%16)==15) printf("\n");
    }
    printf("\n-------------output_array-------------\n");
    for (i = 0; i < 256; i++)
    {
        printf("%d, ", g_output_array[i]);
        if((i%16)==15)printf("\n");
    }
    printf("\n-------------end-------------\n");
}
```

**Figure 5-7  Bit-reversed addressing program example**

In the first for-loop of function `bitreverse` (shown in Figure 5-7), pointer `p0` is used to load array `element_load` with the bit-reversed contents of array `input_array`.

`input_array` was initialized with the values 0 to 255, so `element_load` contains the bit-reversed values of the sequence 0 to 255, as shown in Figure 5-8 below.

In the second for-loop of `bitreverse`, pointer `p1` is used to store the bit-reversed contents of `element_load` into array `output_array`.

Because the contents of `element_load` are already bit-reversed, `output_array` ends up containing the original sequence 0 to 255, as shown in Figure 5-8 below.
Figure 5-8 shows the program output generated by the program in Figure 5-7.

---------element_load---------

0; 128; 64; 192; 32; 160; 96; 224; 16; 144; 80; 208; 48; 176; 112; 240;
8; 136; 72; 200; 40; 168; 104; 232; 24; 152; 88; 216; 56; 184; 120; 248;
4; 132; 68; 196; 36; 164; 100; 228; 20; 148; 84; 212; 52; 180; 116; 244;
12; 140; 76; 204; 44; 172; 108; 236; 28; 156; 92; 220; 60; 188; 124; 252;
2; 130; 66; 194; 34; 162; 98; 226; 18; 146; 82; 210; 50; 178; 114; 242;
10; 138; 74; 202; 42; 170; 106; 234; 26; 154; 90; 218; 58; 186; 122; 250;
6; 134; 70; 198; 38; 166; 102; 230; 22; 150; 86; 214; 54; 182; 118; 246;
14; 142; 78; 206; 46; 174; 110; 238; 30; 158; 94; 222; 62; 190; 126; 254;
5; 133; 69; 197; 37; 163; 99; 225; 17; 145; 81; 209; 49; 177; 113; 241;
9; 137; 73; 201; 41; 169; 105; 233; 25; 153; 89; 217; 57; 185; 121; 249;
3; 131; 67; 195; 35; 161; 97; 223; 19; 147; 83; 211; 51; 179; 115; 243;
13; 141; 77; 205; 45; 173; 109; 237; 29; 157; 93; 221; 61; 189; 125; 253;
7; 135; 71; 199; 39; 167; 103; 231; 23; 151; 87; 215; 55; 183; 119; 247;
15; 143; 79; 207; 47; 175; 111; 239; 31; 159; 95; 223; 63; 191; 127; 255;

---------output_array---------

0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15;
16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31;
32; 33; 34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47;
48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62; 63;
64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79;
80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95;
96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111;
112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126; 127;
128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139; 140; 141; 142; 143;
144; 145; 146; 147; 148; 149; 150; 151; 152; 153; 154; 155; 156; 157; 158; 159;
160; 161; 162; 163; 164; 165; 166; 167; 168; 169; 170; 171; 172; 173; 174; 175;
176; 177; 178; 179; 180; 181; 182; 183; 184; 185; 186; 187; 188; 189; 190; 191;
192; 193; 194; 195; 196; 197; 198; 199; 200; 201; 202; 203; 204; 205; 206; 207;
208; 209; 210; 211; 212; 213; 214; 215; 216; 217; 218; 219; 220; 221; 222; 223;
224; 225; 226; 227; 228; 229; 230; 231; 232; 233; 234; 235; 236; 237; 238; 239;
240; 241; 242; 243; 244; 245; 246; 247; 248; 249; 250; 251; 252; 253; 254; 255;

---------end---------
5.5.9.1 Bit-reversed load

Load value from bit-reversed buffer and update bit-reversed buffer pointer.

\[ Q6\text{-}bitrev\text{-}load\text{-}update\_XX(\text{destination variable}, \text{buffer location pointer}, \log_2(\text{buffer length})) \]

- The \_XX suffix specifies the data type of the intrinsic (Section 5.5.9.3).
- The destination and buffer location pointer must be L-values.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length must be an integral power of 2, with a maximum length of 64K bytes.

Example

For example, consider the following bit-reversed buffer:

\[ \text{int input_array}[256] \_\_\text{attribute}_\_\_((\text{aligned}(1<<16))) ; \]

Access from the first element (i.e., \text{input_array}[0]) to the 256\textsuperscript{th} element (i.e., \text{input_array}[255]) is to be bit-reversed, with access initially starting from the first element (\text{input_array}[0]).

The pointer initialization statement would appear as follows:

\[ \text{int } \_p = \&\text{input_array}[0] ; \]

This C statement initializes the buffer location pointer (which will be used to perform bit-reversed access on the buffer) to the address of the first element of \text{input_array}, with the least-significant 16 bits of the address value bit-reversed.

**NOTE** Because \text{input_array} is aligned on a 64K byte boundary (Section 5.5.9.4) the least-significant 16 bits of the buffer location pointer’s initial value are all set to zero. Bit-reversing all zeros yields all zeros; therefore the C statement shown does in fact generate the value described in the preceding paragraph.

The bit-reversed load statement would appear as follows:

\[ Q6\text{-}bitrev\text{-}load\text{-}update\_W(x, p, 8) ; \]

This C statement loads the data from an address created by bit-reversing the least-significant 16 bits of the buffer location pointer, and then updates the buffer location pointer by post-incrementing it by the fixed value \( M_t \).

The auto-increment value \( M_t \) is determined from the size (in bytes) of the bit-reversed buffer. In the above example, the load intrinsic post-increments by the following amount:

\[ M_t = 1 \ll (16 - \log_2(256 \times 4)) = 1 \ll (16 - 10) = 1 \ll 6 = 64 \text{ bytes} \]

**NOTE** It is not possible to perform bit-reversed access without changing the buffer location pointer.
**NOTE** Normal loads and stores cannot be performed using the buffer location pointer when the buffer length is large enough that the value of Mt is less than the byte alignment required for a normal load or store. Otherwise the buffer location pointer would be incremented by the value in Mt, which in turn could lead to address values that work for the bit-reversed loads and stores but not for normal loads and stores (which are subject to the alignment restrictions).

For example, a normal load or store on a “long long” data element requires an address alignment of 8 bytes. To ensure that this alignment always exists in a buffer location pointer, the bit-reversed buffer length must not be greater than 1024 long long data elements:

\[ Mt = 1 \ll (16 - \log_2(1024 \times 8)) = 1 \ll (16 - 13) = 1 \ll 3 = 8 \text{ bytes} \]

### 5.5.9.2 Bit-reversed store

Store value to bit-reversed buffer and update bit-reversed buffer pointer.

\[
Q6\_bitrev\_store\_update\_XX( \langle \text{source expression} \rangle, \\
\langle \text{buffer location pointer} \rangle, \\
\langle \log_2(\text{buffer length}) \rangle );
\]

- The “_XX” suffix specifies the data type of the intrinsic (Section 5.5.9.3).
- The source expression specifies the value to be stored.
- The buffer location pointer must be an L-value.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length must be an integral power of 2, with a maximum length of 64K bytes.

**Example**

For example, consider the following bit-reversed buffer:

```c
int output_array[256] __attribute__((aligned(1<<16)));
```

Access from the first element (i.e., `output_array[0]`) to the 256th element (i.e., `output_array[255]`) is to be bit-reversed, with access initially starting from the first element (`output_array[0]`).

The pointer initialization statement would appear as follows:

```c
int *p = &output_array[0];
```

This C statement initializes the buffer location pointer (which will be used to perform bit-reversed access on the buffer) to the address of the first element of `output_array`, with the least-significant 16 bits of the address value bit-reversed.

**NOTE** Because `output_array` is aligned on a 64K byte boundary (Section 5.5.9.4) the least-significant 16 bits of the buffer location pointer’s initial value are all set to zero. Bit-reversing all zeros yields all zeros; therefore the C statement shown does in fact generate the value described in the preceding paragraph.
The bit-reversed store statement would appear as follows:

```c
Q6_bitrev_store_update_W(x,p,8);
```

This C statement stores the data to an address created by bit-reversing the least-significant 16 bits of the buffer location pointer, and then updates the buffer location pointer by post-incrementing it by the fixed value Mt.

The auto-increment value Mt is determined from the size (in bytes) of the bit-reversed buffer. In the above example, the store intrinsic post-increments by the following amount:

```
Mt = 1 << (16 - log2(256 × 4)) = 1 << (16 – 10) = 1 << 6 = 64 bytes
```

**NOTE**  It is not possible to perform bit-reversed access without changing the buffer location pointer.

Normal loads and stores cannot be performed using the buffer location pointer when the buffer length is large enough that the value of Mt is less than the byte alignment required for a normal load or store. For more information see Section 5.5.9.1.

### 5.5.9.3 Data type specification

The Hexagon processor can perform memory accesses of different data types; see the *Hexagon V4 Programmer’s Reference Manual* for details.

The data type (size, signed/unsigned, and byte positions) is an important aspect in determining the correct assembly instruction for bit-reversed access. Therefore a data type suffix is appended to the name of the bit-reversed load/store intrinsics:

- The suffix B, H, UB, UBH, UH, W, or D is appended to the name of the bit-reversed load intrinsic.
- The suffix B, HL, HH, W, or D is appended to the name of the bit-reversed store intrinsic.

Table 5-6 and Table 5-7 list the names of the specific bit-reversed load/store intrinsics.

#### Table 5-6  Bit-reversed load intrinsics

<table>
<thead>
<tr>
<th>Intrinsic name</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6_bitrev_load_update_UB</td>
<td>8-bit, with zero extension to 32 bits</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_B</td>
<td>8-bit, with sign extension to 32 bits</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_UH</td>
<td>16-bit, with zero extension to 32 bits</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_H</td>
<td>16-bit, with sign extension to 32 bits</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_UBH</td>
<td>Currently unused</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_W</td>
<td>32-bit, all bits loaded</td>
</tr>
<tr>
<td>Q6_bitrev_load_update_D</td>
<td>64-bit, all bits loaded</td>
</tr>
</tbody>
</table>
NOTE Two versions of the UBH data type exist: one returns a 32-bit result, the other a 64-bit result. This would normally present a problem for C; however, because the UBH types specify vector operations and circular/bit-reversed addressing cannot be vectorized, the UBH data type is ignored in this version.

### Table 5-7 Bit-reversed store intrinsics

<table>
<thead>
<tr>
<th>Intrinsic name</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6_bitrev_store_update_B</td>
<td>8-bit</td>
</tr>
<tr>
<td>Q6_bitrev_store_update_HL</td>
<td>16-bit, store in lower half (bits 15:0)</td>
</tr>
<tr>
<td>Q6_bitrev_store_update_HH</td>
<td>16-bit, store in upper half (bits 31:16)</td>
</tr>
<tr>
<td>Q6_bitrev_store_update_W</td>
<td>32-bit</td>
</tr>
<tr>
<td>Q6_bitrev_store_update_D</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

5.5.9.4 Bit-reversed buffer alignment

To support bit-reversed addressing bit-reversed buffers must be properly aligned in memory. Alignment is performed with the `aligned` attribute (Section 5.2.27). For example:

```c
int input_array[256] __attribute__((aligned(1024)));
```

A bit-reversed buffer is properly aligned when its starting byte address is aligned to a power of 2 greater than or equal to the buffer size (in bytes). For example, the bit-reversed buffer declared above is aligned to 1024 bytes because the buffer size is 1024 bytes (256 integer words × 4 bytes), and 1024 is an integral power of 2.

The buffer location pointer for a bit-reversed buffer must be initialized so the least-significant 16 bits of the address value are bit-reversed. However, the C language offers no simple way to express a pointer value with its least-significant 16 bits bit-reversed.

**NOTE** To simplify the initialization of the bit-reversed pointer, bit-reversed buffers can be aligned to a 64K byte boundary. This has the advantage of allowing the bit-reversed pointer to be initialized to the base address of the bit-reversed buffer, with no bit-reversing required for the least-significant 16 bits of the pointer value (which are all set to 0 by the 64K alignment).

Since variables allocated in the stack should have an alignment of 8 bytes or less, in most cases bit-reversed buffers need to be declared either as global variables or (if declared locally within a function) as static.
5.5.9.5 Special issues

Exceptional cases may arise if ordinary pointer arithmetic is performed on the C pointer variable used for bit-reversed accesses. If the pointer arithmetic adds a value which makes the pointer point to a location outside the intended bit-reversed buffer, the subsequent bit-reversed load/store will access data from a new invalid buffer. For more information, see the Hexagon V4 Programmer’s Reference Manual.

NOTE A pointer used for bit-reversed addressing is not a valid pointer for any other purpose. Therefore, performing any pointer arithmetic on it outside of the bit-reversed intrinsics (except for adding or subtracting a multiple of Mt) is inherently non-useful and should therefore be avoided.
6 Inline Assembly Language

6.1 Overview

GCC provides support for embedding assembly language instructions directly into C programs (which is known as *inline assembly language*).

The primary reason for using inline assembly language is efficiency: non-critical code is written in C, while time-critical routines can be hand-coded in assembly language. Inline assembly eliminates the overhead associated with calling an external assembly function.

Inline assembly language is also useful in stand-alone code for performing hardware-level operations such as interrupt handling.

This chapter covers the following topics:

- The asm statement
- Operands
- Accessed memory
- Clobbered registers
- Side effects
- Condition codes
- Multiple instructions
- Branch instructions
- Coding conventions
- Assembly size
- Operand constraints
- Example program
6.2 The asm statement

GCC supports inline assembly language using the `asm` statement (Section 5.2.36). For example:

```c
asm ("nop");
```

In this example the processor instruction specified in the string is inserted directly into the generated assembly code at its current position in the C program.

In addition to direct inline assembly language, GCC allows the operands of inline assembly instructions to be specified as C expressions. This makes it possible to write assembly code that is more tightly integrated with the surrounding C code. For example:

```c
asm ("%0=add(%1,#%2)" : "=r" (result) : "r" (myvar), "i" (4));
```

This is equivalent to the C statement `result=myvar+4`, where `result` and `myvar` are C variables declared in the program. The strings preceding each operand specify the operand type, and whether the operand is an input or output.

The `asm` instruction has the following syntax:

```c
asm ( instruction_template 
       [: output_operands ]
       [: input_operands ]
       [: clobber_registers ]
);
```

- The instruction template consists of a character string containing the text of the inline assembly code, with embedded parameters specifying where the operands are to be substituted (e.g., "%0=add(%1,#%2)"). The parameter numbers (\%n) correspond to the declaration order of the operands following the template.
- Each input or output operand consists of an operand constraint string followed by a C expression in parentheses (e.g., "=r" (result)). Typical operand constraint characters are `r` (for register operands), `i` (for integer constant operands), and `m` (for memory operands). See Section 6.12.
- Each clobber register consists of a string containing the name of a register.
- Colons are used to separate the instruction template, output operand list, input operand list, and clobber register list. The lists are optional; however, if a list (e.g., output operands) is omitted while a following list (e.g., input operands) is not, then two consecutive colons must appear to indicate the omitted list.
- Commas are used to separate the operands (or registers) within each list.

**NOTE** If you are writing a header file that should be includable in ISO C programs, use the keyword `__asm__` instead of `asm`. See Section 5.2.1.

Branches from one `asm` statement to another `asm` statement are not supported. The compiler code optimizer is not aware of such branches, and therefore cannot take them into account when deciding how to optimize.
It is also possible to specify input and output operands using symbolic names which can be referenced within the assembly code. These names are specified inside square brackets preceding the constraint string, and can be referenced inside the assembly code using `%[name]` instead of a percentage sign followed by the operand number. Using named operands the above example can be specified as:

```c
asm("%[dest]=add(%[src1],#%[src2])"
: [dest] "=r" (result)
: [src1] "r" (myvar), [src2] "i" (4));
```

**NOTE** Symbolic operand names have no relation to other C identifiers. They can be assigned any name (including those of existing C symbols), but no two operands within the same `asm` statement can use the same symbolic name.

### 6.3 Operands

GCC supports three types of inline assembly language operands:

- Input (read-only)
- Output (write-only)
- Input-output (read and write)

The C expressions for output operands must be a valid C l-values; the compiler can check this. Expressions for input operands do not need to be l-values.

If an output operand’s expression cannot be directly addressed (e.g., because it is a bit field), the operand constraint must allow a register. In this case GCC will use the register as the output of the `asm` statement, and store that register into the output.

Output operands are specified with the operand constraint character `=` (Section 6.12.3).

**NOTE** Because output operands are write-only GCC assumes that any pre-existing values in these operands are dead and need not be preserved.

Input-output (also known as read-write) operands are specified with the operand constraint character `+` (Section 6.12.3). Because they are written as well as read, input-output operands must be listed with the output operands. Input-output operands should be used only when the constraints for the operand (including an operand bit field) allow a register.

An input-output operand can be alternatively specified as two separate operands: one input, and one write-only output. The connection between the two is expressed by constraints which specify that they need to be in the same location when the instruction executes. The two operands can accept the same C expression, or different expressions. In the following example, the `asl` instruction accepts the variable `sum` as an input-output operand, and `val` and `shift` as read-only input operands:

```c
asm("%0+=asl(%2,%3)": "=r"(sum): "0"(sum), "r"(val), "r"(shift));
```

The constraint "0" for operand 1 specifies that it occupies the same location as operand 0.
NOTE Numbers can be used as operand constraints only in input operands, and they must refer to an output operand.

Input operands cannot be modified without also being specified as output operands. If all the output operands of an `asm` statement are specified this way (and are thus unused outside the statement), then the `asm` statement itself must be specified as `volatile` (Section 6.6) to prevent GCC from deleting it as unused code.

Only a number in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `sum` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembly code. Thus the following statement would not work reliably:

```
asm("%0=asl(%2,%3)" : "=r"(sum) : "r"(sum), "r"(val), "r"(shift));
```

Reloading or various code optimizations can cause operands 0 and 1 to be assigned to different registers; GCC knows of no reason not to do so. For example, the compiler might find a copy of the value of `sum` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `sum`’s own address). Since the register for operand 1 is not mentioned in the assembly code, the result will not work, but GCC cannot determine that.

A symbolic operand name (Section 6.2) can be used instead of the operand number to specify a matching constraint. For example:

```
asm("%[accum]+=asl(%2,%3)"
 : [accum] "=r"(sum)
 : "[accum]"(sum), "r"(val), "r"(shift));
```

NOTE The compiler does not check whether inline assembly language operands have data types that are reasonable for the instruction being executed. It does not parse the assembly instruction template, and does not know what it means or even whether it is valid assembly language.

If input and output operands are not specified properly in an `asm` statement the execution behavior will be undefined.
6.4 Accessed memory

Inline assembly language can access the contents of memory. In this case the compiler needs to know which memory is accessed so it does not keep the memory contents cached in registers across the assembly language, or optimize away any memory loads or stores in the assembly language.

If the inline assembly language accesses a specific section of memory, you can specify the memory section as an input or output operand. Memory operands are specified with the operand constraint character \texttt{m} (Section 6.12.3).

For example, the following input operand specification indicates that a ten-byte character string will be accessed through the (arbitrarily-typed) pointer operand \texttt{ptr}:

\begin{verbatim}
"m"{__extension__( { struct { char x[10]; } *p = (void *)ptr ; *p; } ) }
\end{verbatim}

The C expression in this example evaluates to a structure containing a 10-byte character array. Arrays can alternatively be specified by listing each array element as a separate operand (e.g., \texttt{a[0]}, \texttt{a[1]}, ...).

In the following example the memory output operand \texttt{"=m" (x)} is required, because otherwise GCC might optimize out of existence the store to \texttt{x}:

\begin{verbatim}
void set1 (int val)
{
    static int x;
    asm ("memw (%0) = %1"
        : "=m" (x)
        : "r" (val));
}
\end{verbatim}

\textbf{NOTE} The argument to \texttt{=m} must be an lvalue.

If inline assembly language accesses arbitrary memory areas in an unpredictable manner, it is not possible to specify the accessed memory areas as operands. In this case the memory access must be specified by adding the string “\texttt{memory}” to the clobbered register list (Section 6.5).

For an example of using “\texttt{memory}” see Section 6.8.

\textbf{NOTE} If the accessed memory in this case is not specified as an operand, the \texttt{asm} statement should additionally be specified as \texttt{volatile} (Section 6.6).

If accessed memory is not specified properly in an \texttt{asm} statement the execution behavior will be undefined.
6.5 Clobbered registers

Inline assembly language can modify the value of specific registers. In this case the compiler needs to know which registers are changed so it can generate correct code around the inline assembly language.

To specify this information in the \texttt{asm} statement, write a third colon after the input operands, followed by the names of the affected registers (which are specified as strings). For example:

\begin{verbatim}
asm ("r25=add(\%0,#4)"
    : /* no output */
    : "r" (myvar)
    : "r25");
\end{verbatim}

In the above example, the \texttt{add} instruction modifies ("clobbers") register \texttt{r25} so it is specified on the clobber list.

Registers specified as clobbered registers must be distinct from any registers associated with the input or output operands.

\textbf{NOTE} Registers cannot be directly used to return values from inline assembly language. Values can be returned only by passing them to C variables which are specified as output operands (Section 6.3).

Registers \texttt{r29} and \texttt{r30} should not be clobbered. If they need to be modified, the inline assembly language must be written to explicitly save and restore them. For more information see the \textit{Hexagon V4 Programmer's Reference Manual}.

If any Hexagon processor control registers (e.g., \texttt{usr} or \texttt{px}) are altered by the inline assembly instruction, they must be specified in the list of clobbered registers. For example:

\begin{verbatim}
asm ("%0=add(%1,%2):sat"
    : "r" (result)
    : "r" (myvar1), "r" (myvar2)
    : "usr");
\end{verbatim}

\textbf{NOTE} If clobbered registers are not specified properly in an \texttt{asm} statement the execution behavior will be undefined.

For compatibility with GCC the symbol \texttt{cc} is accepted in the clobber list but otherwise has no effect.
6.6 Side effects

If an `asm` statement contains output operands, GCC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren’t used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an `asm` statement from being deleted, moved significantly, or combined, by specifying the keyword `volatile` after the `asm` keyword. For example:

```c
#define READ_LOCKED(from, to) \
    asm volatile("%0 = memw_locked (%1);") \
    : "r" (to) \
    : "r" (&from));
```

If you write an `asm` statement with no outputs, GCC will know the instruction has side-effects and will not delete the instruction or move it outside of loops.

The `volatile` keyword indicates that the instruction has important side-effects. GCC will not delete a volatile `asm` statement if it is reachable. (The instruction can still be deleted if GCC can prove that control-flow will never reach the location of the instruction.) In addition, GCC will not reschedule instructions across a volatile `asm`. For example:

```c
#define MEMORY_FENCE asm volatile("barrier;" : : )
```

Note that even a volatile `asm` statement can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can’t expect a sequence of volatile `asm` statements to remain perfectly consecutive. If you want consecutive output, use a single `asm`. Also, note that GCC will perform some optimizations across a volatile `asm` statement.

**NOTE** An `asm` statement without any operands or clobbers (i.e., an “old-style” `asm`) is treated identically to a volatile `asm` statement.

6.7 Condition codes

It is a natural idea to look for a way to provide access to the condition code left by an `asm` statement. However, there is no way to do this reliably because the output operands might need reloading, which would result in additional following “store” instructions. On most processors, these instructions would alter the condition code before there was time to test it. This problem doesn’t arise for ordinary “test” and “compare” instructions because they don’t have any output operands.

**NOTE** For reasons similar to those described above, it is not possible for an `asm` statement to access the condition code left by previous instructions.
6.8 Multiple instructions

You can put multiple assembly instructions together in a single asm statement, if they are separated by the control characters normally used in assembly language source code. The standard character combination is a newline to break the line, plus a tab character to indent to the assembly instruction field (which is written as \n\t).

The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands’ addresses, so you can read and write the clobbered registers as many times as you like. For example:

```c
asm("r20=asl(%1,#3)\n\t%0=add(r20,#1)"
 : "=r" (result)
 : "r" (val)
 : "=r20");
```

The instruction template in a multi-instruction asm statement can be specified as a series of character strings, with one string per instruction line. For example:

```c
asm volatile(
 "1:\n\t"
 " r0 = memub(%1++#1)\n\t"
 " {\n\t"
 " p0 = cmp.eq(r0,#0)\n\t"
 " memb(%0++#1) = r0\n\t"
 " }\n\t"
 " if !p0 jump 1b\n\t"
 : "+r"(s0), "+r"(s1)
 : "r0", "p0", "memory");
```

In the above example (which implements strcpy) “memory” appears in the clobbered register list because the memory affected by the operation is unknown (Section 6.4).

**NOTE** Unless an output operand has the & constraint modifier, GCC may allocate it in the same register as an unrelated input operand, on the assumption the inputs are consumed before the outputs are produced. This assumption may be false if the assembly code actually consists of more than one instruction. In such a case, use & for each output operand that may not overlap an input. See Section 6.12.3 for details.
6.9 Branch instructions

Branches can be included in an `asm` statement, as long as both the branch instruction and its target label are contained in the same multi-instruction `asm` statement (Section 6.8).

Local labels (‘0:’ ... ‘9:’) should be used as branch targets. For example:

```
asm (%0=#0\n
    p1=cmp.eq(%1,#2)\n
    if p1 jump 0f\n
    %0=#1\n
0:

  : "r" (result)

  : "r" (input)

  : "p1");
```

Be sure to use the `f` and `b` notation when specifying local labels as branch operands. For more information see the Hexagon Binutils document.

**NOTE** Using a named label (‘ttt:’) within an `asm` statement is not recommended. The compiler may perform optimizations – such as loop unrolling or function inlining – which create duplicates of the inline assembly code. If this occurs the linker will generate error messages due to multiple definitions of the label.

6.10 Coding conventions

Usually the most convenient way to use `asm` instructions is to encapsulate them in macros that look like functions. For example:

```
#define vsplath(x)  
    ({ long long __value; int __arg = (int) (x);  
        asm("%0=vsplath(%1)" : "=r" (__value) : "r" (__arg));  
        __value; })
```

The `(int)` type conversion in this macro ensures that the operand to the `vsplath` instruction is always a 32-bit register, even if a 64-bit value is specified as an input.

Another way to ensure that the instruction operates on the correct data type is to use a cast within the `asm` statement. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would generate a warning about using a pointer unless the caller explicitly casts it.
To ensure that an instruction operates on a register pair, specify C operands of type `long long` and use the operand constraint character `r` (Section 6.12.3). For example:

```c
{  
    long long a, b=7, c=5;
    asm("%0=add(%1, %2)\n\t"  
        :="r" (a)  
        :"r" (b),"r" (c));

    printf("The result = %lld\n",a);
}
```

### 6.11 Assembly size

Some targets require that GCC track the size of each instruction used in order to generate correct code. Because the final length of an `asm` is only known by the assembler, GCC must make an estimate as to how big it will be. The estimate is formed by counting the number of statements in the pattern of the `asm` and multiplying that by the length of the longest instruction on that processor. Statements in the `asm` are identified by newline characters and whatever statement separator characters are supported by the assembler; on most processors this is the `;` character.

Normally, GCC’s estimate is sufficient to ensure that correct code is generated, but it is possible to confuse the compiler if you use pseudo instructions or assembler macros that expand into multiple real instructions, or if you use assembler directives that expand to more space in the object file than would be needed for a single instruction. If this happens the assembler will generate an error message stating that a label is unreachable.
6.12 Operand constraints

Operand constraints are specified with the input and output operands of an `asm` statement (Section 6.3). Constraints specify the following properties of an operand:

- Whether an operand can be in a register, and what kind of register
- Whether the operand can be a memory reference, and what kind of address
- Whether the operand can be an immediate constant, and what possible values

Constraints can also require two operands to match.

6.12.1 Simple constraints

The simplest form of constraint is a character string, where each letter in the string specifies one kind of operand that is permitted. The following letters can be specified:

- whitespace
  Whitespace characters are ignored and can be inserted at any position except the first. This enables each alternative for different operands to be visually aligned in the `asm` statement even if they have different numbers of constraints and modifiers.

- m
  A memory operand is allowed, with any kind of address that the processor supports in general. The operand must be an lvalue.

- o
  A memory operand is allowed, but only if the address is offsettable. This means that adding a small integer (actually, the width in bytes of the operand, as determined by the address mode) may be added to the address and the result is also a valid memory address.

  For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the processor); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the processor supports.

  Note that in an output operand which can be matched by another operand, the constraint letter `o` is valid only when accompanied by both `<` (if the target processor has predecrement addressing) and `>` (if the target processor has preincrement addressing).

- V
  A memory operand that is not offsettable. In other words, anything that would fit the `m` constraint but not the `o` constraint.

- <
  A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.
A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

A register operand is allowed provided that it is in a general register.

An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use \texttt{n} rather than \texttt{i}.

An immediate integer operand whose value is not an explicit integer is allowed. This might appear strange; if an instruction allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use \texttt{s} instead of \texttt{i}? Sometimes it allows better code to be generated.

For example, some processors support a fullword instruction where it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a \texttt{moveq} instruction. We arrange for this to happen by defining the letter \texttt{K} to mean “any integer outside the range \(-128\) to \(127\)”, and then specifying \texttt{Ks} in the operand constraints.

An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last. This number is allowed to be more than a single digit. If multiple digits are encountered consecutively, they are interpreted as a single decimal integer. There is scant chance for ambiguity, since to-date it has never been desirable that \texttt{10} be interpreted as matching either operand 1 or operand 0. Should this be desired, one can use multiple alternatives instead.

This is called a \textit{matching constraint} and what it really means is that the assembler has only a single operand that fills two roles which \texttt{asm} distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC processors an add instruction really has only two operands, one of them an input-output operand:

\begin{verbatim}
addl #35,r12
\end{verbatim}

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.
An operand that is a valid memory address is allowed. This is for “load address” and “push address” instructions.

A constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

### 6.12.2 Multiple alternative constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on some processors a logical-or instruction can combine a register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so the alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the `?` and `!` characters:

- `?` Disparage slightly the alternative that the `?` appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each `?` that appears in it.

- `!` Disparage severely the alternative that the `!` appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.
6.12.3 Constraint modifier characters

Here are constraint modifier characters.

- Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

+ Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. = identifies an output; + identifies an operand that is both input and output; all other operands are assumed to be input only.

If you specify = or + in a constraint, you put it in the first character of the constraint string.

& Means (in a particular alternative) that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

& applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires & while others do not.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the mulsi3 instruction of the ARM.

& does not obviate the need to write =.

% Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. GCC can only handle one commutative pair in an asm; if you use more, the compiler may fail.

# Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

* Says that the following character should be ignored when choosing register preferences. * has no effect on the meaning of the constraint as a constraint, and no effect on reloading.
### 6.13 Example program

The following example shows a function declaration which contains an inline assembly language statement.

```c
int inlineExample(int a, int b)
{
    int op1, op2, sum, result;
    op1 = a << 12;
    op2 = b >> 7;

    __asm__ ( "r1=and(%1, %2)\n\t"
              "%0=sub(%1, r1)"
              :="r"(sum)
              :"r"(op1), "r"(op2)
              :":r1" );

    result = sum | b | a;
    return(result);
}
```

When this function is compiled at optimization level `-O2` (Section 3.4.9), GCC generates the following assembly language:

```assembly
{  
    r3 = r1
    r4 = r0
    r2 = asr(r1,#7)
}  

r0 = asl(r0,#12)

r1=and(r0, r2)     // inline asm: r1=and(op1,op2)
r0=sub(r0, r1)     // inline asm: sum=sub(op1,r1)

r0 = or(r0,r3)
{
    r0 = or(r0,r4)
    jump r31
}
```
7 Libraries

7.1 Overview

The Hexagon GCC compiler includes the following libraries:

- C standard library
- C++ standard library
- Dynamic loading library
- Processor-specific libraries
- Intrinsics emulation library

The C and C++ standard libraries are derived from the corresponding libraries developed by Dinkumware.

The dynamic loading library enables programs to load and unload libraries at run time. It is derived from the standard UNIX dynamic loading library (libdl).

The processor-specific libraries support the following features:

- Low-level intrinsics (processor instructions, vector access, circular/bit-reversed addressing)
- An API for accessing timer information in the Hexagon processor simulator

The intrinsics emulation library enables developers to compile a C or C++ program that contains instruction intrinsics into a native Linux or Cygwin executable.
7.2 C/C++ standard libraries

The C and C++ standard libraries are derived from the corresponding libraries developed by Dinkumware.

The Dinkum C and C++ libraries are conforming implementations of the standard C and C++ libraries. For more information see the following documents:

- *Hexagon C Library User Guide*
- *Hexagon C++ Library User Guide*

**NOTE**  The Dinkumware libraries do not support the C99 revisions.

7.3 Dynamic loading library

The dynamic loading library enables programs to load and unload libraries at run time. It is derived from the standard UNIX dynamic loading library (*libdl.a*).

The dynamic loading library supports the following operations:

- Load shared object
- Unload shared object
- Find symbol in shared object
- Find symbol by address
- Return error status

The library is accessed by including the library header file *dlfcn.h*.

Table 7-1 lists the dynamic loading functions.

**Table 7-1  Dynamic loading functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlopen</td>
<td>Section 7.3.1</td>
</tr>
<tr>
<td>dlclose</td>
<td>Section 7.3.2</td>
</tr>
<tr>
<td>dladdr</td>
<td>Section 7.3.3</td>
</tr>
<tr>
<td>dlerror</td>
<td>Section 7.3.4</td>
</tr>
</tbody>
</table>
7.3.1 Load shared object

Load shared object into a process address space.

\[
\text{void* dlopen(const char *path, int mode);}
\]

dlopen maps the specified shared object into the address space, relocates it, and resolves its external references.

It accepts the following parameters:

- **path** – Pathname of shared object file
- **mode** – Specifies the relocation type (lazy or immediate)

*path* specifies the absolute pathname of the shared object.

*mode* specifies the type of relocation performed by dlopen. The parameter values RTLD_LAZY and RTLD_NOW are used to specify lazy or immediate relocation, respectively.

dlopen returns a handle which is used in the dynamic load functions dlclose (Section 7.3.2) and dlsym (Section 7.3.3). If the specified shared object has already been loaded by a previous call to dlopen (and not yet unloaded by a call to dlclose), the returned handle points to the loaded instance of the shared object.

**NOTE** If *path* is set to NULL in dlopen, it returns a handle to the global symbol object. This provides access to all the symbols defined in an ordered set of objects consisting of the original program image and any dependencies loaded during program startup.

7.3.2 Unload shared object

Unload shared object from a process address space.

\[
\text{int dlclose(void *handle);}\]

dlclose unlinks and removes from the process address space the object referred to by the specified handle.

It accepts the following parameters:

- **handle** – Handle to a loaded shared object (Section 7.3.1)

Because dlopen can be called multiple times on a given shared object without its being unloaded, the dynamic loader maintains for each loaded object a reference count which is incremented every time dlopen is called on the object. dlclose decrements this reference count, and does not unload an object until its reference count is zero.
7.3.3  Find symbol in shared object

Search for a symbol in a single shared object.

```c
void * dlsym(void * restrict handle,
             const char * restrict symbol);
```

dlsym searches by name for the specified symbol in the specified shared object.

It accepts the following parameters:

- `handle` – Handle to a loaded shared object (Section 7.3.1)
- `symbol` – Name of the symbol to search for

If the symbol is found, `dlsym` returns the symbol address. If the symbol is not found, `dlsym` returns NULL.

7.3.4  Find symbol by address

Search for a symbol in all loaded objects.

```c
int dladdr(void * restrict addr,
           Dl_info * restrict dli);
```

dladdr searches by address for the specified symbol in all loaded objects.

It accepts the following parameters:

- `addr` – Address of the symbol to search for
- `dli` – Address of data structure used to return symbol info

dladdr searches through all currently-mapped shared objects for a symbol whose address – as mapped in the process address space – is closest to the value specified in the parameter `addr`, without exceeding that value.

A shared object’s symbols are potential search matches only if the specified search address lies between the shared object’s base address and the value of its `_end` symbol.

If no symbol is found which matches the search criteria, `dladdr` returns 0. If a symbol is found, `dladdr` returns a nonzero value and assigns to the data structure referenced by `dli` the following information on the found symbol:

- The symbol name
- The symbol base address
- The shared object file that contains the symbol
The `Dl_info` structure used to return the symbol info is defined as follows:

```c
typedef struct {
  const char *dli_fname;     // File defining the symbol
  void *dli_fbase;           // Base address
  const char *dli_sname;     // Symbol name
  const void *dli_saddr;     // Symbol address
} Dl_info;
```

dli_sname references the NULL-terminated name of the found symbol.

dli_saddr contains the actual symbol address (as defined in the process address space).

dli_fname references the file name of the shared object containing the found symbol.

dli_fbase contains the base address of the containing shared object (as defined in the process address space).

**NOTE**  
dli_fname and dli_fbase are set to 0 if the found symbol is contained in the internally-generated “copy” section that is not associated with any file.

The character strings referenced by dli_fname and dli_sname reside in memory that is private to the dynamic loading library. Therefore, these strings should not be modified by the program using the library.

### 7.3.5 Return error status

Return the error status from the last library operation.

```c
char * dlerror(void);
```

dlerror returns a character string indicating the most recent error that occurred while performing one of the dynamic loading library functions.

dlerror returns NULL if no dynamic linking errors have occurred since the last time it was called. Thus, calling it twice in succession will erase any pending error message.
7.4 Processor-specific libraries

The processor-specific libraries support the following features:

■ Instruction intrinsics
■ Vector access intrinsics
■ Circular/bit-reversed addressing intrinsics
■ Simulator timer API

7.4.1 Instruction intrinsics (hexagon_protos.h)

To support efficient coding of the time-intensive sections of a program (without resorting to inline assembly language), GCC provides intrinsics which are used to directly access Hexagon processor instructions.

The instruction intrinsics are accessed by including the library header file “hexagon_protos.h”.

For more information see Section 5.5.3.

7.4.2 Vector intrinsics (hexagon_types.h)

To support efficient coding of vector operations, GCC provides intrinsics which are used to perform the following tasks:

■ Read or write individual words, half-words, or bytes in 32-bit vectors
■ Read or write individual double-words, words, half-words, or bytes in 64-bit vectors
■ Construct 32-bit vectors from a sequence of words, half-words, or bytes
■ Construct 64-bit vectors from a sequence of double-words, words, half-words, or bytes

The vector intrinsics are accessed by including the header file “hexagon_types.h”.

For more information see Section 5.5.4 through Section 5.5.7.
7.4.3 Circular/bitrev intrinsics (hexagon_circ_brev_intrinsics.h)

The Hexagon processor provides hardware-level support for implementing circular buffer and bit-reversed addressing. Because it is difficult for compilers to generate efficient code for these features, the C language has been extended with the following intrinsics to efficiently support circular addressing:

- Q6_circ_load_update
- Q6_circ_store_update
- Q6_bitrev_load_update
- Q6_bitrev_store_update

These intrinsics perform as an atomic operation both the buffer access (i.e., load or store) and the updating of the buffer pointer.

The circular buffer and bit-reversed addressing intrinsics are accessed by including the library header file “hexagon_circ_brev_intrinsics.h”.

For more information see Section 5.5.8 and Section 5.5.9.

7.4.4 Simulation timer (hexagon_sim_timer.h)

The Hexagon processor simulator supports a timer interface to enable users to collect timing information on the execution of target applications.

The timer interface functions are accessed by including the library header file “hexagon_sim_timer.h”.

For more information see the Hexagon Simulator User Guide.
7.5 **Hexagon intrinsics emulation library**

The intrinsics emulation library emulates Hexagon processor instruction intrinsics (Section 5.5.3) on non-Hexagon processor target platforms. The library (named `libnative`) allows developers to compile a C or C++ program that contains intrinsics into a native Linux or Cygwin executable. This enables code portability, easier debugging, and a more efficient development workflow for Hexagon processor applications.

The library is included in the tools release in subdirectory `qc/libnative`.

7.5.1 **Application build procedure**

The intrinsics emulation library enables Hexagon processor C or C++ applications to be compiled into Cygwin or Linux binaries. Therefore, to use the library a native compiler (such as GCC) must be used to build the applications.

To successfully compile the intrinsics the compiler must include the `libnative` header file and link with the `libnative` library. This is done by passing the following three command line options to the native compiler:

```
-I<tools-directory>/qc/libnative/include
-L<tools-directory>/qc/libnative/lib
-lnative
```

For example, the following command can be used to compile the application file `foo.c` which includes calls to instruction intrinsics:

```
gcc -I<tools-dir>/qc/libnative/include foo.c -L<tools-dir>/qc/libnative/lib -lnative
```

**NOTE** The linker options shown above (`-L` and `-lnative`) must follow any source files specified on the compiler command line.

7.5.2 **Restrictions**

- The intrinsics emulation library is not intended for performance modeling, because the emulation does not accurately represent how the Hexagon processor core actually operates. The library is best used for tasks such as unit testing, or to facilitate porting of code from a PC platform to the Hexagon processor.

- The intrinsics emulation library only supports intrinsics that specify instructions directly supported by the Hexagon processor. The library does not support instructions that the assembler maps to an equivalent assembly instruction; for instance, the comparison instruction \( x \geq 8 \) which is mapped by the assembler to \( x > 7 \).

**NOTE** Most assembler-mapped instructions are provided as a convenience for the compiler or assembly language programmer. Some of them enable the V2 processor version to maintain backwards compatibility with V1.
8 Binary Compatibility

Binary compatibility encompasses several related concepts:

**Application binary interface (ABI)**

The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.

**ABI conformance**

A compiler conforms to an ABI if it generates code that follows all of the specifications enumerated by that ABI. A library conforms to an ABI if it is implemented according to that ABI. An application conforms to an ABI if it is built using tools that conform to that ABI and does not contain source code that specifically changes behavior specified by the ABI.

**Calling conventions**

Calling conventions are a subset of an ABI that specify of how arguments are passed and function results are returned.

**Interoperability**

Different sets of tools are interoperable if they generate files that can be used in the same program. The set of tools includes compilers, assemblers, linkers, libraries, header files, startup files, and debuggers. Binaries produced by different sets of tools are not interoperable unless they implement the same ABI. This applies to different versions of the same tools as well as tools from different vendors.

**Intercallability**

Whether a function in a binary built by one set of tools can call a function in a binary built by a different set of tools is a subset of interoperability.
**Implementation-defined features**

Language standards include lists of implementation-defined features whose behavior can vary from one implementation to another. Some of these features are normally covered by a platform’s ABI and others are not. The features that are not covered by an ABI generally affect how a program behaves, but not intercallability.

**Compatibility**

Conformance to the same ABI and the same behavior of implementation-defined features are both relevant for compatibility.

The application binary interface implemented by a C or C++ compiler affects code generation and runtime support for:

- size and alignment of data types
- layout of structured types
- calling conventions
- register usage conventions
- interfaces for runtime arithmetic support
- object file formats

In addition, the application binary interface implemented by a C++ compiler affects code generation and runtime support for:

- name mangling
- exception handling
- invoking constructors and destructors
- layout, alignment, and padding of classes
- layout and alignment of virtual tables

Some GCC compilation options cause the compiler to generate code that does not conform to the platform’s default ABI. Other options cause different program behavior for implementation-defined features that are not covered by an ABI. These options are provided for consistency with other compilers that do not follow the platform’s default ABI or the usual behavior of implementation-defined features for the platform. Be very careful about using such options.

Most platforms have a well-defined ABI that covers C code, but ABIs that cover C++ functionality are not yet common.
Starting with GCC 3.2, GCC binary conventions for C++ are based on a written, vendor-neutral C++ ABI that was designed to be specific to 64-bit Itanium but also includes generic specifications that apply to any platform. This C++ ABI is also implemented by other compiler vendors on some platforms, notably GNU/Linux and BSD systems. We have tried hard to provide a stable ABI that will be compatible with future GCC releases, but it is possible that we will encounter problems that make this difficult. Such problems could include different interpretations of the C++ ABI by different vendors, bugs in the ABI, or bugs in the implementation of the ABI in different compilers. GCC’s -Wabi switch warns when G++ generates code that is probably not compatible with the C++ ABI.

The C++ library used with a C++ compiler includes the Standard C++ Library, with functionality defined in the C++ Standard, plus language runtime support. The runtime support is included in a C++ ABI, but there is no formal ABI for the Standard C++ Library. Two implementations of that library are interoperable if one follows the *de facto* ABI of the other and if they are both built with the same compiler, or with compilers that conform to the same ABI for C++ compiler and runtime support.

When G++ and another C++ compiler conform to the same C++ ABI, but the implementations of the Standard C++ Library that they normally use do not follow the same ABI for the Standard C++ Library, object files built with those compilers can be used in the same program only if they use the same C++ library. This requires specifying the location of the C++ library header files when invoking the compiler whose usual library is not being used. The location of GCC’s C++ header files depends on how the GCC build was configured, but can be seen by using the G++ -v option. With default configuration options for G++ 3.3 the compile line for a different C++ compiler needs to include

```
-Igcc_install_directory/include/c++/3.3
```

Similarly, compiling code with G++ that must use a C++ library other than the GNU C++ library requires specifying the location of the header files for that other library.

The most straightforward way to link a program to use a particular C++ library is to use a C++ driver that specifies that C++ library by default. The g++ driver, for example, tells the linker where to find GCC’s C++ library (libstdc++) plus the other libraries and startup files it needs, in the proper order.

If a program must use a different C++ library and it’s not possible to do the final link using a C++ driver that uses that library by default, it is necessary to tell g++ the location and name of that library. It might also be necessary to specify different startup files and other runtime support libraries, and to suppress the use of GCC’s support libraries with one or more of the options -nostdlib, -nostartfiles, and -nodefaultlibs.
The GCC project would like to thank its many contributors. Without them the project would not have been nearly as successful as it has been. Any omissions in this list are accidental. Feel free to contact law@redhat.com or gerald@pfeifer.com if you have been left out or some of your contributions are not listed. Please keep this list in alphabetical order.

- Analog Devices helped implement the support for complex data types and iterators.
- John David Anglin for threading-related fixes and improvements to libstdc++-v3, and the HP-UX port.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Abramo and Roberto Bagnara for the SysV68 Motorola 3300 Delta Series port.
- Alasdair Baird for various bug fixes.
- Giovanni Bajo for analyzing lots of complicated C++ problem reports.
- Peter Barada for his work to improve code generation for new ColdFire cores.
- Gerald Baumgartner added the signature extension to the C++ front end.
- Godmar Back for his Java improvements and encouragement.
- Scott Bambrough for help porting the Java compiler.
- Wolfgang Bangerth for processing tons of bug reports.
- Jon Beniston for his Microsoft Windows port of Java.
- Daniel Berlin for better DWARF2 support, faster/better optimizations, improved alias analysis, plus migrating GCC to Bugzilla.
- Geoff Berry for his Java object serialization work and various patches.
- Eric Blake for helping to make GCJ and libgcj conform to the specifications.
- Segher Boessenkool for various fixes.
- Hans-J. Boehm for his garbage collector, IA-64 libffi port, and other Java work.
- Neil Booth for work on cpplib, lang hooks, debug hooks and other miscellaneous clean-ups.
- Eric Botcazou for fixing middle- and backend bugs left and right.
Acknowlegements

- Per Bothner for his direction via the steering committee and various improvements to the infrastructure for supporting new languages. Chill front end implementation. Initial implementations of cpplib, fix-header, config.guess, libio, and past C++ library (libg++) maintainer. Dreaming up, designing and implementing much of GCJ.

- Devon Bowen helped port GCC to the Tahoe.

- Don Bowman for mips-vxworks contributions.

- Dave Brolley for work on cpplib and Chill.

- Robert Brown implemented the support for Encore 32000 systems.

- Christian Bruel for improvements to local store elimination.

- Herman A.J. ten Brugge for various fixes.

- Joerg Brunsmann for Java compiler hacking and help with the GCJ FAQ.

- Joe Buck for his direction via the steering committee.

- Craig Burley for leadership of the Fortran effort.

- Stephan Buys for contributing Doxygen notes for libstdc++.

- Paolo Carlini for libstdc++ work: lots of efficiency improvements to the C++ strings, streambufs and formatted I/O, hard detective work on the frustrating localization issues, and keeping up with the problem reports.

- John Carr for his alias work, SPARC hacking, infrastructure improvements, previous contributions to the steering committee, loop optimizations, etc.

- Stephane Carrez for 68HC11 and 68HC12 ports.

- Steve Chamberlain for support for the Renesas SH and H8 processors and the PicoJava processor, and for GCJ config fixes.

- Glenn Chambers for help with the GCJ FAQ.

- John-Marc Chandonia for various libgcj patches.

- Scott Christley for his Objective-C contributions.

- Eric Christopher for his Java porting help and clean-ups.

- Branko Cibej for more warning contributions.

- The GNU Classpath project for all of their merged runtime code.

- Nick Clifton for arm, mcore, fr30, v850, m32r work, --help, and other random hacking.

- Michael Cook for libstdc++ cleanup patches to reduce warnings.

- R. Kelley Cook for making GCC buildable from a read-only directory as well as other miscellaneous build process and documentation clean-ups.

- Ralf Corsepius for SH testing and minor bugfixing.

- Stan Cox for care and feeding of the x86 port and lots of behind the scenes hacking.

- Alex Crain provided changes for the 3b1.
Ian Dall for major improvements to the NS32k port.

Paul Dale for his work to add uClinux platform support to the m68k backend.

Dario Dariol contributed the four varieties of sample programs that print a copy of their source.

Russell Davidson for fstream and stringstream fixes in libstdc++.

Mo DeJong for GCJ and libgcj bug fixes.

DJ Delorie for the DJGPP port, build and libiberty maintenance, and various bug fixes.

Gabriel Dos Reis for contributions to G++, contributions and maintenance of GCC diagnostics infrastructure, libstdc++-v3, including valarray<>, complex<>, maintaining the numerics library (including that pesky <limits> :-) and keeping up-to-date anything to do with numbers.

Ulrich Drepper for his work on glibc, testing of GCC using glibc, ISO C99 support, CFG dumping support, etc., plus support of the C++ runtime libraries including for all kinds of C interface issues, contributing and maintaining complex<>, sanity checking and disbursement, configuration architecture, libio maintenance, and early math work.

Zdenek Dvorak for a new loop unroller and various fixes.

Richard Earnshaw for his ongoing work with the ARM.

David Edelsohn for his direction via the steering committee, ongoing work with the RS6000/PowerPC port, help cleaning up Haifa loop changes, doing the entire AIX port of libstdc++ with his bare hands, and for ensuring GCC properly keeps working on AIX.

Kevin Ediger for the floating point formatting of num_put::do_put in libstdc++.

Phil Edwards for libstdc++ work including configuration hackery, documentation maintainer, chief breaker of the web pages, the occasional iostream bug fix, and work on shared library symbol versioning.

Paul Eggert for random hacking all over GCC.

Mark Elbrecht for various DJGPP improvements, and for libstdc++ configuration support for locales and fstream-related fixes.

Vadim Egorov for libstdc++ fixes in strings, streambufs, and iostreams.

Christian Ehrhardt for dealing with bug reports.

Ben Elliston for his work to move the Objective-C runtime into its own subdirectory and for his work on autoconf.

Marc Espie for OpenBSD support.

Doug Evans for much of the global optimization framework, arc, m32r, and SPARC work.

Christopher Faylor for his work on the Cygwin port and for caring and feeding the gcc.gnu.org box and saving its users tons of spam.

Fred Fish for BeOS support and Ada fixes.
- Ivan Fontes Garcia for the Portuguese translation of the GCJ FAQ.
- Peter Gerwinski for various bug fixes and the Pascal front end.
- Kaveh Ghazi for his direction via the steering committee, amazing work to make `-W -Wall` useful, and continuously testing GCC on a plethora of platforms.
- John Gilmore for a donation to the FSF earmarked improving GNU Java.
- Judy Goldberg for `c++` contributions.
- Torbjorn Granlund for various fixes and the c-torture testsuite, multiply- and divide-by-constant optimization, improved long long support, improved leaf function register allocation, and his direction via the steering committee.
- Anthony Green for his `-Os` contributions and Java front end work.
- Stu Grossman for gdb hacking, allowing GCJ developers to debug Java code.
- Michael K. Gschwind contributed the port to the PDP-11.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Bruno Haible for improvements in the runtime overhead for EH, new warnings and assorted bug fixes.
- Andrew Haley for his amazing Java compiler and library efforts.
- Chris Hanson assisted in making GCC work on HP-UX for the 9000 series 300.
- Michael Hayes for various thankless work he’s done trying to get the c30/c40 ports functional. Lots of loop and unroll improvements and fixes.
- Dara Hazeghi for wading through myriads of target-specific bug reports.
- Kate Hedstrom for staking the G77 folks with an initial testsuite.
- Richard Henderson for his ongoing SPARC, alpha, ia32, and ia64 work, loop opts, and generally fixing lots of old problems we’ve ignored for years, flow rewrite and lots of further stuff, including reviewing tons of patches.
- Aldy Hernandez for working on the PowerPC port, SIMD support, and various fixes.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Kazu Hirata for caring and feeding the Renesas H8/300 port and various fixes.
- Manfred Hollstein for his ongoing work to keep the m88k alive, lots of testing and bug fixing, particularly of GCC configury code.
- Steve Holmgren for MachTen patches.
- Jan Hubicka for his x86 port improvements.
- Falk Hueffner for working on C and optimization bug reports.
- Bernardo Innocenti for his m68k work, including merging of ColdFire improvements and uClinux support.
Christian Iseli for various bug fixes.
Kamil Iskra for general m68k hacking.
Lee Iverson for random fixes and MIPS testing.
Andreas Jaeger for testing and benchmarking of GCC and various bug fixes.
Jakub Jelinek for his SPARC work and sibling call optimizations as well as lots of bug fixes and test cases, and for improving the Java build system.
Janis Johnson for ia64 testing and fixes, her quality improvement sidetracks, and web page maintenance.
Kean Johnston for SCO OpenServer support and various fixes.
Tim Josling for the sample language treelang based originally on Richard Kenner’s “‘toy’ language”.
Nicolai Josuttis for additional libstdc++ documentation.
Klaus Kaempff for his ongoing work to make alpha-vms a viable target.
David Kashtan of SRI adapted GCC to VMS.
Ryszard Kabatek for many, many libstdc++ bug fixes and optimizations of strings, especially member functions, and for auto_ptr fixes.
Geoffrey Keating for his ongoing work to make the PPC work for GNU/Linux and his automatic regression tester.
Brendan Kehoe for his ongoing work with G++ and for a lot of early work in just about every part of libstdc++.
Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination and delay slot scheduling. Richard Kenner was also the head maintainer of GCC for several years.
Mumit Khan for various contributions to the Cygwin and Mingw32 ports and maintaining binary releases for Microsoft Windows hosts, and for massive libstdc++ porting work to Cygwin/Mingw32.
Robin Kirkham for cpu32 support.
Mark Klein for PA improvements.
Thomas Koenig for various bug fixes.
Bruce Korb for the new and improved fixincludes code.
Benjamin Kosnik for his G++ work and for leading the libstdc++-v3 effort.
Charles LaBrec contributed the support for the Integrated Solutions 68020 system.

Jeff Law for his direction via the steering committee, coordinating the entire egcs project and GCC 2.95, rolling out snapshots and releases, handling merges from GCC2, reviewing tons of patches that might have fallen through the cracks else, and random but extensive hacking.

Marc Lehmann for his direction via the steering committee and helping with analysis and improvements of x86 performance.

Ted Lemon wrote parts of the RTL reader and printer.

Kriang Lerdsuwanakij for C++ improvements including template as template parameter support, and many C++ fixes.

Warren Levy for tremendous work on libgcj (Java Runtime Library) and random work on the Java front end.

Alain Lichnewsky ported GCC to the MIPS CPU.

Oskar Liljeblad for hacking on AWT and his many Java bug reports and patches.

Robert Lipe for OpenServer support, new testsuites, testing, etc.

Weiwen Liu for testing and various bug fixes.

Dave Love for his ongoing work with the Fortran front end and runtime libraries.

Martin von Löwis for internal consistency checking infrastructure, various C++ improvements including namespace support, and tons of assistance with libstdc++/compiler merges.

H.J. Lu for his previous contributions to the steering committee, many x86 bug reports, prototype patches, and keeping the GNU/Linux ports working.

Greg McGary for random fixes and (someday) bounded pointers.

Andrew MacLeod for his ongoing work in building a real EH system, various code generation improvements, work on the global optimizer, etc.

Vladimir Makarov for hacking some ugly i960 problems, PowerPC hacking improvements to compile-time performance, overall knowledge and direction in the area of instruction scheduling, and design and implementation of the automaton based instruction scheduler.

Bob Manson for his behind the scenes work on dejagnu.

Philip Martin for lots of libstdc++ string and vector iterator fixes and improvements, and string clean up and testsuites.

All of the Mauve project contributors, for Java test code.

Bryce McKinlay for numerous GCJ and libgcj fixes and improvements.

Adam Megacz for his work on the Microsoft Windows port of GCJ.

Michael Meissner for LRS framework, ia32, m32r, v850, m88k, MIPS, powerpc, haifa, ECOFF debug support, and other assorted hacking.
■ Jason Merrill for his direction via the steering committee and leading the G++
effort.
■ David Miller for his direction via the steering committee, lots of SPARC work,
  improvements in jump.c and interfacing with the Linux kernel developers.
■ Gary Miller ported GCC to Charles River Data Systems machines.
■ Alfred Minarik for libstdc++ string and ios bug fixes, and turning the entire
  libstdc++ testsuite namespace-compatible.
■ Mark Mitchell for his direction via the steering committee, mountains of C++
  work, load/store hoisting out of loops, alias analysis improvements, ISO C
  restrict support, and serving as release manager for GCC 3.x.
■ Alan Modra for various GNU/Linux bits and testing.
■ Toon Moene for his direction via the steering committee, Fortran maintenance,
  and his ongoing work to make us make Fortran run fast.
■ Jason Molenda for major help in the care and feeding of all the services on the
  gcc.gnu.org (formerly egcs.cygnus.com) machine--mail, web services, ftp
  services, etc etc. Doing all this work on scrap paper and the backs of envelopes
  would have been... difficult.
■ Catherine Moore for fixing various ugly problems we have sent her way,
  including the haifa bug which was killing the Alpha & PowerPC Linux kernels.
■ Mike Moreton for his various Java patches.
■ David Mosberger-Tang for various Alpha improvements, and for the initial IA-64
  port.
■ Stephen Moshier contributed the floating point emulator that assists in cross-
  compilation and permits support for floating point numbers wider than 64 bits and
  for ISO C99 support.
■ Bill Moyer for his behind the scenes work on various issues.
■ Philippe De Muyter for his work on the m68k port.
■ Joseph S. Myers for his work on the PDP-11 port, format checking and ISO C99
  support, and continuous emphasis on (and contributions to) documentation.
■ Nathan Myers for his work on libstdc++-v3: architecture and authorship through
  the first three snapshots, including implementation of locale infrastructure, string,
  shadow C headers, and the initial project documentation (DESIGN, CHECKLIST,
  and so forth). Later, more work on MT-safe string and shadow headers.
■ Felix Natter for documentation on porting libstdc++.
■ Nathanael Nerode for cleaning up the configuration/build process.
■ NeXT, Inc. donated the front end that supports the Objective-C language.
■ Hans-Peter Nilsson for the CRIS and MMIX ports, improvements to the search
  engine setup, various documentation fixes and other small fixes.
■ Geoff Noer for this work on getting cygwin native builds working.
■ Diego Novillo for his SPEC performance tracking web pages and assorted fixes in the middle end and various back ends.

■ David O’Brien for the FreeBSD/alpha, FreeBSD/AMD x86-64, FreeBSD/ARM, FreeBSD/PowerPC, and FreeBSD/SPARC64 ports and related infrastructure improvements.

■ Alexandre Oliva for various build infrastructure improvements, scripts and amazing testing work, including keeping libtool issues sane and happy.

■ Melissa O’Neill for various NeXT fixes.

■ Rainer Orth for random MIPS work, including improvements to GCC’s o32 ABI support, improvements to dejagnu’s MIPS support, Java configuration clean-ups and porting work, etc.

■ Hartmut Penner for work on the s390 port.

■ Paul Petersen wrote the machine description for the Alliant FX/8.

■ Alexandre Petit-Bianco for implementing much of the Java compiler and continued Java maintainership.

■ Matthias Pfaller for major improvements to the NS32k port.

■ Gerald Pfeifer for his direction via the steering committee, pointing out lots of problems we need to solve, maintenance of the web pages, and taking care of documentation maintenance in general.

■ Andrew Pinski for processing bug reports by the dozen.

■ Ovidiu Predescu for his work on the Objective-C front end and runtime libraries.

■ Jerry Quinn for major performance improvements in C++ formatted I/O.

■ Ken Raeburn for various improvements to checker, MIPS ports and various cleanups in the compiler.

■ Rolf W. Rasmussen for hacking on AWT.

■ David Reese of Sun Microsystems contributed to the Solaris on PowerPC port.

■ Volker Reichelt for keeping up with the problem reports.

■ Joern Rennecke for maintaining the sh port, loop, regmove & reload hacking.

■ Loren J. Rittle for improvements to libstdc++-v3 including the FreeBSD port, threading fixes, thread-related configury changes, critical threading documentation, and solutions to really tricky I/O problems, as well as keeping GCC properly working on FreeBSD and continuous testing.

■ Craig Rodrigues for processing tons of bug reports.

■ Gavin Romig-Koch for lots of behind the scenes MIPS work.

■ Ken Rose for fixes to GCC’s delay slot filling code.

■ Paul Rubin wrote most of the preprocessor.

■ Pétur Runólfsson for major performance improvements in C++ formatted I/O and large file support in C++ filebuf.
Chip Salzenberg for libstdc++ patches and improvements to locales, traits, Makefiles, libio, libtool hackery, and “long long” support.

Juha Sarlin for improvements to the H8 code generator.

Greg Satz assisted in making GCC work on HP-UX for the 9000 series 300.

Roger Sayle for improvements to constant folding and GCC’s RTL optimizers as well as for fixing numerous bugs.

Bradley Schatz for his work on the GCJ FAQ.

Peter Schauer wrote the code to allow debugging to work on the Alpha.

William Schelter did most of the work on the Intel 80386 support.

Bernd Schmidt for various code generation improvements and major work in the reload pass as well a serving as release manager for GCC 2.95.3.

Peter Schmid for constant testing of libstdc++ - especially application testing, going above and beyond what was requested for the release criteria - and libstdc++ header file tweaks.

Jason Schroeder for jcf-dump patches.

Andreas Schwab for his work on the m68k port.

Joel Sherrill for his direction via the steering committee, RTEMS contributions and RTEMS testing.

Nathan Sidwell for many C++ fixes/improvements.

Jeffrey Siegal for helping RMS with the original design of GCC, some code which handles the parse tree and RTL data structures, constant folding and help with the original VAX & m68k ports.

Kenny Simpson for prompting libstdc++ fixes due to defect reports from the LWG (thereby keeping GCC in line with updates from the ISO).

Franz Sirl for his ongoing work with making the PPC port stable for GNU/Linux.

Andrey Slepuhin for assorted AIX hacking.

Christopher Smith did the port for Convex machines.

Danny Smith for his major efforts on the Mingw (and Cygwin) ports.

Randy Smith finished the Sun FPA support.

Scott Snyder for queue, iterator, istream, and string fixes and libstdc++ testsuite entries.

Brad Spencer for contributions to the GLIBCPP_FORCE_NEW technique.

Richard Stallman, for writing the original GCC and launching the GNU project.

Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.

Nigel Stephens for various mips16 related fixes/improvements.

Jonathan Stone wrote the machine description for the Pyramid computer.

Graham Stott for various infrastructure improvements.
■ John Stracke for his Java HTTP protocol fixes.
■ Mike Stump for his Elxsi port, G++ contributions over the years and more recently his vxworks contributions
■ Jeff Sturm for Java porting help, bug fixes, and encouragement.
■ Shigeya Suzuki for this fixes for the bsdi platforms.
■ Ian Lance Taylor for his mips16 work, general configury hacking, fixincludes, etc.
■ Holger Teutsch provided the support for the Clipper CPU.
■ Gary Thomas for his ongoing work to make the PPC work for GNU/Linux.
■ Philipp Thomas for random bug fixes throughout the compiler
■ Jason Thorpe for thread support in libstdc++ on NetBSD.
■ Kresten Krab Thorup wrote the run time support for the Objective-C language and the fantastic Java bytecode interpreter.
■ Michael Tiemann for random bug fixes, the first instruction scheduler, initial C++ support, function integration, NS32k, SPARC and M88k machine description work, delay slot scheduling.
■ Andreas Tobler for his work porting libgcj to Darwin.
■ Teemu Torma for thread safe exception handling support.
■ Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the VAX machine description.
■ Tom Tromey for internationalization support and for his many Java contributions and libgcj maintainership.
■ Lassi Tuura for improvements to config.guess to determine HP processor types.
■ Petter Urkedal for libstdc++ CXXFLAGS, math, and algorithms fixes.
■ Brent Verner for work with the libstdc++ cshadow files and their associated configure steps.
■ Todd Vierling for contributions for NetBSD ports.
■ Jonathan Wakely for contributing libstdc++ Doxygen notes and XHTML guidance.
■ Dean Wakerley for converting the install documentation from HTML to texinfo in time for GCC 3.0.
■ Krister Walfridsson for random bug fixes.
■ Stephen M. Webb for time and effort on making libstdc++ shadow files work with the tricky Solaris 8+ headers, and for pushing the build-time header tree.
■ John Wehle for various improvements for the x86 code generator, related infrastructure improvements to help x86 code generation, value range propagation and other work, WE32k port.
■ Ulrich Weigand for work on the s390 port.
■ Zack Weinberg for major work on cpplib and various other bug fixes.
■ Matt Welsh for help with Linux Threads support in GCJ.
■ Urban Widmark for help fixing java.io.
■ Mark Wielaard for new Java library code and his work integrating with Classpath.
■ Dale Wiles helped port GCC to the Tahoe.
■ Bob Wilson from Tensilica, Inc. for the Xtensa port.
■ Jim Wilson for his direction via the steering committee, tackling hard problems in various places that nobody else wanted to work on, strength reduction and other loop optimizations.
■ Carlo Wood for various fixes.
■ Tom Wood for work on the m88k port.
■ Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
■ Kevin Zachmann helped ported GCC to the Tahoe.
■ Gilles Zunino for help porting Java to Irix.

In addition to the above, all of which also contributed time and energy in testing GCC, we would like to thank the following for their contributions to testing:

■ Michael Abd-El-Malek
■ Thomas Arend
■ Bonzo Armstrong
■ Steven Ashe
■ Chris Baldwin
■ David Billinghurst
■ Jim Blandy
■ Stephane Bortzmeyer
■ Horst von Brand
■ Frank Braun
■ Rodney Brown
■ Sidney Cadot
■ Bradford Castalia
■ Ralph Doncaster
■ Richard Emberson
■ Levente Farkas
■ Graham Fawcett
■ Robert A. French
■ Jörgen Freyh
- Mark K. Gardner
- Charles-Antoine Gauthier
- Yung Shing Gene
- David Gilbert
- Simon Gornall
- Fred Gray
- John Griffin
- Patrik Hagglund
- Phil Hargett
- Amancio Hasty
- Bryan W. Headley
- Kevin B. Hendricks
- Joep Jansen
- Christian Joensson
- David Kidd
- Tobias Kuipers
- Anand Krishnaswamy
- Ilewellyn
- Damon Love
- Brad Lucier
- Matthias Klose
- Martin Knoblauch
- Jesse Macnish
- Stefan Morrell
- Anon A. Mous
- Matthias Mueller
- Pekka Nikander
- Jon Olson
- Magnus Persson
- Chris Pollard
- Richard Polton
- David Rees
- Paul Reilly
- Tom Reilly
Torsten Rueger
Danny Sadinoff
Marc Schifer
David Schuler
Vin Shelton
Tim Souder
Adam Sulmicki
George Talbot
Gregory Warnes
David E. Young
And many others

And finally we’d like to thank everyone who uses the compiler, submits bug reports and generally reminds us why we’re doing this work in the first place.
## B License Statements

### B.1 History

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Authors</th>
<th>Publisher</th>
<th>Network Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexagon GNU C/C++ Compiler, A GNU Manual</td>
<td>2010</td>
<td>Richard Stallman, GCC Development Community, QUALCOMM Incorporated</td>
<td>Code Aurora Forum</td>
<td>Source files supplied directly to users</td>
</tr>
</tbody>
</table>
B.2 Funding free software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers--the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, “We will donate ten dollars to the Frobnitz project for each disk sold.” Don’t be satisfied with a vague promise, such as “A portion of the profits are donated,” since it doesn’t give a basis for comparison.

Even a precise fraction “of the profits from this disk” is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is “the proper thing to do” when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright © 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted without royalty; alteration is not permitted.
B.3 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

B.3.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (Independent of having been made by running the Program). Whether that is true depends on what the Program does.

You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)
These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS
B.3.2 How to apply these terms to your new programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and a brief idea of what it does.

Copyright (C) year name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than show w and show c; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

### B.4 GNU FREE DOCUMENTATION LICENSE

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

**PREAMBLE**

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

**APPLICABILITY AND DEFINITIONS**

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS
You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

State on the Title page the name of the publisher of the Modified Version, as the publisher.

Preserve all the copyright notices of the Document.

Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

Include an unaltered copy of this License.

Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.
Do not retile any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

COLLECTIONS OF DOCUMENTS
You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE
The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

**B.4.1 ADDENDUM: How to use this license for your documents**

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) year your name.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

with the Invariant Sections being list their titles, with the Front-Cover Texts being list, and with the Back-Cover Texts being list.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>__builtin___fprintf_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___memcpy_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___memmove_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___memcmp_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___memset_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___printf_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___stpcpy_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___strcat_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___strcpy_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___strncat_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___strncpy_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___vfprintf_chk</td>
<td>199</td>
</tr>
<tr>
<td>__builtin___vprintf_chk</td>
<td>199</td>
</tr>
<tr>
<td>[suffix]:</td>
<td>119</td>
</tr>
<tr>
<td>* [spec name]:</td>
<td>118</td>
</tr>
<tr>
<td>-### 33</td>
<td></td>
</tr>
<tr>
<td>#pragma GCC diagnostic kind option</td>
<td>189</td>
</tr>
<tr>
<td>#pragma GCC optimize (&quot;string&quot;...)</td>
<td>191</td>
</tr>
<tr>
<td>#pragma GCC pop options</td>
<td>191</td>
</tr>
<tr>
<td>#pragma GCC push options</td>
<td>191</td>
</tr>
<tr>
<td>#pragma GCC reset options</td>
<td>191</td>
</tr>
<tr>
<td>#pragma GCC visibility pop</td>
<td>190</td>
</tr>
<tr>
<td>#pragma GCC visibility push(visibility)</td>
<td>190</td>
</tr>
<tr>
<td>#pragma implementation [&quot;objects.h&quot;]</td>
<td>221</td>
</tr>
<tr>
<td>#pragma interface [&quot;subdir/objects.h&quot;]</td>
<td>220</td>
</tr>
<tr>
<td>#pragma message string</td>
<td>190</td>
</tr>
<tr>
<td>#pragma pack()</td>
<td>188</td>
</tr>
<tr>
<td>#pragma pack(n)</td>
<td>188</td>
</tr>
<tr>
<td>#pragma pack(pop)</td>
<td>188</td>
</tr>
<tr>
<td>#pragma pack(push,[n])</td>
<td>188</td>
</tr>
<tr>
<td>#pragma pop_macro(&quot;macro_name&quot;)</td>
<td>191</td>
</tr>
<tr>
<td>#pragma push_macro(&quot;macro_name&quot;)</td>
<td>191</td>
</tr>
<tr>
<td>%:function(args)</td>
<td>124</td>
</tr>
<tr>
<td>.SUFFIX</td>
<td>121</td>
</tr>
<tr>
<td>%(name)</td>
<td>122</td>
</tr>
<tr>
<td>[%name]</td>
<td>122</td>
</tr>
<tr>
<td>{%SiX}</td>
<td>125</td>
</tr>
<tr>
<td>{%SiX}</td>
<td>125</td>
</tr>
<tr>
<td>{%SiX;}Ty; ... ;D}</td>
<td>125</td>
</tr>
<tr>
<td>{%SiX}</td>
<td>124</td>
</tr>
<tr>
<td>{%S}</td>
<td>124</td>
</tr>
<tr>
<td>{%S*}</td>
<td>125</td>
</tr>
<tr>
<td>{%S*}</td>
<td>124</td>
</tr>
<tr>
<td>{%S*&amp;)</td>
<td>124</td>
</tr>
<tr>
<td>{%S</td>
<td>P}</td>
</tr>
<tr>
<td>%*</td>
<td>123</td>
</tr>
<tr>
<td>%</td>
<td>120</td>
</tr>
<tr>
<td>%S</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>suffix</td>
</tr>
<tr>
<td>%l</td>
<td>123</td>
</tr>
<tr>
<td>%s</td>
<td>123</td>
</tr>
<tr>
<td>%t</td>
<td>123</td>
</tr>
<tr>
<td>%u</td>
<td>123</td>
</tr>
<tr>
<td>%w</td>
<td>121</td>
</tr>
<tr>
<td>%X</td>
<td>122</td>
</tr>
<tr>
<td>%Y</td>
<td>122</td>
</tr>
<tr>
<td>%Z</td>
<td>122</td>
</tr>
<tr>
<td>%c</td>
<td>272</td>
</tr>
<tr>
<td>%d</td>
<td>273</td>
</tr>
<tr>
<td>%j</td>
<td>273</td>
</tr>
<tr>
<td>%l</td>
<td>121</td>
</tr>
<tr>
<td>%m</td>
<td>123</td>
</tr>
<tr>
<td>%n</td>
<td>121</td>
</tr>
<tr>
<td>%o</td>
<td>121</td>
</tr>
<tr>
<td>%p</td>
<td>121</td>
</tr>
<tr>
<td>%s</td>
<td>122</td>
</tr>
<tr>
<td>%t</td>
<td>123</td>
</tr>
<tr>
<td>%u</td>
<td>121</td>
</tr>
<tr>
<td>%w</td>
<td>121</td>
</tr>
</tbody>
</table>
-fus...
This function, if given a string literal, is evaluated early enough that it is considered a compile-time constant.

```c
float __builtin_nanf(const char *str)
```

- `* file` 32
- `-o file` 98
- `-o0` 69
- `-o1` 68
- `-o2` 69
- `-o3` 69
- `OBJC_INCLUDE_PATH` 126
- `OBJC_INCLUDE_PATH` 128
- `OBJC_INCLUDE_PATH` 129
- `OBJC_INCLUDE_PATH` 130
- `OBJECT_FILE_NAME` 102
- `OPTIMIZE` 171
- `-Os` 69
- `-p` 66
- `-P` 98
- `-pg` 66
- `-p` 66
- `-P` 98
- `-pack` 177
- `-packed` 181
- `-param` 83
- `-pass-exit-codes` 32
- `-pedantic` 47
- `-pedantic` 98
- `-pedantic-errors` 48
- `-pedantic-errors` 98
- `-pg` 66
- `-pie` 103
- `-pipe` 32
- `-print-file-name=library` 66
- `-print-libgcc-file-name` 67
- `-print-multi-directory` 67
- `-print-multi-lib` 67
- `-print-multi-os-directory` 67
- `-print-prog-name=program` 67
- `-print-search-dirs` 67
- `-pure` 171
- `-q` 67
- `-r` 273
- `-remap` 98
- `-S` 103
- `S` 273
- `-S` 32
- `-save-temps` 67
- `section` 171
- `section` 182
- `sentinel` 172
- `-shared` 103
- `shared` 182
- `-shared-libgcc` 104
- `size_t __builtin_object_size (void * ptr, int type)` 198
- `size_t __builtin_offsetof (typename, offsetof_member_designator)` 198
- `-specs=file` 108
- `-static` 104
- `-static-libgcc` 104
- `-std=standard` 37
- `-std=standard` 98
- `SUNPRO_DEPENDENCIES` 128
- `-symbolic` 104
- `-target-help` 31
- `-target-help` 99
Published by Code Aurora Forum.

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.