MetaWare® C/C++

Programmer's Guide for ARC®

4124-037

MetaWare® C/C++ Programmer's Guide for ARC®

© 1997-2007 ARC® International. All rights reserved.

ARC International

North America
3590 N. First Street, Suite 200
San Jose, CA 95134 USA
Tel. +1-408-437-3400
Fax +1-408-437-3401

Europe
Verulam Point
Station Way
St Albans, AL1 5HE
UK
Tel.+44 (0) 20-8236-2800
Fax +44 (0) 20-8236-2801

www.ARC.com

ARC Confidential and Proprietary Information

Notice

This document, material and/or software contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws, and may be embodied in patents issued or pending. Its receipt or possession does not convey any rights to use, reproduce, disclose its contents, or to manufacture, or sell anything it may describe. Reverse engineering is prohibited, and reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden. ARC and the ARC logotype are trademarks of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the "readme" and/or "release notes" that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked "ARC Confidential & Proprietary" may be provided to Licensee's subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples—source code software, programmer guide, documentation.
- Manuals marked "ARC Confidential" may be provided to subcontractors or manufacturers for use in Licensed Products. Examples—product presentations, masks, non-RTL or non-source format.
- Manuals marked "Publicly Available" may be incorporated into Licensee's documentation with appropriate ARC permission. Examples—presentations and documentation that do not embody confidential or proprietary information.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, California 95134.

Trademark Acknowledgments

ARC-Based, ARC-OS Changer, ARCangel, ARCform, ARChitect, ARCompact, ARCtangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, IPShield, MetaDeveloper, the MQX Embedded logo, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/MQXsim, Precise/RTCS, Precise/RTCSsim, RTCS, SeeCode, TotalCore, Turbo186, Turbo86, V8 µ-RISC, V8 microRISC, and VAutomation are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX and MQX Embedded are registered under ARC International. All other trademarks are the property of their respective owners.

4124-037 August 2007

Contents

List of Examples xi

Chapter 1 — Before You Begin 13

Technical Support 14
About This Book14
Where to Go for More Information 15
Conventions17
Tips 17
Notes 17
Cautions 17
Terminology 17

Chapter 2 — MetaWare C/C++ 19

MetaWare C 20
MetaWare C++20
Support for Embedded C++21
The MetaWare C/C++ Driver21
Standards, Conformance, and Portability21

Chapter 3 — Using the Driver 23

Overview 24
Introducing the MetaWare C/C++ Driver24
Driver Syntax 24
MetaWare C/C++ Driver Command 24
Driver Options 24
Recognized Input File Types 27
Driver Argument Files 28
Executable File Names 28
Using the MetaWare C/C++ Driver28
Getting Help on the Command Line 28
Compiling and Linking in One Step 29
Compiling and Linking in Two Steps 29
Compiling and Linking C++ Modules 30
Compiling and Linking Embedded C++ Code 30
Generating Source and Assembly Listings 31
Preparing Your Program for Debugging 31
Linking Run-Time Libraries32
Linking in the Thread-Safe Library 32
Locating MetaWare C/C++ Run-Time Libraries 33
Including Additional Libraries 33
Customizing the Compilation34
Setting Compiler Controls 34
Setting an Alternate Directory for Temporary Files 35
Using the Preprocessor 35
Using the Driver Configuration File 38

Using Makefiles 38
Using .pro Files 39

Chapter 4 — Using Driver Options 41

Overview 42
Driver Option Reference 42
Target-Processor Driver Options 42
Extension Driver Options 45
Floating-Point Driver Options 51
General Driver Options 52
Format Strings for Output-File Names 84

Chapter 5 — Using Compiler Toggles 85

Overview 86
Using Toggles 86
Alphabetical List of Compiler Toggles 86

Chapter 6 — Using Compiler Pragmas 127

Overview 128
Setting Pragmas 128
Pragma Reference 129
Alphabetical List of Pragmas 129
Customizing Your Program with Pragmas 148
Aligning Functions and Data 149
Controlling Initialization, Startup, and Exit 153
Eliminating Prolog and Epilog Overhead 154
Aliasing External Names 154
Specifying an Alternate Static Data Section 157
Specifying an Alternate Code Section 159
Specifying the Name of the Default BSS Section 160
Specifying an Alternate Data Section 160
Specifying an Alternate Literals Section 161
Accessing a Common Block from Assembly 161
Placing Comments in Object Code 162
Defining Registers 163
Using and Optimizing Extensions in C/C++ 164
Using Extension Condition Codes in C/C++ 166

Chapter 7 — Optimizing Performance 169

Overview 170
Default Optimization Level 170
Debugging Optimized Code 170
Changing the Optimization Level 171
Optimization Levels 171
Applying Manual Optimizations 173
Addressing Global Variables and Objects As Near 173
Aligning Code and Data 174
Branch Prediction and Branch Profiling 174
Compiler-Intrinsic Functions 174
Conditional-Jump Minimization 175
Constants in Code 175
Designating Certain Functions As “Called Infrequently” 176

Eliminating Unreferenced Functions and Variables 176
Expression Simplification: Converting Division to Multiplication 176
High-Level/Intermediate Language Merging 177
Inline Division 177
Inlining C++ Inline Functions 178
Inlining Certain ANSI Standard C Functions 178
Inlining Selected Functions 179
Tail Recursion 179

Chapter 8 — Inlining 181

Overview 182
What Is Inlining? 182
Several Ways to Inline Code 182
The Outboard Inliner 183
Automatically Invoking the Inliner 183
Semi-Automatically Invoking the Inliner 184
Manually Invoking the Inliner for Specific Routines 185
Turning off the Outboard Inliner 186
Outboard Inliner Option Reference 186
The Built-in C++ Inliner 188
Inlining C++ Functions with the Outboard Inliner 189
Inlining and throw Statements 189
Inlining across Modules (Wide Inlining) 189
Unique Names in Wide-inlined Code 189

Chapter 9 — Inline Assembly Code 191

Overview 192
Using GCC-Style Inline Assembly 192
Syntax 192
GCC Extended Asm: Usage Tips and Techniques 194
Real-World Examples 199
Inserting an Inline Assembly Function Using the AT&T Facility 202
_Asm Macro Syntax 203
Sample _Asm Macros 204
Inserting a Single Line of Assembly with _ASM() 208
Limitations of _ASM() 209
Branch Instructions and Function _ASM() 209
Discarding Dead _ASM() Code 209

Chapter 10 — Embedded Development 211

Overview 212
Using a Stack 212
Using the Default Stack 212
Specifying Stack Size 212
Aligning Stack-Based Variables 212
Using a Heap 213
Specifying a Heap 213
Using Small-Data Sections 214
Placing Data in Small-Data Sections 214
Scope of Small-Data Specifications 216
Limitations of Small-Data Specifications 217
How Small-Data Sections Are Accessed 217

Handling Link Problems Due to Size of Small Data 218
Adding Your Own Small-Data Sections 218
Adding Your Own Small-Data Sections: Example 220
Initializing Memory 221
Working with Volatile and Uncached Variables 222
Data-Cache Issues on ARC Processors 222
Cache Incoherency: Example of a Volatile Bit Field 222
Using Library Functions for Volatile Copies 223
Using the Volatile Type Qualifier 223
Using the _Uncached Type Qualifier 224
For More Information 224
Writing Interrupt Handlers 224
Reserving Registers for Fast Interrupt Handling 225
Generating Software Interrupts 225
Macros 226
Providing Space for Extensions to Save State 226
Library Functions 226
Adapting the Libraries to an RTOS 226
Determining and Eliminating Library Function Calls 227
Working with Startup Modules 227
Startup Module crt1.o 227
Startup Module crt1pt.o 228
Startup Module initpic.o 228
Startup Module arc_main.o 229
Startup Module crti.o 229
Providing System Calls for Debugging from a Host 229
Providing Low-Level System Functions 230
File Processing 230
Memory Management 234
Process Management 234
Error Handling 235
Environment Identification 235
Directory Identification 236
Time Calculation 236
Argument Processing 236
Signal Handling 237
Building Thread-Safe Applications 237
Thread Safety and Reentrancy Contrasted 237
Considerations for Developing Multi-Threaded Applications 238
Porting an Interface for RTOS Thread Safety 238
Thread-Local Storage 239
Global State Variable errno and Thread Safety 239
Building a Thread-Safe Interface 239
Avoiding Library Usage of Semaphores and Thread-Local Storage 240
Rebuilding the C Run-Time Libraries 241
Rebuilding the Libraries 241
Providing Support for Statistical Profiling 242
Timer Extension Support 242
Providing Instrumentation Code 242
Using Closely Coupled Memory 244
Using Position-Independent Code and Data 244
Building Position-Independent Code and Data 244
Pictable Processing and Startup Files 245

Pictable Processing and Memory Initialization 245

Chapter 11 — Data Storage 247

Overview 248
Implementation-Defined Behavior 248
Identifiers 248
Characters 248
Integers 249
Floating-Point Numbers 250
Arrays and Pointers 251
Placing Objects in Registers 251
How the Compiler Maps Variables to Storage 252
Thread-Local Storage and the thread Attribute 254
Structures, Unions, Enumerations, and Bit Fields 255
Accessing a Volatile Object 258
Preprocessing Directives 258
Compiler Limits 259
Storage Mapping for Class Objects 260
Minimum Size and Alignment 260
Virtual-Function Tables 260
Direct and Indirect Bases 261

Chapter 12 — Assembly Interface 265

Overview 266
Making Assembly Routines Accessible from MetaWare C/C++ 266
Function-Naming Conventions 266
Naming Conventions in C 266
Naming Conventions in C++ 267
Calling Assembly Routines from C or C++ 267
Calling C or C++ Functions from Assembly Routines 268
Calling Functions Directly 268
Calling Functions Indirectly 269
Sharing Variables across Modules 269
Direct Register Assignment for Variables 270

Chapter 13 — Run-Time Organization 273

Overview 274
Register Usage 274
General-Purpose Registers 274
Auxiliary Registers 275
Stack Conventions 276
Stack Mechanics 276
Interrupt Stack Frames 279
Passing Arguments 283
Return Values 284
Prolog and Epilog Code 284
Prolog Code Examples 284
Epilog Code Examples 286
Prolog and Epilog Variations 288
Dynamic Stack Allocation 290

Chapter 14 — Diagnostics 291

Overview 292
Generating Debugging Information 292
Reducing Debugging Information 292
Tracing Function Calls 292
Execution Profiling 293
- Using Statistical Profiling 293

Chapter 15 — Generating Listings 297

Overview 298
Customizing the Listing with Pragmas 298
Listing Format 298
- Ruler 298
- Level Numbers 299
- Include Files 299
- The .pro File 299

Appendix A — Intrinsic Functions 301

Overview 302
Compiler-Intrinsic-Function Reference 302
- Alphabetical Listing of Intrinsics 302

Calling an Equivalent Library Function 306
Defining Intrinsics for Extensions Using C/C++ 306
Using Intrinsics to Access Auxiliary Registers 307

Appendix B — Configuring the Driver 309

Overview 310
Modifying Configuration Variables 310
- Alphabetical List of User-Modifiable Variables 310

Appendix C — Calling Conventions 313

Overview 314
Using the **__attribute__** Syntax 314
The Calling-Convention Pragma 314
The Calling-Convention Type Qualifier 315
Calling-Convention Attributes 315
- Alphabetical Listing of Attributes 315

Predefined Calling-Convention Names 317
Replacing One Function with Another 318

Appendix D — The Heap Manager 319

Overview 320
Allocated Memory 320
Free Memory 321
Integrity Checking 322
- Levels of Heap Integrity Checking 323
- First-fit versus Best-Fit Strategy 323

Appendix E — Branch Prediction 325

Overview 326
How to Perform Branch Prediction 326
- Instrumenting Your Application 327
- Generating Branch-Profiling Data 327
- Recompiling with Branch-Profiling Data 327

The bpmerge Utility 327
- Invoking bpmerge 327
- Determining Endianness and Mode of the Output 328

Branch Prediction and Run-Time Performance 329
The bplist Utility 330
- Invoking bplist 330
- Specifying Format of the Output 331

Appendix F — Diagnostic Messages 333

Overview 334
File I/O Errors 334
System Errors 334
User Error and Warning Messages 335
- Lexical Error Messages 336
- Syntactic Error Messages 336
- Constraint Error and Warning Messages 336

Changing the Message Format 337
- Predefined Message Formats 337
- Changing the Error Message Format 338
- Changing the Warning Message Format 338

Controlling Warning Messages 338
- Controlling Groups of Warnings by Level 338
- Controlling Individual Warnings by Number 339
- Common Superfluous Warnings 340

Checking the Compiler Return Code 340

Appendix G — Quick Options List 343

Appendix H — Quick Toggles List 351

Appendix I — Quick Pragmas List 357

Appendix J — Quick Inliner Options List 361

Index 363

List of Examples

Example 1	Specifying ARCtangent-A4 with the Default Core Version	25
Example 2	Specifying ARCtangent-A4 with an Alternate Core Version	25
Example 3	Specifying ARCtangent-A5 with the Default Core Version	25
Example 4	Specifying ARCtangent-A5 with an Alternate Core Version	25
Example 5	Specifying ARC 600	25
Example 6	Specifying ARC 700 with the Default Core Version	25
Example 7	Specifying ARC 700 with an Alternate Core Version	25
Example 8	Linking Appropriate Libraries (ARCtangent-A5 and Later)	32
Example 9	Accessing Preprocessor Macros from C/C++	45
Example 10	Accessing Preprocessor Macros from Assembly	45
Example 11	Program for Determining Default Character Representation Setting	90
Example 12	Aligning a Named Block for Storage	149
Example 13	Specifying Your Own BSS Section	158
Example 14	Accessing a Common Block from Assembly	161
Example 15	Using a Pragma to Find Field Offsets	162
Example 16	Defining an Auxiliary Register to the Compiler	164
Example 17	Using Extensions to Generate and Use Assembly Instructions	164
Example 18	Generating an Instruction with Blocking and Latency	165
Example 19	Setting and Testing Extension Condition Codes	166
Example 20	Basic Example of How to Use GCC-Style Inline Assembly	194
Example 21	Basic Example Repeated Using Named Operands	194
Example 22	GCC __asm__ Insertion in a Macro	195
Example 23	Using volatile to Prevent Deletion of a GCC __asm__ Insertion	195
Example 24	Multiple Instructions in a Single GCC Inline Assembly Insertion	196
Example 25	Using GCC __asm__ with Instructions That Kill Specific Registers	197
Example 26	Hypothetical Instruction without Numeric Constraints	198
Example 27	Hypothetical Instruction with Numeric Constraints	198
Example 28	Using GCC __asm__ to Force an Operand into a Specific Register	198
Example 29	Using GCC __asm__ with Register Variables	198
Example 30	Using GCC __asm__ to Rename a Global Variable	199
Example 31	Using GCC __asm__ to Halt the Processor	199
Example 32	Using GCC __asm__ to Load and Initialize Registers	200
Example 33	Using GCC __asm__ to Disable Interrupts	200
Example 34	Using GCC __asm__ in an Atomic Exchange with Memory	200
Example 35	Using GCC __asm__ to Set a Bit in a Shared-Memory System	201
Example 36	Using GCC __asm__ to Copy Memory	201
Example 37	Using _SAVE_ALL_REGS with an _Asm Macro	203
Example 38	_Asm macro with One Argument	204
Example 39	_Asm Macro with Multiple Arguments	206
Example 40	Aligning Data Storage on the Stack	213
Example 41	Creating a New Small-Data Section	219
Example 42	Initializing the Base Register for a Small-Data Section (C)	219
Example 43	Initializing the Base Register for a Small-Data Section (Assembly)	220

Example 44	Danger of Using Volatile Bit Fields	222
Example 45	Using the Volatile Qualifier to Generate the .di Suffix	223
Example 46	A thread-safe Implementation of errno	239
Example 47	Rebuilding the ARCtangent-A5 Little-Endian Extension Libraries	241
Example 48	An Empty Class Generates a Four-Byte Object	260
Example 49	Layout for an Instance of Class D	262
Example 50	Class Layout for Virtual Bases That Themselves Have Virtual Bases	262
Example 51	Calling an Assembly Function from MetaWare C/C++	268
Example 52	Calling a C or C++ Function from Assembly	268
Example 53	Calling a MetaWare C/C++ Function from Assembly Indirectly	269
Example 54	Calling a MetaWare C/C++ Function from Assembly Indirectly	269
Example 55	Sharing Variables across C and Assembly Modules	270
Example 56	Assigning a Global Variable Directly to a Register	271
Example 57	Compiler Output for an Interrupt Handler (ARCtangent-A4)	279
Example 58	Interrupt Handler with a Function Call (ARCtangent-A5 and Later)	282
Example 59	Interrupt Handler without a Function Call (ARCtangent-A5 and Later)	283
Example 60	Using the _prefetch() intrinsic to generate a pre-fetch instruction	305
Example 61	A Hidden Use of Intrinsics to Access Auxiliary Registers	307
Example 62	Importing a Rarely Called Function That Never Returns	315
Example 63	Using _FORCE_INLINE with C++	316
Example 64	Sample Output from the bplist utility	331

Chapter 1 — Before You Begin

In This Chapter

- Technical Support
- About This Book
- Conventions

Technical Support

We encourage customers to visit our Technical Support site for technical information before submitting support requests via telephone. The Technical Support site is richly populated with the following:

ARCSolve — Frequently asked questions

ARCSpeak — Glossary of terms

White papers — Technical documents

Datasheets — ARC product flyers

Known Issues — Known bugs and workarounds.

Training — Future ARC Training sessions

Support — Entry point to sending in a query

Downloads — Access to useful scripts and tools

You can access this site via our corporate website at `http://www.ARC.com`, or you can access the site directly: `http://support.ARC.com`.

Note that you must register before you can submit a support request via our Extranet site.

About This Book

This is a guide to the operation of the MetaWare C/C++ globally optimizing compiler for the ARC family of processors.

NOTE Online editions of this book may be more recent than any printed editions you receive.

This *Programmer's Guide* contains all the system-specific information you need to use the compiler effectively. It contains the following topics:

- Chapter 1 — Before You Begin
- Chapter 2 — MetaWare C/C++
- Chapter 3 — Using the Driver
- Chapter 4 — Using Driver Options
- Chapter 5 — Using Compiler Toggles
- Chapter 6 — Using Compiler Pragmas
- Chapter 7 — Optimizing Performance
- Chapter 8 — Inlining
- Chapter 9 — Inline Assembly Code
- Chapter 10 — Embedded Development
- Chapter 11 — Data Storage
- Chapter 12 — Assembly Interface

- Chapter 13 — Run-Time Organization
- Chapter 14 — Diagnostics

This *Programmer's Guide* is written for experienced software developers who have a working knowledge of the C and C++ languages and the specific processor being targeted.

Windows Hosts Only
If you are using a version of the tools that includes an IDE, you can use the IDE to change the most important settings. Use this *Programmer's Guide* for conceptual information and information about the abilities of the compiler. The IDE also allows you to make settings beyond those selectable in the GUI, by typing in options, toggles, and pragmas. For listings of the supported options, toggles, and pragmas, see the following:

- Driver Option Reference on page 42
- Alphabetical List of Compiler Toggles on page 86
- Alphabetical List of Pragmas on page 129

Where to Go for More Information

The release note or readme file describes any special files and provides information that was not available at the time this *Programmer's Guide* was published.

The *Getting Started* manual provides information to help you get the MetaWare C/C++ toolset up and running quickly.

The *MetaWare C/C++ Language Reference* discusses differences between C and C++, migration from C to MetaWare C++, and MetaWare C/C++ extensions to the C and C++ languages. It also describes the syntax and semantics of the C++ language.

The *MetaWare C Library Reference* provides descriptions and examples of C functions.

The *MetaWare C++ I/O Streams Library Reference* provides information about C++ input and output streams.

The *ELF Assembler User's Guide* describes how to use the assembler.

The *ELF Linker and Utilities User's Guide* describes how to use the linker, archiver, and other ancillary utilities for object modules and libraries conforming to the Executable and Linking Format (ELF).

The *MetaWare Debugger User's Guide* contains information about using this graphical debugger for C and C++ programs.

C and C++ Programming Documents

For understanding the C language, we recommend the following texts:

- Plauger, P. J., and J. Brodie,
  Standard C.
  Microsoft Press, 1989.
- Kernigan, Brian W., and Dennis M. Ritchie,
  The C Programming Language, Second Edition.
  Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1988.
- Harbison, Samuel P., and Guy L. Steele Jr.,
  C: A Reference Manual, Second Edition.
  Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1987.

For a definition of the C++ language, consult the following texts:

- Stroustrup, Bjarne, AT&T Bell Laboratories,
  The Design and Evolution of C++.
  Addison-Wesley Publishing Company, 1994.
- Stroustrup, Bjarne, AT&T Bell Laboratories,
  The C++ Programming Language, Second Edition.
  Addison-Wesley Publishing Company, 1991.
- Stroustrup, Bjarne., and M. A. Ellis, AT&T Bell Laboratories,
  <bAsIs>Annotated C++ Reference Manual.
  Addison-Wesley Publishing Company, 1990.

Processor-Specific Documents

ARCtangent Programmer's Reference.
ARC® International
Verulam Point
Station Way
St Albans, AL1 5HE
U.K.
www.ARC.com

ARC Programmer's Reference.
ARC® International
Verulam Point
Station Way
St Albans, AL1 5HE
U.K.
www.ARC.com

Specifications and ABI Documents

We recommend the following documents.

ANSI C Standard

Programming Language C.
ANSI document X3.159
American National Standards Institute, 1989.

ANSI C++ Standard

Working Paper for Draft Proposed International Standard for Information System — Programming Language C++.
ANSI draft document X3J16/96-0018, 1996.

This document specifies the form and establishes the interpretation of programs expressed in the programming language C++. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C++ language programs on a variety of computing systems.

DWARF

DWARF Debugging Information Format.
Version 1.1.0, October 6, 1992.
UNIX International, Programming Languages SIG

DWARF Debugging Information Format.
Revision: Version 2.0.0., July 27, 1993.
UNIX International, Programming Languages SIG

IEEE
IEEE-approved standard for binary floating-point arithmetic, document IEEE 754, 1985.

UNIX System V ABI
System V Release 4 Application Binary Interface.
Third Edition.
UNIX System Laboratories, 1994 (ISBN 0-13-100439-5).

Conventions

Tips

Tips point out useful information.

TIP If your CD-ROM drive is designated by another drive letter, substitute that drive letter in the command.

Notes

Notes point out important information.

NOTE Non-strict semaphores do not have priority inheritance.

Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION If you modify MQX data types, some tools may not operate properly.

Terminology

install_dir
We use *install_dir* to refer to the top-level directory of your MetaWare C/C++ toolset installation.

**/ and **
UNIX and Linux path names use a forward slash (/) to separate directories, while DOS-style path names use a backslash (\). In this manual, we use the UNIX-style forward slash when specifying path names.

Thread-Safe and Reentrant
A *thread-safe* function is one that can safely be called by multiple threads of a multi-threaded operating system, and a *reentrant* function is one that can safely be entered more than once by the same thread.

Chapter 2 — MetaWare C/C++

In This Chapter

- MetaWare C
- MetaWare C++
- Support for Embedded C++
- The MetaWare C/C++ Driver
- Standards, Conformance, and Portability

MetaWare C

MetaWare C is an ANSI Standard C implementation that provides extensions carefully designed to be consistent with the philosophy of C, including some aspects of the C99 Standard. Incompatibilities have been minimized by introducing a minimum of new keywords and by retaining the original syntax. The compiler flags the extensions when you run it in ANSI mode. In addition, the compiler provides many checking features usually available only in a separate "lint" program.

Important MetaWare C language extensions include the following:

- efficient compiler-intrinsic functions (see Appendix A — Intrinsic Functions)
- nested functions complete with up-level references, as in Pascal
- nested functions passable as parameters to other functions, as in Pascal
- iterators: a powerful construct for traversing data structures
- **long long** data type
- in-line assembly code (see Chapter 8 — Inlining)

For more information about MetaWare C and its extensions, see the *MetaWare C/C++ Language Reference*.

MetaWare C++

MetaWare C/C++ is a full implementation of the C++ language as defined in the *Annotated C++ Reference Manual* by Ellis and Stroustrup, with the addition of some of the new features of the ANSI C++ Standard, such as namespaces, run-time type identification (RTTI), templates, exception handling, new cast syntax (**static_cast**), and **mutable**.

MetaWare C/C++ supports the gradual introduction of C++ features into C programs. You can use almost all of C++ — intermixing it with C — while preserving the semantics of your C code. For example, you can enable support for the **bool** data type using toggle `Bool` to avoid interference with older code that defines **bool** itself.

NOTE As the C++ standard developed, the semantics changed so that variables declared in the initialization portion of a **for** statement are local to the **for** statement. By default, MetaWare C++ uses the older semantics, that is, it treats such loop-counter variables as if they were declared outside the **for** loop unless you turn on toggle `For_dcln_local`.

For example:

```
for (int i = 0; i < MAX; i++) {
   // body of loop; "i" is defined here whether
   // toggle is on or off
}

// after loop; "i" is undefined when toggle is
// on, which is correct C++ semantics.
```

Support for Embedded C++

The compiler supports Embedded C++ (EC++) by providing a driver option (**-Hecpp**) that disables certain C++ language features to minimize the use of code or data space in embedded applications.

Some of the disabled features consume code or data space, but some do not and thus can be applied to embedded applications without impacting size.

The compiler is designed to make minimal demands on a program's size. If you do not use a particular feature, the size of the object code is unaffected.

There is no need to disable a feature to produce a program suitable for the embedded environment. For example, the compiler does not generate run-time type information or exception-handling information unless you explicitly request those features.

Because some of these features do not cost embedded programs any space or time, the compiler also provides toggles to selectively turn off non-EC++ features, based on need.

For more information, refer to Chapter 10 — Embedded Development.

The MetaWare C/C++ Driver

The MetaWare C/C++ compiler is distributed with a driver program that you can use to compile and link your program in one step. You can also use the driver to separately invoke the compiler, assembler, or linker.

Windows Hosts Only

> **NOTE** If you are using a version of the tools that includes an IDE, you can use the IDE to change the most important settings without using the driver or command line directly. For more information, see your IDE documentation.

See Chapter 3 — Using the Driver for more information about the driver.

Standards, Conformance, and Portability

ANSI C

MetaWare C/C++ supports ANSI Standard C (ANSI document X3.159) and several extensions. MetaWare C programs can be compiled with an ANSI option that turns off the extensions and reduces the language to the ANSI C Standard core.

C++

For more information about the ANSI C++ features implemented in MetaWare C++, see MetaWare C++ on page 20 and Support for Embedded C++ on page 21.

IEEE

The MetaWare C/C++ compiler conforms to the IEEE 754 standard (1985) for binary floating-point arithmetic, to the extent necessary to meet the ANSI C/C++ requirement for floating-point arithmetic and logical operations.

Additional Information

See Specifications and ABI Documents in Chapter 1 — Before You Begin for more information.

Chapter 3 — Using the Driver

In This Chapter

- Overview
- Introducing the MetaWare C/C++ Driver
- Using the MetaWare C/C++ Driver
- Linking Run-Time Libraries
- Customizing the Compilation

Overview

To create a load module or an executable program, you link one or more object modules generated by the compiler (and the assembler) with MetaWare C/C++ Run-Time Libraries. The process of compiling and linking a program is sometimes referred to as *building* a program. Any assembly-language files that you include go directly to the assembler. The assembler, like the compiler, produces object modules and sends them to the linker.

Introducing the MetaWare C/C++ Driver

The MetaWare C/C++ compiler is distributed with a driver program that you can use to compile and link your program in one step. You can also use the driver to separately invoke the compiler, assembler, or linker.

Windows Hosts Only

> **NOTE** If you are using a version of the tools that includes an IDE, you can use the IDE to change the most important settings without using the driver or command line directly. For more information, see your IDE documentation.

Using the driver to compile and link your program in one step has several advantages. Based on default settings and any (optional) compiler controls you use, the driver knows exactly which libraries to include in the link. This driver-controlled linking significantly decreases the possibility of generating undefined or multiply defined symbol errors. The driver also handles default linker settings, duplicate compiler/linker options, and linker options.

The following sections describe the driver syntax and its various elements.

Driver Syntax

This is the generic syntax of the MetaWare C/C++ driver command line:

driver_cmd [*driver_opts*] *src_file(s)* [@*arg_file*]

The components of the command line have the following meanings:

driver_cmd	MetaWare C/C++ driver command
driver_opts	Driver options, especially the processor series and core version; see Specifying the Target Processor Series and Core on page 25.
src_file	Source-file name(s)
arg_file	ASCII file with more command-line arguments

The following sections discuss each of these command-line components in more detail.

MetaWare C/C++ Driver Command

The driver command is **mcc**. For ARCtangent-A4, add **-a4**.

Driver Options

You can specify any number of driver options in any order on the command line. Each option applies to all the files on the command line (unless the option does not make sense for a given file). You can

also set options in an argument file — an ASCII file read by the driver at compile time. See Driver Argument Files on page 28 for more information. Additionally, you can set options on the `ARGS=` line in the driver configuration file. See Using the Driver Configuration File on page 38 for more information.

> **TIP** The easiest way to specify options is using the `compile.arg` file that ARChitect places in the `/build` directory; see Driver Argument Files on page 28.

The driver options are grouped into the following categories in this manual:

- Processor options specify for which processors or extensions you are generating code. See Target-Processor Driver Options on page 42.
- Floating-point options generate or customize floating-point code. See Floating-Point Driver Options on page 51.
- General options customize other aspects of your application build. See General Driver Options on page 52.

Specifying the Target Processor Series and Core

The driver command, processor series option, and processor core option together specify for which processor the compiler should emit code and select the correct libraries for the processor series.

You must specify the series option (**-a4**, **-a5**, **-arc600**, or **-arc700**), and also specify a core option or accept the default (for **-arc600**, currently only **-core1** is available).

Example 1 Specifying ARCtangent-A4 with the Default Core Version

`hcarc -a4` *myfile.c*

Example 2 Specifying ARCtangent-A4 with an Alternate Core Version

`hcarc -a4 -core6` *myfile.c*

Example 3 Specifying ARCtangent-A5 with the Default Core Version

`mcc -a5` *myfile.c* `(same as mcc -a5 -core3` *myfile.c*`)`

Example 4 Specifying ARCtangent-A5 with an Alternate Core Version

`mcc -a5 -core1` *myfile.c*

Example 5 Specifying ARC 600

`mcc -arc600` *myfile.c*

Example 6 Specifying ARC 700 with the Default Core Version

`mcc -arc700` *myfile.c*

Example 7 Specifying ARC 700 with an Alternate Core Version

`mcc -arc700 -core2` *myfile.c*

Macro _ARCVER

Each **-core** option sets macro **_ARCVER** to a different value. You can use this value for conditional compilation, for example with **#ifdef** directives.

***Table 1 Effect on* Macro _ARCVER of Processor Series and Core Options**

Option Combination	Sets Macro _ARCVER to
-a4 -core5	5
-a4 -core6	6

Table 1 Effect on **Macro _ARCVER of Processor Series and Core Options**

Option Combination	Sets Macro _ARCVER to
-a4 -core7	7
-a4 -core8	8
-a5 -core1	0x11
-a5 -core2	0x12
-a5 -core3	0x13
-arc600 -core1	0x21
-arc700 -core1	0x31
-arc700 -core2	0x32

Passing Options to the Linker

Any option not recognized by the driver is assumed to be a linker option and is passed to the linker. To explicitly pass options to the linker, use driver option **-Hldopt**. If an option passed to the linker has arguments, you must use option **-Hldopt** to pass it.

Resolving Conflicts in Linker and Compiler Option Names

A limited number of linker and compiler options share the same name. In that case, the driver assumes the option is a compiler option. To force such options to be passed to the linker, use driver option **-Hldopt**. For example, the following command passes option **-V** to the linker rather than the compiler:

ARCtangent-A5 and Later

```
mcc file1.c file2.c -Hldopt=-V
```

Table 2 Conflicting Linker and Compiler Option Names lists conflicting linker and compiler option names.

Table 2 Conflicting Linker and Compiler Option Names

Option	Meaning As a Compiler Option	Meaning As a Linker Option
-C	None, but conflicts	Display a listing of a specified type
-h	Searches mini-help files or lists driver options	Refers to a generated shared library (DLL)
-H	Displays names of files included during compilation	Displays linker command-line syntax help screen
-I	Specifies an alternate directory to search for include files	Writes interpreter path name into the program header
-H	Specifies the memory model	Reads an input map file (AT&T-style command file)
-V	Displays options and toggles being passed to the compiler	Displays linker version number prior to linking

Passing Options to the Assembler

When you specify assembly source files directly to the driver, the driver automatically passes options **-I**, **-D**, and **-g** to the assembler. For other assembler options, you must use driver option **-Hasopt** to pass the options to the assembler.

NOTE If an argument to an option passed to the linker or assembler contains embedded whitespace, you must enclose the entire argument in quotes.

Recognized Input File Types

The MetaWare C/C++ driver can process a variety of input file types, including those containing assembly source code, C or C++ source code, preprocessed source code, and object code. The driver recognizes file names with the extensions shown in Table 3 Default Filename Extensions.

If you specify a file name with no extension on the command line, the driver assumes that it is an object file. For example, if you type **hcarc** `alpha` on the command line, the driver invokes the linker without first invoking the compiler, because it assumes `alpha` is an object file.

If you specify a source file with extension `.C` (UNIX and Linux hosts), `.cpp`, `.ii`, or `.cc`, the driver invokes the C++ compiler. (See option **-Hcppext** for exceptions.)

If the source-file extension is `.c` or `.i`, the driver invokes the C compiler, unless you specify Incremental C++ with option **-Hcpplvl**=*1*. (See "Migrating from C to C++" in the *MetaWare C/C++ Language Reference*.)

Table 3 Default Filename Extensions

Extension	Meaning
`.a`	Archive library
`.asm`	Assembly language file; the driver invokes the assembler to assemble this file
`.c`	C source file
`.C`	C++ source file (UNIX and Linux hosts)
`.cc`	C++ source file
`.cpp`	C++ source file
`.i`	Preprocessed C source file
`.ii`	Preprocessed C++ source file
`.lib`	Library file
`.o`	Object file
`.obj`	Object file
`.s`	Assembly language file; the driver invokes the assembler to assemble this file

Changing Default Extensions

To change the default C and C++ filename extensions, you can edit the following variables:

CEXT — Default filename extension for C source files (usually `.c`)

CPPEXT — Default filename extension for C++ source files (usually `.cpp`, `.cc`, and `.C`)

These variables are defined in the driver configuration file (see Using the Driver Configuration File on page 38).

NOTE The same extensions must not appear in both CEXT and CPPEXT.

You can also use command-line option **-Hcppext** to change the C++ filename extensions.

Preprocessed Files

By convention, the compiler assigns extension `.i` (C) or `.ii` (C++) to source files that have been preprocessed by a preprocessor but have not been compiled. Use option **-P** to generate `.i` and `.ii`

files, as explained in General Driver Options on page 52. See Using the Preprocessor on page 35 for more information.

Driver Argument Files

You can package driver arguments (compiler options and file names) in a driver argument file, an ASCII file that contains a list of arguments. This is sometimes called a response file or an @ ("at") file. In the argument file, line boundaries are interpreted as whitespace.

You specify the driver argument file name, prefixed by @, on the driver command line. The arguments in the driver argument file are interpreted as if they appeared on the command line.

Example Command Line Using Argument Files

For example, the following command compiles `my_prog.c` with the options and arguments in `my_args.lst`:

```
mcc @my_args.lst my_prog.c
```

Argument File Conventions

The following conventions apply to argument files:

- You can list arguments on multiple lines.
- A newline is treated as whitespace.
- An exclamation point (!) in the first column denotes a comment line.
- You can specify more than one argument file on the command line.

Executable File Names

The linker combines object files and library functions into an executable file named `a.out`.

You can specify a different executable name with option **-o**. (For more information, see Driver Option Reference on page 42.)

Using the MetaWare C/C++ Driver

This section contains examples of using the MetaWare C/C++ driver. The examples demonstrate how to perform some common compiler operations, using default driver settings.

For information about techniques available for customizing the compilation process, see Customizing the Compilation on page 34.

Getting Help on the Command Line

The MetaWare C/C++ compiler includes a command-line mini-help feature that displays information about compiler controls (options, pragmas, and toggles). To access the mini-help, enter the following command at the command line:

```
mcc -hhelp_topic
```

These requirements apply to the request for online mini-help:

- **-h** must be lowercase.
- *help_topic* is all or part of a control or function name, or a topic.

For example, to get mini-help on all controls for include-directory specification, enter this command:

```
mcc -hinclude
```

To see a listing of all the MetaWare C/C++ driver command-line options (and what they do), enter this command at the command line:

```
mcc -h
```

Compiling and Linking in One Step

Preprocessing and Compiling

The driver invokes the preprocessor (see Using the Preprocessor on page 35) and, if needed, reads the `.pro` file (see Using .pro Files on page 39). The driver then invokes the MetaWare C/C++ compiler, which converts C and C++ source programs into object files.

Linking

After invoking the compiler, the driver calls the linker if *both* of the following conditions are true:

- The compiler has not detected any compilation errors.
- You have not suppressed linking with a compiler option (for example, option **-c**, **-S**, **-E**, or **-Hnoobject**).

The linker links the object files into an executable file. Any arguments not recognized by the driver as compiler arguments are passed to the linker. Any file name with an unrecognized suffix (extension) is passed directly to the linker. (For a list of recognized filename suffixes, see Table 3 Default Filename Extensions on page 27.)

Sample Compile-and-Link Invocations

To compile and link a C program in file `my_prog.c` with optimization level 5, where **-O5** is the option that sets the optimization level, enter this command:

```
mcc -O5 my_prog.c
```

To compile the C program in file `my_sort.c`, link it, and generate an executable file named `testsort`, enter this command:

```
mcc my_sort.c -o testsort
```

To compile and link the C++ program in file `c_plus.cpp`, enter this command:

```
mcc c_plus.cpp
```

For more information about compiling and linking C++ programs, see Compiling and Linking C++ Modules on page 30.

Compiling and Linking in Two Steps

You can use the driver to compile and link your program in two separate steps, as follows.

Step 1

Invoke the driver to run the compiler alone by specifying option **-c** to suppress linking:

```
mcc -c myprog.cpp
```

When you suppress linking, the compiler creates one or more relocatable object files and places them in your working directory.

Step 2

Invoke the driver to run the linker by specifying the object file:

```
mcc myprog.ext
```

where *ext* is the default object-filename extension for your host (either `obj` or `o` — see Table 3 Default Filename Extensions on page 27), or by specifying no extension at all:

```
mcc myprog
```

Passing Linker Options Directly

If a linker option name is the same as a compiler option name (that is, the names conflict), use compiler option **-Hldopt** to bypass the compiler and send the linker option directly to the linker.

For information about how to link a program's object files independently (not using the driver to invoke the linker), see the *ELF Linker and Utilities User's Guide*.

Compiling and Linking C++ Modules

The MetaWare C/C++ distribution includes both a C compiler and a C++ compiler in one executable. C++ has its own set of standard header files that are distinct from those of C. C++ also has its own run-time library.

The driver checks the file extension of the source file you specify on the command line to determine whether it should be compiled as C or C++ code. See Recognized Input File Types on page 27 for more information.

The `CPLUS` symbol in the configuration file tells the MetaWare C/C++ compiler to compile and link C++ files. The following happens when `CPLUS` is set:

- The C++ header files, such as `iostream.h`, are placed on the compiler's search path.
- The C++ library files are included when a program is linked.

`CPLUS` is set by default when you install the MetaWare C/C++ compiler. If you have unset `CPLUS` for any reason, you can reset it on the command line with option **-Hcplus**. You can also set `CPLUS` by assigning it a non-zero value in the configuration file.

NOTE The `CPLUS` symbol must be set when you compile a C++ program. If `CPLUS` is not set, the compilation or the link might fail unless you specify option **-I***cpp_header_directory* to access the C++ header files (if you need them). To gain access to any of the C++ library functions, you must also specifically include the C++ libraries in the link.

If you are compiling and linking C-only programs, you might want to set `CPLUS` to zero (off) in the configuration file; this can decrease link time, because one less library is searched.

The C++ library files contain identifiers that intrude upon the name space of ANSI Standard C programs. Do not use identifiers with these names in C if you mix C and C++:

```
cerr
cin
clog
cout
```

Predefined Symbols

Preprocessor symbols `__CPLUSPLUS__` and `__cplusplus` are defined when you compile C++ modules. You can use these symbols in your programs to conditionally compile C or C++ code. For a complete list of predefined preprocessor symbols, see Table 4 Predefined Macro Symbols on page 36.

Compiling and Linking Embedded C++ Code

The compiler supports embedded C++ through driver option **-Hecpp**, several toggles, and symbolic constant `__embedded_cplusplus`.

Refer to Chapter 10 — Embedded Development for information on developing embedded applications.

Generating Source and Assembly Listings

Use these command-line options to generate source and assembly listings:

-Hlist	Write a source listing to standard output.
-Hanno	Annotate the assembly file output (from option **-S**) with lines from the original source file.
-S	Generate an assembly-language file.

For more information about these options, see General Driver Options on page 52. See also Chapter 15 — Generating Listings.

Preparing Your Program for Debugging

If you want to use a debugger on your program, specify option **-g** on the driver command line. Option **-g** causes the compiler to include debugging information in compiled object files.

> **NOTE** You can produce smaller object files by using option **-g0** or **-g1**. For usage details, see the listings in Chapter 4 — Using Driver Options.

```
mcc -g my_prog.c
```

> **NOTE** Early variants of the ARCtangent-A5 processor have a restriction that does not allow breakpoint instructions to be placed adjacent to compare-and-branch instructions. This can cause problems when statement stepping with source code, as the debugger may attempt to place breakpoint instructions in the illegal locations. This processor restriction was removed in later versions of the core.
>
> If you are using a version of the processor that has this restriction, compile code for debugging with toggle `Branch_reductions` turned off in addition to **-g**. Turning off this toggle is necessary only when compiling with **-g** for source-level debugging.

DWARF Debugging Information

The compiler generates debugging information in DWARF version 2 format by default. For DWARF version 1, specify option **-dwarf1** in addition to option **-g**.

```
mcc -g -dwarf1 my_prog.c
```

For ARCtangent-A4, add **-a4**.

Considerations for Debugging on Hardware

If debugging on hardware, be sure the size of the stack and heap allocated for your program do not exceed the memory limits of your hardware. For information on setting the stack and heap size, see the following:

- Option **-Hstack**
- Using a Stack on page 212
- Using a Heap on page 213

For information about using the debugger, see the *MetaWare Debugger User's Guide*.

Linking Run-Time Libraries

When you invoke the driver to compile and link your program, several factors determine which libraries the driver links into your program, including the following:

- whether you are linking a C or C++ program
- whether the target code is big-endian or little-endian
- whether you have specified option **-Hexcept** to link in exception-aware versions of the C++ run-time libraries
- whether you have specified option **-Hnothread** to link in non-thread-safe versions of the run-time libraries
- whether you have specified option **-Heos**=*rtos_name* to compile and link for your RTOS
- which processor series or extension options you have specified (see Target-Processor Driver Options on page 42)

  For ARCtangent-A5 and later, driver options automatically determine at the link stage which libraries are linked. Example 8 Linking Appropriate Libraries (ARCtangent-A5 and Later) on page 32 shows a command line that links two object files into an executable called `app.out` using libraries optimized for the following:

- ARC 600 processors (**-arc600**)
- Big-endian orientation (**-HB**)
- Reduced register set (**-rf16**)
- Common extension options (**-Xlib**)

Example 8 Linking Appropriate Libraries (ARCtangent-A5 and Later)

```
mcc file1.o file2.o -o app.out -arc600 -Xlib -rf16 -HB
```

For more information, see the following:

- Linking in the Thread-Safe Library on page 32
- Locating MetaWare C/C++ Run-Time Libraries on page 33
- Including Additional Libraries on page 33
- the Release Note or readme file

Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Linking in the Thread-Safe Library

The MetaWare C/C++ Run-Time Libraries are thread-safe by default. The libraries are named `libct.a` and `libcct.a` for C and C++, respectively (the ‘t’ denotes thread safety). The variants of these libraries are distinguished by their location in subdirectories below *install_dir*`/lib`.

For non-thread-safe libraries, compile using option **-Hnothread**, which links in single-threaded versions of the libraries, called `libc.a` and `libcc.a`. These single-threaded libraries are not provided in the distribution, but you can build them using the makefiles provided with the library source.

For more information, see *MetaWare Development Toolkit Getting Started*.

The thread-safe libraries call routines to obtain mutual exclusion (mutex) semaphores and use thread-local data where appropriate. Thread safety as implemented in these libraries has a generally negligible impact on the performance of compiled code, except in the case of C++ exceptions, where the impact is greater. The MetaWare C Run-Time Library also requires thread-local storage, which you can implement using semaphores without additional RTOS features.

For more information, see the following:

- option **-Hthread** in Driver Option Reference on page 42
- option **-Hnothread** in General Driver Options on page 52
- toggle `Thread` in Alphabetical List of Compiler Toggles on page 86
- Building Thread-Safe Applications on page 237
- Thread-Local Storage and the thread Attribute on page 254

Locating MetaWare C/C++ Run-Time Libraries

To find the installed location of the run-time libraries, invoke the driver with command-line option **-Hbatch** or **-V**.

- Option **-Hbatch** creates a batch file that includes the paths to all the run-time libraries needed and specified.
- Option **-V** displays on your monitor the full path names of the libraries being linked.

Including Additional Libraries

By default, the driver links the appropriate and necessary C++, C, and operating-system libraries and other files to produce your executable file.

To include specific libraries and tell the linker where to find libraries, use these options:

-L*paths*	Search directory *paths* for a library specified with option **-l**.
-l*name*	Include in the link a library whose name contains *name*.
-YP*dir1*,*dir2*,…	Use *dir1*, *dir2*, … as default directories for locating libraries

Additional libraries can also be specified directly on the driver or linker command line, in which case these options are not required.

For information on how the linker determines the library search paths, see the *ELF Linker and Utilities User's Guide*.

NOTE Because the target processor does not support shared-object files and dynamic-link libraries (`.so` and `.dll` files), the linker searches only for statically linked libraries (libstring.a and string.lib).

See Providing System Calls for Debugging from a Host on page 229 for information on linking in the provided hostlink library or your own hostlink library to supply low-level function calls from a host computer when debugging.

Customizing the Compilation

You can run the driver as installed, without modification, to compile and link your applications. You can also use special features and utilities included in the installation to fine-tune your compilation process.

Setting Compiler Controls

Compiler controls consist of options, toggles, and pragmas. They modify the compilation process and let you control many aspects of compilation, from the format of a listing to subtle variations in the kind of code generated.

Compiler controls are listed alphabetically and described fully in the following chapters:

- Chapter 4 — Using Driver Options
- Chapter 5 — Using Compiler Toggles
- Chapter 6 — Using Compiler Pragmas

To get a listing of all the controls being passed to the compiler and linker, use option **-V**.

Driver Options

You can specify any number of options on the command line. Each option applies to all the files being compiled in that particular command-line invocation. You can also set options on the `ARGS=` line in the driver configuration file. See Using the Driver Configuration File on page 38 for more information.

Another place to set options is in an argument file — an ASCII file read by the driver at compile time. See Driver Argument Files on page 28 for more information.

NOTE Options set as parameters to ARGS are invisible on the actual command line, so it is easy to forget them. To display the options you have set, specify option **-V** on the command line.

Toggles and Pragmas

You can set most toggles and pragmas either on the command line or within a source-code module. (Toggles that specify the target processor and floating-point variations are an exception; they must be set on the command line.)

Scope of Controls

You can set some toggles and pragmas more than once in a given module. If a control can vary within a module, each time the control changes, the new value takes effect. The default setting stays in effect until the compiler encounters a user-specified setting.

If a control cannot vary within a module, and it is set more than once, the compiler uses the last value encountered before the information is required.

Precedence of Controls

In general, for those controls that can be specified more than one way, the following precedence rules apply:

- Default values in the compiler can be overruled by values set in the driver configuration file.
- Configuration-file values can be overruled by options.
- Options can be overruled by pragmas and toggles in a `.pro` file. (See Using .pro Files on page 39.)

- Pragmas and toggles in a `.pro` file can be overruled by pragmas and toggles in regular source files.
- Pragmas and toggles in regular source files can be overruled by subsequent pragmas and toggles in source.

Pragmas and toggles set in your program's source files are the final user-specified values encountered. These controls are processed in the order in which they appear in the source code.

Exceptions to Precedence Rules

There are exceptions to the rules of precedence. Some command-line controls cannot be overridden. Also, some options (such as **-g** for debugging) have no corresponding pragmas or toggles.

In some cases, a control requires the compiler to make a decision about the control's setting (value) before processing any source code. For instance, if a control modifies the size of an object, that control must be set before the object is declared.

For controls that fall into this category, the control setting used is always the last setting specified before the object's first declaration in the program being compiled; the compiler ignores any subsequent settings. This characteristic is noted in the description of each such control.

Setting an Alternate Directory for Temporary Files

On Windows hosts, the compiler ordinarily creates temporary files in your current working directory. To designate an alternate directory for temporary files, set the `TMPPREFIX` environment variable.

On UNIX and Linux hosts, the compiler ordinarily creates temporary files in `/tmp` or `/var/tmp`. To designate an alternate directory for temporary files, set `TMPDIR` or `TMPPREFIX`.

NOTE The driver first searches `TMP` (Windows hosts) or `TMPDIR` (UNIX and Linux hosts), then searches `TMPPREFIX`.

You can also set `TMPPREFIX` and `TMPDIR` by adding them to the driver configuration file; for more information, see Modifying Configuration Variables on page 310.

Using the Preprocessor

Before a source file is compiled, it is preprocessed by the macro preprocessor. The preprocessor performs the following operations:

- includes in a compilation the contents of source files specified in **#include** (or **#include_next**) statements.
- performs macro expansion on tokens that appear in source files
- directs the compiler to skip over portions of source code
- inserts **#line** directives in source code

Invoking the Preprocessor

Use option **-P** (see Driver Option Reference on page 42) to preprocess your source file, suppress compilation, and save the preprocessed source code to an intermediate `.i` (C) or `.ii` (C++) file.

Choosing the Internal or External Preprocessor

NOTE This section applies only to versions of the MetaWare C/C++ compiler hosted on systems (such as UNIX) that provide a C preprocessor.

The MetaWare C/C++ compiler has an integrated preprocessor that conforms to the ANSI C Standard.

> **NOTE** The default preprocessor is the integrated preprocessor. You can change the default by editing the variable CPP in the driver configuration file. For more information, see Modifying Configuration Variables on page 310.

When you specify option **-Hpcc**, the preprocessor emulates most of the behavior of the AT&T Portable C Compiler, and can therefore be used to compile what is commonly known as "K&R C". (K&R C is the version of the C language documented in the first edition of *The C Programming Language* by Kernighan and Ritchie, before the ANSI C Standard was defined.) For more information about the compiler's PCC mode, see toggle PCC.

C programs that were originally written for K&R C or the AT&T Portable C Compiler might rely on idiosyncrasies of the PCC preprocessor that the MetaWare C/C++ preprocessor does not duplicate. For this reason, MetaWare C/C++ provides option **-Hpcc**, which invokes an "outboard" (external) preprocessor. This option is supported only on those host systems that provide an outboard preprocessor.

> **NOTE** If you use option **-Hpcc** to invoke the external preprocessor, the compiler reads the `.pro` file after preprocessing the source files. If you use **-Hnocpp** to invoke the internal preprocessor, the compiler reads the `.pro` file before preprocessing takes place. See Using .pro Files on page 39.

Using Predefined Macros

The MetaWare C/C++ compiler provides several predefined macro symbols that you can use with the **#if**, **#ifdef**, or **#ifndef** preprocessor directives to include code written for a specific compilation.

For example, if the `__STDC__` macro symbol is defined, as it is when you compile with option **-Hansi**, the compiler accepts code that conforms to the ANSI C Standard. If the `__HIGHC__` symbol is defined, the compiler supports the MetaWare C/C++ extensions.

Table 4 Predefined Macro Symbols lists the preprocessor macros predefined in MetaWare C/C++.

Table 4 Predefined Macro Symbols

Macro Symbol	Defined When...
_ARC	Using the MetaWare C/C++ compiler for any target made by ARC® International
_ARC_BE	Compiling in big-endian mode
_ARCOMPACT	Compile for the ARCompact instruction set (ARCtangent-A5 and later)
_ARC_LE	Compiling in little-endian mode
_ATT4	Compiling for UNIX VR4 compliance
_BE	Compiling in big-endian mode
`__CPLUSPLUS__` or `__cplusplus`	Compiling C++ modules; C++ only
__DATE__	Always defined; produces the source-file's date of translation as a string of the form *mmm* `dd` *yyyy* by calling ASCII function **localtime()**
__embedded_cplusplus	Compiling embedded C++ code (option **-Hecpp**); embedded C++ only
__EXCEPTIONS__	Your program is aware of C++ exception handling (option **-Hexcept**); C++ only

Table 4 Predefined Macro Symbols (con't)

Macro Symbol	Defined When...
`__FILE__`	Always defined; produces the name of the current source file
__HIGHC__	Using MetaWare C or MetaWare C PCC mode (option **-Hansi** is not specified)
__HIGHC_ANSI__	Compiling only programs that conform to ANSI Standard C (option **-Hansi** is specified)
`_LE`	Compiling in little-endian mode
__LINE__	Always defined; produces the line number of the current source line
_MT	Linking in the multi-threaded MetaWare C/C++ Run-Time Libraries (option **-Hthread**)
__PPROFILE__	Generating profiling information (option **-p** or **-pg**)
_REENTRANT	Linking in the multi-threaded MetaWare C/C++ Run-Time Libraries (option **-Hthread**)
__STDC__	Using standard C or MetaWare C (option **-Hpcc** is not specified)
__TIME__	Always defined; produces the time the source file was translated as a string of the form `hh:mm:ss` by calling ASCII function **localtime()**

Assertions (Preprocessor Predicates)

Assertions are a preprocessor construct invented by AT&T to provide a name that can be referenced in a **#if** statement without affecting the macro name space. This means you could have an assertion with the same name (or predicate) as a macro and not suffer any collisions.

Assertions always begin with **#assert** or **#unassert**. The **#assert** declaration defines a predicate by associating a token sequence with it, as follows:

`#assert` *predicate*`(`*token_sequence*`)`

You can then use the assertion in **#if** tests in your code. To test for an assertion in an **#if** statement, use the following syntax:

`#if` *#predicate*`(`*token_sequence*`)`

Note the repetition of the `#` symbol.

Example

The **#if** statement following this assertion evaluates true:

```
#assert target(mychip)
...
#if #target(mychip)
...
```

Predefined Preprocessor Assertions

The following assertions are predefined:

```
#assert cpu(arc)
#assert machine(arc)
```

Removing Assertions

The **#unassert** directive removes assertions. To remove all assertions against a given predicate, use the following syntax:

`#unassert` *predicate*

To remove a single assertion against a predicate, use this syntax:

`#unassert` *predicate*`(`*token_sequence*`)`

NOTE For information about related options **-A-** and **-A**, see General Driver Options on page 52.

Using the Driver Configuration File

The MetaWare C/C++ driver configuration file, `hcarc.cnf`, is an ASCII file the driver reads to configure the compilation process. The configuration file is written in a driver language and is located in the `/bin` subdirectory of your MetaWare C/C++ installation.

For most installations, there is no need to change the contents of the configuration file. However, it can be customized as required.

See Configuring the Driver on page 309 for more information about the driver and the driver configuration file.

Using Makefiles

To build a MetaWare C/C++ application, it is often more convenient to write and execute a makefile than to place all the information needed for the compilation on the driver command line. A makefile is an ASCII file that specifies which target files (object files) depend on your program's source and header files.

To write a makefile, you must know how your source files depend upon each other during the compiling and linking process. The following MetaWare C/C++ controls can help you determine the file dependencies needed to write makefiles.

Option **-Hdepend**	Use with **-Hmake**, **-Hmakeof**, or **-Humake** to specify a directory where the makefile dependency file should be placed.
Option **-Hmake**	Directs the compiler to write the dependencies needed to create a makefile to standard output
Option **-Hmakeof**	Directs the compiler to write the dependencies needed to create a makefile to a file you specify.
Option **-Humake**	Directs the compiler to generate makefile dependencies, but write them to a file with the filename extension `.u` instead of to standard output
Option **-Hrel**	Directs the compiler not to place absolute include files into the list of dependencies produced by the **-Hmake**, **-Hmakeof**, or **-Humake** option
Toggle `Normalize_file_names`	Directs the compiler to normalize all pathnames and file names associated with any **#include** directive

NOTE You can examine sample makefiles in the directories below *install_dir*/`lib/src/c/archive`.

Using .pro Files

A `.pro` file is an ASCII file that is treated as a prefix to the source file. Its purpose is to customize the compiler to particular needs. The compiler reads the `.pro` file immediately before reading the source files used in compilation. The default `.pro` file name is `hc.pro`, but you can specify an alternate name with option **-Hpro**, as explained in Driver Option Reference on page 42.

You can use a `.pro` file to initialize toggles, specify aliasing conventions, define macros, invoke pragmas, define global types, and so on. These new "defaults" can then be applied to all source modules in a global fashion. In other words, you can use the `.pro` file to alter compiler defaults. For example, given a `.pro` file containing these lines:

```
#pragma On(List)
typedef long int integer;
```

the compiler does the following:

- turns on toggle *`List`* (with pragma `On`) to produce a source listing
- creates a **typedef** named `integer`, synonymous with **long int**

NOTE We recommend that you use the driver configuration file to make configuration changes, rather than the `.pro` file. See Using the Driver Configuration File on page 38.

Searching for the .pro File

When you compile a program, the compiler searches for a `.pro` file following the same paths it uses to search for other include files.

- If the compiler finds a `.pro` file, it reads the `.pro` file before it reads your application's source files.
- If the compiler does not find a `.pro` file, it assumes there is none and immediately starts reading your application's source file(s) — even if you specify option **-Hpro**. This happens because option **-Hpro** specifies the name of a `.pro` file but does not require that the file be found.

Using Conditional Compilation Directives in a .pro File

You can use **#ifdef**/**#else** compiler directives in your `.pro` file to generate different code, depending on whether a symbolic constant is defined on the command line.

For example, if `.pro` file `my.pro` contains conditional code depending on **#ifdef** `DEF`, you can include that code in your program by entering the following on the command line:

```
mcc myprog.c -Hpro=my.pro -DDEF
```

Using a .pro File and the Preprocessor

If you use option **-Hcpp** to invoke the external preprocessor, the compiler reads the `.pro` file after preprocessing the source files. If you use **-Hnocpp** to invoke the internal preprocessor, the compiler reads the `.pro` file before preprocessing takes place.

See Choosing the Internal or External Preprocessor on page 35.

Chapter 4 — Using Driver Options

In This Chapter

- Overview
- Driver Option Reference
- Format Strings for Output-File Names

Overview

Driver options tell the compiler how to compile your program. You specify options directly on the command line or, alternatively, in an argument file designated to contain additional options. Options affect the entire program being compiled.

Windows Hosts Only
If you are using a version of the tools that includes an IDE, you can set options in the IDE without using the driver or command line directly. For more information, see the IDE user's guide.

See Driver Options on page 24 for details about how to specify driver options.

Driver Option Reference

The options documented in this section are grouped into the following categories.

- Target-Processor Driver Options on page 42
- Extension Driver Options on page 45
- Floating-Point Driver Options on page 51
- General Driver Options on page 52

For a quick-access listing of all options, see Quick Options List on page 343.

For general information about the driver command line, see Driver Syntax on page 24.

Target-Processor Driver Options

You must place a processor option in the **hcarc** command line to specify the processor series and core version, or accept the defaults.

NOTE Target-processor options are mutually exclusive.

Defaults

ARCtangent-A4
By default, the ARCtangent-A4 compiler generates code conforming to the base-case instruction set for version 8 of the processor core. The default processor option is **-core8**.

ARCtangent-A5 and Later
By default, the compiler for ARCtangent-A5 and later generates code conforming to the base-case instruction set of the ARC 600 processor core. No core specification is required as ARC 600 has only one core version.

-a4 — Specify the ARCtangent-A4 processor series

Option **-a4** specifies the ARCtangent-A4 processor series.

You can also specify a core version using a **-core*** option (**-core8** is the default).

-a5 — Specify the ARCtangent-A5 processor series

Option **-a5** specifies the ARCtangent-A5 processor series and links libraries compiled for that processor series. You may also specify a **-core*** option to select the target processor core.

-a6 — Specify the ARC 600 processor series

Synonym for option **-arc600**.

-a7 — Specify the ARC 700 processor series

Synonym for option **-arc700**.

-arc600 — Specify the ARC 600 processor series

Option **-arc600** specifies the ARC 600 processor series and links libraries compiled for that processor series. This option includes the barrel-shifter instructions. You may also specify a **-core*** option to select the target processor core. Currently the only **-core*** option for **-arc600** is **-core1**, which is specified by default when you specify **-arc600**.

Code generated using this option is not guaranteed to run on earlier processors. To generate ARC 600 code that runs on ARC 700, add option **-compat_a7**.

-arc700 — Specify the ARC 700 processor series

Option **-arc700** specifies the ARC 700 processor series and links libraries compiled for that processor series. You may also specify a **-core*** option to select the target processor core. Currently the default core option is **-core1**. This option includes the barrel-shifter, **NORM**, **SWAP**, and extended-arithmetic instructions.

Code generated using this option is not guaranteed to run on earlier processors; for ARC 600 compatibility, add option **-compat_a6**.

-compat_a6 — Optimize for ARC 700 with ARC 600 compatibility

Use option **-compat_a6** with **-arc700** to generate code optimized for the ARC 700 processor series that also runs on the ARC 600 processor series. Note that ARC 600 compatibility is possible only if you do *not* use the **-Xmpy** extension, as **-Xmpy** is not supported on the ARC 600 processor series. In addition, your target ARC 600 processor must have the **-Xnorm** and **-Xswap** extensions, as they are assumed by **-arc700**.

-compat_a7 — Optimize for ARC 600 with ARC 700 compatibility

Use option **-compat_a7** with **-arc600** to generate code optimized for the ARC 600 processor series that also runs on the ARC 700 processor series. Note that ARC 700 compatibility is possible only if you do *not* use the **-Xmult32** extension, as **-Xmult32** is not supported on the ARC 700 processor series.

-core1 — Generate code specific to core version 1

Option **-core1** instructs the compiler to generate code for ARCtangent-A5 or ARC 700 core version 1. Note that this option does not enable the barrel-shifter instructions for ARCtangent-A5 processors. If you are using this option to eliminate the ARCtangent-A5 barrel-shifter instructions, you must specify **-a5** as well.

The **_ARCVER** macro setting associated with option **-core1** depends on which processor *series* (such as **-a5**) you specify with option **-core1**. See Effect on Macro _ARCVER of Processor Series and Core Options on page 25 for values.

-core2 — Generate code specific to core version 2

When used with **-a5**, **-core2** instructs the compiler to generate code for ARCtangent-A5 core version 2 and sets macro **_ARCVER** to `0x12`. Note that this option does not enable the barrel-shifter instructions.

When used with **-arc700**, **-core2** instructs the compiler to generate code for ARC 700 core version 2 and sets macro **_ARCVER** to `0x32`.

-core3 — Generate code specific to ARCtangent-A5 core version 3

When used with **-a5**, **-core3** instructs the compiler to generate code for ARCtangent-A5 core version 3 (ARCtangent-A5.2) and sets macro **_ARCVER** to `0x13`.

Option **-core3** assumes the existence of the barrel-shifter instructions, which are included in the base-case ARCtangent-A5.2 processor. If you wish to compile for an ARCtangent-A5 processor without the barrel shifter, use option **-core1** or **-core2** in addition to **-a5**.

-core5 — Generate code specific to core version 5 (ARCtangent-A4 and previous)

Option **-core5** instructs the compiler to generate code for version 5 of the processor core. This option links in a breakpoint table (discontinued as of version 7) and is the default for ARCtangent-A4 and previous. Option **-core5** sets macro **_ARCVER** to `5`.

A breakpoint table lets you set software breakpoints in processors that predate the **BRK** instruction. For more information about breakpoints, see the *MetaWare Debugger User's Guide*.

-core6 — Generate code specific to core version 6 (ARCtangent-A4 and previous)

Option `-core6` instructs the compiler to generate code for version 6 of the processor core. This option links in a breakpoint table (discontinued as of version 7). Option **-core6** sets macro **_ARCVER** to `6`.

A breakpoint table lets you set software breakpoints in processors that predate the **BRK** instruction.

-core7 — Generate code specific to core version 7 (ARCtangent-A4 and previous)

Option **-core7** instructs the compiler to generate code for version 7 of the processor core. A breakpoint table is no longer required beginning with version 7. Option **-core7** sets macro **_ARCVER** to `7`.

A breakpoint table lets you set software breakpoints in processors that predate the **BRK** instruction.

-core8 — Generate code specific to core version 8 (ARCtangent-A4)

Option **-core8** instructs the compiler to generate code for version 8 of the processor core, corresponding to ARCtangent-A4. Option **-core8** sets macro **_ARCVER** to `8`.

-rf16 — Support "register file 16" reduced register set

For ARC 600 Only

Option **-rf16** instructs the compiler to generate code for a 16-entry register file, and links run-time libraries compiled for this reduced register set.

NOTE This option works only in conjunction with **-arc600**.
When you use this option, registers `r4` to `r9` and `r16` to `r25` are not available.

Extension Driver Options

The MetaWare C/C++ compiler provides options to enable extension instructions. Some of the extension options listed in this section affect both the compiler and the assembler, while others affect only the assembler.

Assembly Files Included

Most extension options include assembly files to define the associated extension instructions.

- Most extension options include `bin/arcexlib.s` to define the associated extension instructions. If you want to use your own version of `arcexlib.s`, specify option **-Hnoarcexlib** in addition to your extension options.
- Option **-Xsimd** includes `bin/arcsimd.s` to define the SIMD instructions. If you want to use your own version of `arcsimd.s`, specify option **-Hnoarcsimd** in addition to **-Xsimd**.

Preprocessor Macros

All of these options also define a preprocessor macro to both the compiler and the assembler. You can use the macros to provide alternate code for when the extension is present and when it is not. You can access the macros from MetaWare C/C++ or assembly, as shown in the following examples.

Example 9 Accessing Preprocessor Macros from C/C++

```
#ifdef __Xmult32
  /* code that uses scoreboarded multiply instruction */
#else
  /* same operation without scoreboarded multiply */
#endif
```

Example 10 Accessing Preprocessor Macros from Assembly

```
.ifdef __Xmult32
    ; code that uses scoreboarded multiply instruction
.else
    ; same operation without scoreboarded multiply
.endif
```

For more information about extensions, see the *Extension Instructions Reference* and *DSP Options Reference* for your specific target processor.

-Hnoxcheck — Disable run-time checking for ARC extensions

Specify **-Hnoxcheck** to disable run-time checking for ARC extensions. When used with the driver when compiling or assembling, **-Hnoxcheck** eliminates the output of extension-checking information. When used with linking, **-Hnoxcheck** causes the extension-checking information in the object files to be ignored at run time.

This option disables extension checking by including a stub module in the object file. To suppress inclusion of the stub module, specify option **-Hnoxcheck_obj** as well.

-Hnoxcheck_obj — Exclude -Hnoxcheck stub from link

When you disable run-time extension checking with **-Hnoxcheck**, an object file is included in the link to stub out the extension checking. Option **–Hnoxcheck_obj** excludes the stub module from the link.

-Xadds — Enable support for ADDS and SUBS instructions

Option **-Xadds** enables support for the saturating addition and subtraction extension instructions (**ADDS** and **SUBS**). This option affects only the assembler.

This option defines preprocessor macro **__Xadds**.

-Xbarrel_shifter — Enable support for barrel-shifter instructions

Option **-Xbarrel_shifter** enables support for the barrel-shifter instructions and defines preprocessor macro **__Xbarrel_shifter**.

For ARCtangent-A5, this option is specified by default unless you specify **-core1** with **-a5**.

For ARCtangent-A4, the compiler implements multi-bit shifting inline with either sequential single-bit shifts or with a zero-delay loop. Specifying **-Xbarrel_shifter** causes the compiler to emit single multi-bit shift instructions instead.

-Xcrc — Enable support for CRC instruction

Option **-Xcrc** enables support for the variable polynomial CRC accelerator instruction for ARCtangent-A5 and later. This option affects only the assembler and defines preprocessor macro **__Xcrc**.

-Xdpfp[_*fast*|_*compact*] — Enable double-precision floating-point extensions

For ARC 600 and Later

Option **-Xdpfp** enables the double-precision floating-point extensions and defines preprocessor macro **__Xdpfp**. We recommend adding toggle `Dpfp_div_em` to use faster routines during emulation of double-precision floating-point division.

You can use an optional modifier to further specify your extensions. For the fast double-precision extensions, specify **-Xdpfp_*fast***; for the compact double-precision extensions, specify **-Xdpfp_*compact***. If you do not specify *_fast* or *_compact*, the default is compact.

Used with the compiler, this option allows inline generation of floating-point extension instructions. Used with the assembler, this option enables these instructions in the assembler. Used with the linker, this option links an additional library with ANSI Standard C math functions compiled with floating-point extensions to override the default emulated floating-point versions from the C library. For information about using the driver to separately invoke the compiler, assembler, or linker, see Chapter 3 — Using the Driver.

NOTE When you use this option to debug with the instruction-set simulator, you must inform the debugger of the location of the support DLLs generated by ARChitect during the "Build test scripts" phase. Otherwise, the debugger issues an error message.

Architect issues a message such as `Writing unprocessed file 'C:\ARC\Projects\project8\build\lib\win32\dpfpfast_arc600_iss.dll'` when generating the DLLs. Specify this file to the debugger using **-simext_dlls=***location*.

-Xdsp_packa — Use DSP Pack A Extension Instructions

For ARCtangent-A5 and later

Option **-Xdsp_packa** enables the DSP Pack A extension instructions and defines macro **__Xdsp_packa** to the compiler and assembler. This option is primarily intended for use when invoking the **mcc** driver on assembly files containing DSP Pack A extension instructions.

-Xdvbf — Enable support for ARCtangent-A4 dual Viterbi instruction

Option **-Xdvbf** enables support for the dual-word Viterbi decoder butterfly instruction. This option affects only the assembler and defines preprocessor macro **__Xdvbf**. This option accelerates Viterbi decoding algorithms by performing multiple add-compare-select operations in hardware.

For the ARCtangent-A5 version of this option, see **-Xvbfdw**.

-Xea — Enable support for extended arithmetic instructions

For ARCtangent-A5 and Later

Option **-Xea** enables support for the extended arithmetic instructions. This option defines preprocessor macro **__Xea**.

-Xlib — Enable commonly used extension options

Option **-Xlib** defines the most commonly used extension options with a single option.

Option **-Xlib** has different behavior depending on the target processor series. For all processor series, this option corresponds to the MetaWare debugger option of the same name.

For ARCtangent-A4, **-Xlib** enables the following list of extension options:

-Xbarrel_shifter
-Xnorm
-Xmult32
-Xswap
-Xmin_max

For ARCtangent-A5 and ARC 600, **-Xlib** enables the following list of extension options (**-Xbarrel_shifter** and **-Xmin_max** are part of the base case for these two processor series):

-Xmult32
-Xnorm
-Xswap

For ARC 700, **-Xlib** enables **-Xmpy** (**-Xbarrel_shifter**, **-Xmin_max**, **-Xnorm**, and **-Xswap** are part of the base case).

-Xlibd — Enable commonly used extension options with DSP compatibility

For ARCtangent-A5 and ARC 600 Only

Option **-Xlibd** defines the most commonly used extension options with a single option.

-Xmult32d
-Xnorm
-Xswap

This option is similar to option **-Xlib**, but includes the DSP-compatible 32-bit multiplier **-Xmult32d**.

-Xmin_max — Enable support for MIN and MAX instructions

For ARCtangent-A4 Only

Option **-Xmin_max** enables support for the **MIN** and **MAX** instructions. This option affects the compiler and the assembler. This option also defines preprocessor macro **__Xmin_max**.

-Xmpy — Enable the ARC 700 general-purpose multiply extension

For ARC 700 and Later Only

Option **-Xmpy** directs the compiler to generate code for the general-purpose multiply extension available with the ARC 700 and later processors. This option also defines preprocessor macro **__Xmpy**.

-Xmul32x16 — Enable 32x16 multiply instructions

Option **-Xmul32x16** causes the compiler to perform multiplication using the 32x16 multiply extension instructions. Either **-Xmult32** or **-Xmpy**, if also specified, takes precedence over **-Xmul32x16** in the compiler.

This option also defines preprocessor macro **__Xmul32x16**.

-Xmul_mac — Enable support for MULMAC extension instructions

The MULMAC extension instructions have been deprecated; use the **-Xxmac*** options instead.

-Xmult32 — Enable support for scoreboarded multiplier instructions

Option **-Xmult32** enables support for the 32-by-32 scoreboarded multiplier instructions. This option also defines preprocessor macro **__Xmult32**.

-Xmult32d — Remap registers to allow multiply and XMAC together

For ARCtangent-A5 and ARC 600 Only

Option **-Xmult32d** provides an alternative mapping of the `mlo` multiplier register so that the multiply extension can co-exist with the XMAC extension block. When you specify **-Xmult32d**, the compiler does not use register `mmid` (`r58`), and maps register `mlo` (normally `r57`) to `r58` instead.

This option links run-time libraries compiled with this form of multiplier if the **-Xnorm** and **-Xswap** options are also specified.

> **TIP** Use single option **-Xlibd** instead of **-Xmult32d -Xnorm -Xswap**.

Related Topic

- Rebuilding the C Run-Time Libraries on page 241

-Xnorm — Enable support for the NORM instruction

Option **-Xnorm** enables support for the **NORM** instruction. This option also defines preprocessor macro **__Xnorm**.

By default, the compiler implements integer divide operations by calling a library function. If you specify **-Xnorm** for the compilation, the compiler calls an alternate library function that uses the **NORM** instruction. This alternate function is also supplied with the base library.

Selecting **-Xnorm** can also speed up the inline multiply algorithm when the 32-by-32 multiplier (option **-Xmult32**) is not selected.

> **NOTE** For ARCtangent-A5 core versions 1 and 2 and for all ARCtangent-A4 processors, you must specify both **-Xnorm** and **-Xbarrel_shifter** to use the **NORM** instruction for division.

-Xscratch_ram — Enable support for the scratch RAM extension

Option **-Xscratch_ram** enables support for the scratch RAM extension. This option affects only the assembler and defines preprocessor macro **__Xscratch_ram**.

-Xsimd — Enable SIMD Extensions

For ARC 700 Only

Option **-Xsimd** enables the ARC Media SIMD extension instructions. This option defines preprocessor macro **__Xsimd**.

By default, this option includes `bin/arcsimd.s` to define the SIMD instructions. If you want to use your own version of `arcsimd.s`, specify option **-Hnoarcsimd** in addition to **-Xsimd**.

-Xspfp[_*fast*|_*compact*] — Enable single-precision floating-point extension instructions

For ARC 600 and Later

Option **-Xspfp** enables the single-precision floating-point extension instructions and defines preprocessor macro **__Xspfp**.

For the fast single-precision extensions, specify **-Xspfp_***fast*; for the compact single-precision extensions, specify **-Xspfp_***compact*. If you do not specify _*fast* or _*compact*, the default is compact.

Used with the compiler, this option allows inline generation of floating-point extension instructions. Used with the assembler, this option enables these instructions in the assembler. Used with the linker, this option links an additional library with ANSI Standard C math functions compiled with floating-point extensions to override the default emulated floating-point versions from the C library. For information about using the driver to separately invoke the compiler, assembler, or linker, see Chapter 3 — Using the Driver

> **NOTE** When you use this option to debug with the instruction-set simulator, you must inform the debugger of the location of the support DLLs generated by ARChitect during the "Build test scripts" phase. Otherwise, the debugger issues an error message.
>
> Architect issues a message such as `Writing unprocessed file 'C:\ARC\Projects\project8\build\lib\win32\dpfpfast_arc600_iss.dll'` when generating the DLLs. Specify this file to the debugger using **-simext_dlls=***location*.

-Xswap — Enable support for the SWAP instruction

Option **-Xswap** enables support for the **SWAP** instruction. This option defines preprocessor macro **__Xswap**.

-Xtimer0 — Use timer 0 extension

Option **-Xtimer0** enables assembler and linker support for enhanced system timer 0.

When you link using this option, it directs the driver to link the program so that execution profiling uses the timer 0 extension. Use this option in conjunction with option **-p** or **-pg** to instrument the program for execution profiling. For more information, see Execution Profiling on page 293.

> **NOTE** We recommend you not use option **-Xtimer0** for profiling on the MetaWare debugger simulator, as option **-Xtimer0** provides only a statistical or approximate profile. For maximal accuracy, use the simulator's instruction counts by profiling using the MetaWare debugger.

This option defines preprocessor macro **__Xtimer0**.

-Xtimer1 — Use timer 1 extension

Option **-Xtimer1** enables assembler and linker support for enhanced system timer 1.

When you link using this option, it directs the driver to link the program so that execution profiling uses the timer 1 extension. Use this option in conjunction with option **-p** or **-pg** to instrument the program for execution profiling. For more information, see Execution Profiling on page 293.

NOTE We recommend you not use option **-Xtimer1** for profiling on the MetaWare debugger simulator, as option **-Xtimer1** provides only a statistical or approximate profile. For maximal accuracy, use the simulator's instruction counts by profiling using the MetaWare debugger.

This option defines preprocessor macro **__Xtimer1**.

-Xvbfdw — Enable support for ARCtangent-A5 dual Viterbi instruction

Option **-Xvbfdw** enables support for the dual-word Viterbi decoder butterfly instruction for ARCtangent-A5. This option affects only the assembler and defines preprocessor macro **__Xvbfdw**. This option accelerates Viterbi decoding algorithms by performing multiple add-compare-select operations in hardware.

For the ARCtangent-A4 version of this option, see **-Xdvbf**.

-Xvraptor — Build for VRaptor Instruction Set

For ARC 700 Only

This option defines macros **__Xvraptor** and **__Xsimd** (for both C and assembly files) and defines the VRaptor ISA to the assembler.

By default, this option includes `bin/arcsimd.s` to define the SIMD instructions. If you want to use your own version of `arcsimd.s`, specify option **-Hnoarcsimd** in addition to **-Xvraptor**.

-Xxmac_16 — Enable support for 16-bit XMAC extensions

For ARCtangent-A4 Only

Option **-Xxmac_16** enables support for the 16-bit XMAC extensions for the assembler. This option affects only the assembler and defines preprocessor macros **__Xxmac** and **__Xxmac_16**.

-Xxmac_24 — Enable support for 24-bit XMAC extensions

For ARCtangent-A5 and Later

Option **-Xxmac_24** enables support for the 24-bit XMAC instructions. This option defines preprocessor macros **__Xxmac** and **__Xxmac_24**.

By default, the compiler implements integer multiply inline with a zero-delay loop. Selecting **-Xxmac_24** causes the compiler to use the dual 24x24 DSP multiply extension to perform multiplication.

-Xxmac_d16 — Enable support for dual 16-bit XMAC extensions

For ARCtangent-A5 and Later

Option **-Xxmac_d16** enables support for the dual 16-bit XMAC instructions. This option defines preprocessor macros **__Xxmac** and **__Xxmac_d16**.

By default, the compiler implements integer multiply inline with a zero-delay loop. Specifying **-Xxmac_d16** causes the compiler to use the dual 16x16 DSP multiply extension to perform multiplication.

-Xxy — Enable support for the XY memory extension

Option **-Xxy** enables assembler support for the XY memory extension. This option defines preprocessor macro **__Xxy**. For ARCtangent-A5 and later, this option is also required to use the XY memory features of DSP-C.

Floating-Point Driver Options

You can place a floating-point option on the **hcarc** command line to specify which type of floating-point code the compiler should emit.

Default

Floating-point option **-fsingle** is specified by default.

NOTE For the floating-point *extensions*, see options **-Xdpfp** and **-Xspfp** in Target-Processor Driver Options on page 42.

-fdouble — Use not less than double-precision floating-point arithmetic

Option **-fdouble** specifies that all floating-point computations are to be done in no less than double precision. Operands of type **float** are widened appropriately. This option can improve the accuracy of floating-point arithmetic at the expense of execution speed.

NOTE For the double-precision floating-point *extension*, see option **-Xdpfp**.

-fsingle — Use single-precision floating-point arithmetic

Option **-fsingle** specifies that single-precision arithmetic is to be used in floating-point computations involving only **float** operands.

NOTE For the single-precision floating-point *extension*, see option **-Xspfp**.

-fsingle2 — Do not widen float types to double types

Option **-fsingle2** suppresses the widening of **float** arguments to type **double** in calling non-prototyped functions and in passing varying arguments. This option, in conjunction with **-fsingle**, can improve performance in some programs.

NOTE Option `-fsingle2` turns Off toggle `Widen_float_args`.

CAUTION Linking a file compiled with option `-fsingle2` to one compiled without it can produce unpredictable results.

General Driver Options

This section lists driver options in alphabetical order. For specific options related to target processors and floating point, see the following:

- Target-Processor Driver Options on page 42.
- Floating-Point Driver Options on page 51

-A- — Recognize only predefined assertions and macros that begin with two underscores

Option **-A-** instructs the compiler to disregard all predefined assertions and all predefined macros except those that begin with two underscores (__). For more information about assertions, see Assertions (Preprocessor Predicates) on page 37.

-A*name*(*token*) — Create a preprocessor #assert directive

Option **-A** makes *token* a predicate of *name*. The effect is the same as using the **#assert** preprocessing directive. For more information about assertions, see Assertions (Preprocessor Predicates) on page 37.

-c — Generate object file but suppress linking

Option **-c** suppresses invocation of the linker, forcing an unlinked relocatable object file to be produced even if only one file is compiled. By default, the object file is written to a file in the current working directory with the same file name as the C or C++ source file, but with the source-file extension replaced by `.o`.

> **NOTE** You can use option **-o** to specify an alternate name for the object file. If you specify **-o** *output* and exactly one file is being compiled, *output* is taken as the name of the object file to be produced.

-D*name*[=*def*] — Define a name to the preprocessor

Option **-D** defines the name *name* to the preprocessor. The effect is the same as using the preprocessor directive **#define**.

- If you do not specify a *def* value, *name* is defined to be 1 (one).
- If the *def* value includes whitespace, you must enclose the entire option in double quotes.
- If the *def* value includes double quotes, you must escape each one with a backslash (\").

For example:

```
-Dname              expands to: #define name 1
-Dname=xx           expands to: #define name xx
"-Dname=xx yyy"     expands to: #define name xx yyy
"-Dname=\"xx yyy\"" expands to: #define name "xx yyy"
```

Option **-U***name* undefines *name*.

-dwarf1 — Generate debugging information in DWARF 1 format

Option **-dwarf1** causes the compiler to use the DWARF 1 format for debugging information. Option **-dwarf1** specifies only the *format* of the debugging information; to generate debugging information, you must also specify option **-g**.

NOTE This option applies only to C and C++ source, not assembly source (`.s` files) assembled using the driver.

-E — Preprocess without compiling; write preprocessed file to standard output

Option **-E** invokes the macro preprocessor but does not compile the source file. Instead, the preprocessed source file is written to standard output. See also option **-P**.

-g — Generate debugging information

Option **-g** generates debugging information and provides the linker with information needed to link the file for debugging. Option **-g** turns off many of the optimizations the compiler otherwise performs if they would hamper the debugger.

Option **-g** turns on toggle `Source_code_order` for easier debugging at the assembly language level

Option **-g** specifies optimization level 1 if you do not specify another optimization level.

The compiler generates debugging information in DWARF version 2 format by default. For DWARF version 1, specify option **-dwarf1** in addition to option **-g**.

See Debugging Optimized Code on page 170 on the relationship between optimizations and debugging.

-g0 — Generate debugging information with line numbers only

Option **-g0** turns on toggle `Emit_line_records`, which generates line-number information only, instead of the substantial debugging information that **-g** generates.

Option **-g0** allows source line debugging information to be emitted without impairing the optimization of the generated code. See Debugging Optimized Code on page 170 on the relationship between optimizations and debugging.

-g1 — Generate only "optimization-friendly" debugging information

Option **-g1** generates debugging information with minimal impact on optimization. This option generates more than the line numbers that **-g0** generates, including information on constructs such as types and variables.

CAUTION When you use **-g1** with optimized code, source-level examination of program data is reliable only at function entry and exit points. Examination of data within the body of a function might yield unreliable results; for complete debugging information, use option **-g**.

-h[*string*] — Search mini-help files for information about driver options or about *string*

Specify option **-h*string*** to search the mini-help files (provided with the distribution) for text relating to *string*. When a match is found, any help text containing *string* is returned. *string* is case sensitive and can be quoted. A pair of double quotes ("") within a quoted *string* denotes one double quote (").

To see a listing of all the driver options (and what they do), use the following command line:

```
mcc -h
```

See Getting Help on the Command Line on page 28 for more information.

-H — Display names of files included during compilation

Option **-H** directs the compiler to write to standard error the name of each file included during compilation, one name per line. The equivalent toggle is `List_includes`.

-Hall_sdata — Place all data in small-data sections

Option **-Hall_sdata** directs the compiler to place all data in small-data sections. When the compiler allocates storage for data in a small-data section, references to the data are faster, because such data can be addressed with signed offsets from the base of the small-data section.

> **NOTE** The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** into small data unless you qualify them as **_Near**.

Option **-Hall_sdata** is off by default.

These are related compiler controls:

- options **-Hnosdata** and **-Hsdata0**
- pragmas `Push_small_data` and `Pop_small_data`
- toggles `Unsized_arrays_in_small_data`, `Use_sdata`, and `Common_in_small_data`

Option **-Hall_sdata** turns on toggle `Unsized_arrays_in_small_data`.

> **CAUTION** Use option **-Hall_sdata** only if your application has limited data requirements. Otherwise, placing all data in small-data areas might cause a link error.

Related Topic

- Using Small-Data Sections on page 214

-Hanno — Annotate assembly listing with lines from the source file

Option **-Hanno** directs the compiler to annotate the assembly file output with lines from the original source file. Use this option in conjunction with option **-S**.

-Hansi — Accept only ANSI C Standard-conforming programs

Option **-Hansi** directs the compiler to accept only programs that conform to the ANSI C Standard.

Option **-Hansi** also defines the **__HIGHC_ANSI__** preprocessor macro and undefines the **__HIGHC__** preprocessor macro.

-Haom — Compile application to use the Automated Overlay Manager

Use this option to compile code with overlays in mind (even if no overlay is in the current file). This option does the following:

- Reserves the registers used by the Automated Overlay Manager
- Turns off extension checking
- Appends the `inc/aom` directory to the search path
- Includes the Automated Overlay Manager (`libaom.a`) in the link

For more information about the Automated Overlay Manager, see the *Automated Overlay Manager User's Guide*.

Related Topics

- -Haom_nolib — Compile overlay-safe code without Automated Overlay Manager run time
- *Automated Overlay Manager User's Guide*

-Haom_nolib — Compile overlay-safe code without Automated Overlay Manager run time

Use option **-Haom_nolib** to build code compatible with the Automated Overlay Manager but not include the supplied AOM run time (`libaom.a`) at the link stage.

> **NOTE** If your application uses overlays or calls overlay functions, you must use option **-Haom**.

Related Topics

- -Haom — Compile application to use the Automated Overlay Manager
- *Automated Overlay Manager User's Guide*

-Haom_overlay_only — Prepare module for inclusion in a overlay package

Use this option on code modules you plan to include in an overlay package. This option removes all libraries and startup code and tells the compiler not to emit any `.rodata` or other sections except those required for overlays. For more informaton about overlays, see the *Automated Overlay Manager User's Guide*.

This option implies option **-Haom**; you need not specify both.

> **CAUTION** The compiler issues an error message if your code causes a `.rodata` (read-only data) section to be emitted within the scope of this option, such as **char** `*msg = "hello world"`.

-Hasext=*ext-list* — Recognize new assembly-language filename extension

Option **-Hasext=***ext-list* alerts the driver to recognize a new source-file extension as designating an assembly-language source file. Argument *ext-list* is a comma-separated list of filename extensions without the leading dot (`'.'`). The filename extensions in *ext-list* are added to the built-in list. For example, to designate files with extension `.arc` as assembly files:

```
mcc -c -Hasext=arc file.arc ..
```

-Hasmcpp — Process C-style preprocessing directives in assembly source file

Option **-Hasmcpp** invokes the C preprocessor to process C-style preprocessing directives in assembly source files passed to the driver. This preprocessing occurs before the assembler begins processing the file.

Refer to the *ELF Assembler User's Guide* for a discussion of the built-in assembler macro processor.

> **CAUTION** Using **-Hasmcpp** can lead to problems if the C preprocessor finds invalid C tokens in the assembly source. For example, unmatched C comments and unpaired quotes may result in preprocessor errors. Before using this option, make sure you have only legal C preprocessing tokens in your assembly source code.

-Hasopt=*xxx* — Pass assembler options to the assembler

Option **-Hasopt** passes one or more assembler options to the assembler. For example, to pass option **-op1** to the assembler, you can execute this command:

`mcc` *files* `... -Hasopt=-op1`

To specify multiple options, you can use **-Hasopt** on the command line multiple times, or use it once and separate the options with commas. The driver passes multiple options in the order you specify them. You must use a comma to separate an option from its corresponding argument(s).

For example, if you want to pass options **-op1** and **-op2** *arg* to the assembler, use one of the following commands:

`hcarc` *files* `... -Hasopt=`*-op1*`,`*-op2*`,`*arg*

`hcarc` *files* `... -Hasopt=`*-op1* `-Hasopt=`*-op2*`,`*arg*

For a list of assembler options, see the *ELF Assembler User's Guide*.

-HB — Compile in big-endian mode

Option **-HB** turns on big-endian compilation mode to generate code for big-endian targets. See also option **-HL**, which has the opposite effect.

The default is little-endian mode.

Option **-HB** directs the driver to link in big-endian versions of `crt1.o` (startup code) and the big-endian MetaWare Run-Time Libraries for your processor series.

> **NOTE** If you plan to compile and link your application separately, use the driver at both steps and specify **-HB** at both the compile step and the link step.

-Hbatch[=*filename*] — Create a batch file to call the compiler and linker

Option **-Hbatch** instructs the driver to create a batch file *filename*, instead of calling the compiler and linker directly. Use *filename* to invoke the compiler and linker with the remaining options you specified on the driver command line. When you omit =*filename*, the driver sends the batch file contents to standard output.

The driver creates temporary files that are subsequently deleted by the batch file generated when you specify **-Hbatch**. Thus, you can successfully invoke the batch file only once unless you edit it to remove the delete-file commands. To create a batch file without temporaries, use option **-Hbatchnotmp**.

> **CAUTION** If you invoke the driver a second time with option **-Hbatch** specified, the driver overwrites the first and possibly the second temporary files.
>
> To preserve the batch commands for later use, rename the temporary files and edit the batch file to reference the new names.
>
> Option **-Hbatch** can take a format string in place of a specific filename. See Format Strings for Output-File Names on page 84 for details.

-Hbatchnotmp[=*filename*] — Create a batch file to call the compiler and linker without temporaries

Option **-Hbatchnotmp** instructs the driver to create a batch file *filename*, instead of calling the compiler and linker directly. Use *filename* to invoke the compiler and linker without retyping the remaining options you specified on the driver command line. When you omit =*filename*, the driver sends the batch file contents to standard output.

Compare to option **-Hbatch**, which also creates temporary files that are subsequently deleted by the batch file.

-Hccm — Build for closely coupled memory (a Harvard architecture)

Option **-Hccm** builds for closely coupled memory (a Harvard architecture) by preventing loads and stores in executable code memory. This option thus also prevents the compiler from placing address constants in code.

-Hcext=*ext-list* — Recognize new C source-file extension

Driver option **-Hcext**=*ext-list* alerts the driver to recognize a new source-file extension as designating a C-language source file. Argument *ext-list* is a comma-separated list of filename extensions without the leading dot ('`.`'). The filename extensions in *ext-list* are added to the built-in list.

Compare existing C++ option **-Hcppext**=*ext-list*, where the extensions in *ext-list* override the built-in defaults. By contrast, **-Hcext**=*ext-list* simply adds new extensions to the default list. For example, to designate files with extension `.pp` as C source files:

```
mcc -c -Hcext=pp file.pp ...
```

-Hcomdat — Manually enable COMDAT solution for template instantiation

C++ only

Option **-Hcomdat**, which is specified by default, manually enables COMDAT support. The COMDAT solution is the default MetaWare C/C++ solution for template instantiation and virtual function table generation. When COMDAT support is enabled, these toggles are turned on:

- `Inline_common`
- `RTTI_common`
- `Template_common`
- `Vtable_common`
- `Class_common_ok`

A related toggle, which option **-Hcomdat** does not turn on, is `Class_common_always`.

To disable COMDAT support (for example, if you are using a linker that does not support COMDAT sections), specify option **-Hnocomdat**.

For more information about the COMDAT solution, see the *MetaWare C/C++ Language Reference*.

NOTE Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

-Hcplus — Include C++ header files and link C++ run-time library

C++ only

Option **-Hcplus** includes C++ header files at compile time and links in the C++ run-time library at link time. The C++ compiler uses this option by default. Option **-Hnocplus** has the opposite effect. For more information, see Compiling and Linking C++ Modules on page 30.

-Hcpp — Use the outboard macro preprocessor

For UNIX and Linux Hosts Only

Option **-Hcpp** directs the compiler to use the outboard macro preprocessor supplied with the system, rather than the inboard preprocessor. Option **-Hnocpp** has the opposite effect.

> **NOTE** In most versions of UNIX before System V Release 4, the outboard UNIX preprocessor does not comply with the ANSI C Standard.

If you specify option **-Hcpp**, the compiler does not read `.pro` files (include files that can invoke pragmas and options) until after preprocessing.

For more information, see Using the Preprocessor on page 35 and Using .pro Files on page 39.

-Hcppext=*ext1*, *ext2*, ...*extn* — Specify the C++ source-file extensions

For C++ Only

Option **-Hcppext** specifies which file-name extensions represent C++ files. The defaults are `.C` (for Linux- and UNIX-hosted environments only), `.cc`, `.cpp`, and `.ii`. You can override the defaults by listing other extensions (without the preceding "."). For example, this option allows only files with extensions `.cpl` and `.CC` to be interpreted as C++ source:

```
-Hcppext=cpl,CC
```

-Hcppfile_lvl=*n* — Specify the level of C++ compilation by file type

Option **-Hcppfile_lvl** specifies the C++ level of files recognized as C++ files. Contrast with option **-Hcpplvl**, which specifies the C++ level for all files in the compilation.

- For definitions of the levels you may specify for *n*, see option **-Hcpplvl**.
- For information on the extensions recognized as C++ by default and how to change them, see Recognized Input File Types on page 27.

-Hcpplvl=*n* — Specify the level of C++ compilation

C++ Only

Option **-Hcpplvl** specifies the level (*n*) of C++ compilation to be used by the compiler, where *n* is the numeric value of one of the levels listed below. The default is level 4 C++.

> **CAUTION** Levels 2 and greater cause the compiler to treat `.c` files as C++ files. This might result in link errors unless you compile the entire application at the same C++ level. Use option **-Hcppfile_lvl** to specify a C++ level only for files recognized as C++ files.

Level 1 C++

Level 1 is "weak" C++. All aspects of C++ implemented by MetaWare are available, through possibly alternative syntax, but certain aspects of C++ that change the semantics of ANSI Standard C programs are not enabled.

Use level 1 only for Incremental C++; see "Migrating from C to C++" in the *MetaWare C/C++ Language Reference*.

The following characteristics apply to level 1:

- The type of an enumeration literal is **int** rather than **enum**.
- The type of a character is an integer type rather than type **char**.
- The storage class of constant objects is not **static** by default.
- Functions are not by default overloadable.
- An **int** can be converted into an **enum** type.
- A **struct** or **union** does not generate a local scope, but a **class** does.
- Anonymous unions behave as in ANSI Standard C — that is, they do (almost) nothing.
- Class names do not become **typedef** names.
- In a function declaration, an argument specification of `()` is not interpreted as `(void)` unless the function appears within a class.
- The following words are not reserved; you can access these C++ features only by using alternative syntax. For more information, see "Migrating from C to C++" in the *MetaWare C/C++ Language Reference*.
  - **catch**
  - **class**
  - **const_cast**
  - **delete**
  - **dynamic_cast**
  - **friend**
  - **inline**
  - **mutable**
  - **namespace**
  - **new**
  - **operator**
  - **private**
  - **protected**
  - **public**
  - **reinterpret_cast**
  - **static_cast**
  - **template**
  - **this**
  - **throw**

— **typename**

— **typeid**

— **using**

— **virtual**

Level 2 C++

Level 2 means that all aspects of C++ supported by MetaWare are implemented, but the following prohibitions in the C++ language are not enforced:

- The C++ language requires that a reference can be bound to either (a) something of the same type (or of a derived type), or (b) something of a compatible type; but the reference must be to a **const**. The restriction that the reference is to a **const** is not enforced at level 2, due to prior practice.
- Converting an **int** into **enum** generates only a warning.
- Calling an undeclared function is allowed. The compiler assumes an undeclared function is an externally available C function.
- Initializing an array of *n* **char**s with a string of length *n* is allowed.

Level 3 C++

Level 3 means that all aspects of C++ supported by MetaWare are implemented, including enforcement of the restrictions that were relaxed at level 2. This level does not support newer features that might break existing code.

Level 4 C++

This level includes all C++ features supported by the compiler that are not available at the default level of 3 because of compatibility concerns with existing code, but do not require additional code or data overhead. This level supports the **bool** data type by default, and local variables declared in the initialization portion of a **for** loop are strictly local to the scope of the loop.

Level 5 C++

This level includes exception-aware classes for exception handling and full run-time type-identification information. These features require additional execution overhead and data space, or both.

-Hcrt=*path_or_file_name* — Specify a run-time startup object file to be linked in

Option **-Hcrt** directs the driver to search at link time for run-time startup object file *path_or_file_name* instead of searching for the default run-time startup file (`crt1.o`).

If `path_or_file_name` is a file name with no path information, the driver searches for the specified file in the appropriate subdirectory of *install_dir*`/lib`.

For more information about library subdirectories, see *Getting Started* or the `readme` file.

For information about run-time startup files for embedded applications, see Working with Startup Modules on page 227.

-Hc_wrap={*xxx*|^*yyy*} — Interpret header files included in C++ files as C, not C++

For C++ Only

Option **-Hc_wrap** directs the compiler to treat files included in C++ files as C header files rather than as C++ header files.

If an include file has *xxx* anywhere in its name (for **-Hc_wrap**=*xxx*), or begins with *yyy* (for **-Hc_wrap**=^*yyy*), the include file is preceded by file `c_wrapl.h` and followed by file `c_wrapr.h`.

These two files (included in the compiler distribution) cause the include file to be interpreted as a C header file, not a C++ header file.

You can use option **-Hc_wrap** to include C header files with C++ code. A common usage on UNIX-based systems is **-Hc_wrap**=`^/usr/include`. You might have other C include files that you want to use with C++ programs; you can specify **-Hc_wrap** any number of times.

Toggle `Empty_is_void` is turned off in file `c_wrapl.h` and it is popped in file `c_wrapr.h`.

-Hdepend=*depend-dir*/ — Specify directory for makefile dependency file

Driver option **-Hdepend=***depend-dir*/ specifies a directory where the makefile dependency file should be placed. Specify the directory with a trailing '/'. Use this option with a makefile-dependency option such as **-Hmake**, **-Humake**, and **-Hmakeof**.

-Hecpp — Disable certain C++ features for Embedded C++ support

For Embedded C++ Only

Disabling certain C++ features and keywords facilitates use of C++ in embedded applications. Option **-Hecpp** defines macro **__embedded_cplusplus** and directs the compiler to disable the following C++ keywords and features:

> **NOTE** Some of these features and keywords bear a run-time cost in space or speed, while others do not, as indicated in this list.

- Keywords:
  - **const_cast** (no run-time cost)
  - **dynamic_cast** (run-time cost)
  - **mutable** (no run-time cost)
  - **reinterpret_cast** (no run-time cost)
  - **static_cast** (no run-time cost)
  - **typeid** (run-time cost)
  - **typename** (no run-time cost)
  - **using** (no run-time cost)
- Features:
  - exceptions (run-time cost)
  - multiple inheritance (run-time cost)
  - namespaces (no run-time cost)
  - run-time type information (run-time cost)
  - templates (no run-time cost)
  - virtual bases (run-time cost)

Specifying option **-Hecpp** disables the listed features and keywords by turning off the following toggles:

- Support_cast_keywords
- Support_exceptions
- Support_multiple_inheritance
- Support_mutable
- Support_namespaces
- Support_run_time_type_info
- Support_templates
- Support_typename
- Support_vbases

For more information about these toggles, see their listings in Alphabetical List of Compiler Toggles on page 86.

Crafting your own EC++ subset

You can compile with option **-Hecpp** specified, to generate diagnostic messages naming the `Support_*` toggles for features you want to enable. Given those toggle names, you can then turn on the appropriate toggle(s) and recompile with option **-Hecpp** specified.

You can also turn off only those `Support_*` toggles that support the features and keywords you want to disable, then compile without option **-Hecpp** specified.

To generate embedded C++ code without disabling any of the listed keywords and features, specify option **-Hecpp** and also turn off toggle `Ecpp_error`.

-Hefile={@[E]|filename} — If output sent to stderr, redirect it to stdout or to a file

Option **-Hefile** tells the compiler to redirect output that was sent to `stderr`, sending it either to standard output or to a file. This option is particularly useful on systems that do not have a convenient way to redirect `stderr`. This is the syntax of option **-Hefile**:

-Hefile=@	Direct output to `stdout`
-Hefile=`@E`	Direct output to `stderr`
-Hefile=*filename*	Direct output to file *filename*

Option **-Hefile** can take a format string in place of a specific *filename*. See Format Strings for Output-File Names on page 84 for details.

-Hefmat=*format* — Set the format of error messages

Option **-Hefmat** directs the compiler to format error messages according to the variable *format*, where *format* is a quoted string containing a sequence of characters with special meanings. (Related options are **-Hmscerr**, **-Hturboerr**, **-Hunixerr**, and **-Hwfmat**.) For more information, see Changing the Message Format on page 337.

-Heos=*rtos_name* — Select the specified RTOS library

Option **-Heos** with the argument *rtos_name* directs the linker to select the provided support library for a specific RTOS when linking, as required to implement thread safety and thread-local data for that

RTOS. Use option **-Heos** in combination with **-Heoslib**=*pathname* to include the actual RTOS library in the link.

When *rtos_name* is anything other than `mqx`, option **-Heos** specifies a heap 256 K in size. When *rtos_name* is `mqx`, the default heap size is 0. For more information, see Using a Heap on page 213.

The recognized values for *rtos_name* are the following (all lowercase):

ARCtangent-A5 and Later	ARCtangent-A4
`mqx`	`mqx`
`threadx`	`nucleus`
	`threadx`
	`vxworks` (no run-time libraries or startup)

-Heoslib=*pathname* — Specify path to the RTOS library file

Option **-Heoslib** specifies the path name to include the actual RTOS library selected using **-Heos**=*rtos_name*. This option is required for correct placement of the library relative to the list of standard libraries linked. If you are specifying multiple libraries, use commas to separate them.

> **NOTE** You might also have to specify explicit objects on the command line. For details, see your RTOS documentation.

-Hexcept — Compile and link exception-aware code

For C++ Only

Option **-Hexcept** compiles exception-aware code by turning on toggle `Exception_aware_class` and certain optimizations.

Option **-Hexcept** also tells the linker to link in the exception-aware run-time library instead of the exception-unaware run-time library.

To generate exception-aware code, you must both compile and link with option **-Hexcept** specified.

> **CAUTION** When option **-Hexcept** is specified, the compiler might not emit run-time type information (RTTI) for all instances. To force generation of RTTI for all exception-aware code, also turn on toggle `Class_common_always`.

> **NOTE** The toolset distribution does not include exception-aware run-time libraries, because it is unlikely that developers of embedded applications will want to use them.
>
> To obtain exception-aware libraries, you must rebuild the supplied C++ run-time library source code with option **-Hexcept** specified. You can then explicitly link your application with those libraries.

See the *MetaWare C/C++ Language Reference* for details about C++ exception handling. See also option **-Hnonexcept_wrap**.

-Hexesize=*size* — Specify maximum size of executable

Option **-Hexesize** specifies the maximum size of your executable. This option overrides the default maximum of 64 M with the value you specify in *size*. The linker issues an error message if the executable exceeds the specified size.

-Hextfile=file[, file...] — Specify extension directives in assembly files

Option **-Hextfile** specifies assembly-language files that contain extension directives. For example, you can use it to access extensions when your source code contains inline assembly or uses pragma `Intrinsic`.

If you specify more than one file, the driver places them on the assembler command line in the order listed.

NOTE If you specify any of the extension options, the assembler includes the appropriate assembly-language file from *install_dir*/`bin` in addition to the files you specify using **-Hextfile**.

The assembly-language files in *install_dir*/`bin` define the standard extensions that are implemented when you use the extension options. See also the *ELF Assembler User's Guide*.

-Hfirst=*obj_file* — Link specified object file first

Option **-Hfirst** causes the *obj_file* you specify to be placed first in the list of files sent to the linker.

-Hgac=*format* — Set a value for pragma Global_aliasing_convention

Option **-Hgac** sets a value for pragma `Global_aliasing_convention`. If your development environment requires an aliasing convention different from the compiler's default, you might want to preset a value for **-Hgac** in the driver configuration file, as a parameter to `ARGS`.

For more information, see Aliasing External Names on page 154 and Appendix B — Configuring the Driver.

-Hgcc — Improve compatibility with GCC

Use option **-Hgcc** when compiling code written for the GCC compiler from GNU, such as the Linux kernel.

This option relaxes various type-checking constraints in an attempt to mimic GCC behavior. It also turns off toggle `Asm_percent_reg` and defines preprocessor macro **__GNUC__** with the value `3`.

-Hgenbp_data — Instrument application with code to generate branch-profiling data

When you compile your application for the first time, option **-Hgenbp_data** instruments it with code to generate a file containing branch profiling information at execution time. After you obtain the branch-profiling data, you can recompile with option **-Husebp** to optimize your code accordingly.

For more information, see Branch Prediction and Branch Profiling on page 174.

-Hharvard — Prevent loads and stores in executable code memory

Synonym for option **-Hccm**.

-Hheap=*size* — Specify the heap size

Option **-Hheap** overrides the default heap size with the value specified in *size*. When you use option **-Heos** to link with an RTOS, the default heap size is 256 K; otherwise, the default heap size is zero.

> **NOTE** When the heap size is zero, the program allocates memory beginning from symbol `_end` (created by the linker to contain the lowest address not allocated to the program).

-Hhlsize=*size* — Specify hostlink buffer size

If your application uses the hostlink feature for run-time communication with the debugger, you can use option **-Hhlsize=***size* when linking to change the size of the buffer used by the hostlink mechanism to perform I/O. The default buffer size is 1K. Some applications might run faster with a larger buffer size.

> **TIP** You might wish to combine this feature with the use of function **setvbuf()** in your application to increase the size of the `FILE` buffer used by `stdio` functions.

Related Topics

- Providing System Calls for Debugging from a Host on page 229
- *MetaWare Debugger User's Guide*

-Hhostlib=*library* — Specify a library to implement run-time system calls

Use option **-Hhostlib** to specify a host library that implements the system calls required at run time.

> **NOTE** The hostlink library used is whatever you specify after the equals sign. For no library, specify **-Hhostlib=** with nothing after the equals sign (=). Then the linker includes stub routines and a single output buffer instead of a hostlink library.

-Hi — Inline routines type-qualified with _Inline

Use option **-Hi** when you want to use the outboard inliner to inline only routines designated individually with type qualifier **_Inline**, and when you are not specifying any other inlining option on the command line. Option **-Hi** is turned on by default at optimization levels 4, 5, 6, 7, and s. It is turned off by default at optimization levels 0, 1, 2, and 3.

> **NOTE** Option **-g** inhibits inlining.

For more information on inlining, see Chapter 8 — Inlining.

-Hi[*modifier*] — Invoke the outboard inliner during compilation

The **-Hi***modifier* options invoke the outboard inliner on specified categories of functions, as described in the following list:

-Hia	Do not inline functions that have their address taken.
-Hib=*n*	Specify temporary-file buffer size.

-Hic=*n*	Inline functions called *n* times.
-HiC=*t*,*c*	Inline functions with *t* or fewer tree nodes if called *c* or fewer times.
-Hih	Show which functions have been inlined and/or defined.
-Hir	Do not inline recursive functions.
-His=*n*	Do not inline functions with stack size greater than or equal to *n* bytes.
-Hit=*n*	Inline functions with fewer than *n* tree nodes.
-Hiw	Invoke the inliner in multi-module (wide inlining) mode.

NOTE **-Hiw** and **-Hmake** are mutually exclusive.

-Hix	Do not inline exported functions.

NOTE Option **-g** inhibits inlining.

For more information on inlining, see Chapter 8 — Inlining.

-Hinlfactor=*nnn* — Suppress inlining if function would expand by more than specified percentage

Option **-Hinlfactor** prevents a function fro being inlined if inlining would make it more than *nnn* percent larger than the function without inlining.

The default value for *nnn* is 400, meaning that a function that would be more than four times larger with inlining is not subjected to inlining.

Option **-Hinlfactor** affects functions only if their inlined size is more than twice the value of **-Hinlsize**. This prevents suppression of inlining for small functions.

NOTE Option **-Hinlfactor** applies only to the built-in C++ inliner. Option **-Hinlfactor** does not apply to the outboard inliner. For a description of the inliners, see Chapter 8 — Inlining.

-Hinlsize=*size* — Specify the maximum size of inlined functions

For C++ Only

Option **-Hinlsize** specifies the maximum size of functions to be inlined by the C++ compiler's built-in inliner. The greater the value of the variable *size*, the greater the size of functions that can be inlined. The default value for *size* is `125`, which is typically about 15 statements. (Specifically, the value of *size* is the number of expression tree nodes required to represent the function.)

A related option is **-Hinlfactor**.

NOTE Option **-Hinlsize** applies only to the built-in C++ inliner. Option **-Hinlsize** does not apply to the outboard inliner. For a description of the inliners, see Chapter 8 — Inlining.

For information about the outboard inliner, see Chapter 8 — Inlining.

-Hipname=*NAME* — Set the name of the environment variable containing the include-file search path

Option **-Hipname** establishes the name of the environment variable to be used for specifying a directory search path for include files.

The default value for *NAME* is `IPATH`, but you can change this value to agree with conventions used by other compilers. For example, you can specify **-Hipname=**`INCLUDE` or **-Hipname=**`CIPATH`, to avoid conflict with the `IPATH` name of another MetaWare C/C++ compiler or other development projects.

You can also set this value in the driver configuration file as a parameter to `ARGS`. For more information, see Setting an Alternate Directory for Temporary Files on page 35 and Appendix B — Configuring the Driver.

-Hkanji — Recognize Kanji character set

Option **-Hkanji** directs the compiler to recognize the Japanese Kanji (multi-byte) character set in source code. For more information, see *Kanji Support* in the *MetaWare C/C++ Language Reference*.

-Hkeep — Do not delete object file at link time

Option **-Hkeep** prevents the driver from deleting the generated object file when you compile and link a single source file. By default, the driver deletes the object file after the link if there is only one source file.

-Hkeepasm — Do not delete intermediate assembly source files

Option **-Hkeepasm** prevents the driver from deleting the intermediate assembly source files, even when generating an object or executable file.

-HL — Compile in little-endian mode

Option **-HL** turns on little-endian compilation mode to generate code for little-endian targets and link with the little-endian libraries in *install_dir*`/lib/le`. (Compare to option **-HB**, which directs the compiler to compile in big-endian mode.)

If you plan to compile and link your application separately, use the driver for both steps and specify option **-HL** at both the compile step and the link step.

-Hldopt=*xxx* — Pass linker options to the linker

Option **-Hldopt** passes a linker option (or options) to the linker. For example, if you want to pass option *-op1* (representative; not a real option) to the linker, use the following:

`hcarc` *files* `...` `-Hldopt=`*-op1*

To specify multiple options, you can use **-Hldopt** on the command line multiple times, or use it once and separate the options with commas. The driver passes multiple options in the order you specify them. You must use a comma to separate an option from its corresponding argument(s).

For example, to pass options *-op1* and *-op2 arg* to the linker, use one of the following commands:

`hcarc` *files* `...` `-Hldopt=`*-op1*`,`*-op2*`,`*arg*
`hcarc` *files* `...` `-Hldopt=`*-op1* `-Hldopt=`*-op2*`,`*arg*

For a list of linker options, see the *ELF Linker and Utilities User's Guide*.

-Hlibobj=*filename*[, *filename*...] — Link in alternative library object files

Option **-Hlibobj** directs the linker to link in one or more alternative or additional object files located in the library directory. This is the syntax:

`-Hlibobj=`*file.o* [, *file.o* . . .]

> **NOTE** Options **-Hpfnoflt** and **-Hsfnoflt** are equivalent to options **-Hlibobj=pfnoflt.o** and **-Hlibobj=sfnoflt.o**, respectively.

-Hlines=*n* — Specify number of lines printed per listing page

Option **-Hlines** causes one page of source-code listing to be ejected after every *n* lines printed. (Option **-Hlines** is used with the **-Hlist** option, which controls the listing of source code; for more information, see Chapter 15 — Generating Listings.)

The default value of `n` is `60`, an appropriate setting for most 6-lines-per-inch printers, which generally print a maximum of 66 lines per page on 11-inch long paper. (The extra 6 lines per page allowed by the **-Hlines** default produce some extra blank space between pages.)

For 8-lines-per-inch (88 lines per page) printers, *n* can be set to another value; 80 and 82 both produce good results. If *n* is set to 0 (zero), no pages are ejected.

-Hlist[=*filename*] — Generate a source listing

Option **-Hlist** writes a source listing to standard output by setting toggle `List` to on. To direct the listing to a file, use the **-Hlist=***filename* form. Option **-Hlist** can take a format string in place of a specific *filename*.

For more information, see Format Strings for Output-File Names on page 84 and Chapter 15 — Generating Listings.

-Hlm=*xxx* — Enable left margin settings

Option **-Hlm** enables left margin settings. **-Hlm=***xxx* sets the left margin of the file being compiled. Text to the left of column *xxx* (an integer) is ignored. See also option **-Hrm** (enable right margin settings).

-Hmake[=*name*] — Write a list of makefile dependencies to standard output

Option **-Hmake** directs the compiler to generate a list of dependencies that can be used to create a makefile, and to redirect the list of such files to standard output. The list contains the names of files included in the source file, as well as the name of the source file itself.

If you specify the optional parameter *name*, each source file listed is prefixed with `"$(name)/"`. This feature permits source files to be in separate directories from the generated object files. Related controls are options **-Hmakeof**, **-Hrel**, and **-Humake** and toggle `Normalize_file_names`.

> **NOTE** **-Hiw** and **-Hmake** are mutually exclusive.

-Hmakeof=*filename* — Write a list of makefile dependencies to the named file

Option **-Hmakeof** directs the compiler to generate a list of dependencies that can be used to create a makefile, and to redirect the list of such files to the named file. The list contains the names of files

included in the source file, as well as the name of the source file itself. To get output dependency files alone (without generating object files) use **-Hmakeof** in combination with **-Hmake**.

Related controls are options **-Hmake**, **-Hrel**, and **-Humake**, and toggle `Normalize_file_names`.

Option **-Hmakeof** can take a format string in place of a specific *filename*. For more information, see Format Strings for Output-File Names on page 84.

-Hmerge — Concatenate intermediate source

Option **-Hmerge** gives the optimizer and code generator direct access to all of the source files being compiled. It is similar to wide inlining (**-Hiw**), but is more powerful as it concatenates the intermediate language for all of the source files and passes it to the optimizer as one file containing all code and data (the other generated object files are left empty).

To improve optimization, use this option with toggle `Treat_exports_as_locals` to make all the code and data in multiple source files available to the optimizer at one time.

NOTE This option is useful only when compiling all of the necessary source files at once, for example:

```
mcc *.c -Hmerge -Hon=treat_exports_as_locals
```

-Hmscerr — Use Microsoft error message format

Option **-Hmscerr** directs the compiler to format error and warning messages according to the Microsoft error-message format. Related options are **-Hefmat**, **-Hturboerr**, **-Hunixerr**, and **-Hwfmat**. For more information, see Changing the Message Format on page 337.

-Hnalib — Allow extra-ANSI function calls without leading underscore

Option **-Hnalib** allows extra-ANSI functions to be called without a leading underscore on the function name. For option **-Hnalib** to have effect, you must **#include** the supplied header files that contain the declarations of the extra-ANSI functions you want to call without the leading underscore. You must also link with the supplied (ANSI) library. See the *MetaWare C Library Reference* for the ANSI status of individual functions.

-Hnoarcexlib — Do not include bin/arcexlib.s

Most extension options include `bin/arcexlib.s` to define the associated extension instructions. If you want to use your own version of `arcexlib.s`, specify option **-Hnoarcexlib** in addition to your extension options. For **-Xsimd**, the corresponding option is **-Hnoarcsimd**.

-Hnoarcsimd — Do not include bin/arcsimd.s

By default, options **-Xsimd** and **-Xvraptor** include `bin/arcsimd.s` to define the SIMD instructions. If you want to use your own version of `arcsimd.s`, specify option **-Hnoarcsimd** in addition to **-Xsimd** or **-Xvraptor**.

-Hnocomdat — Disable the COMDAT solution for template instantiation

For C++ Only

Option **-Hnocomdat** disables COMDAT support. (Compare to option **-Hcomdat**, which has the opposite effect.)

The COMDAT solution is the default solution for template instantiation and virtual function table generation; that is, option **-Hcomdat** is specified by default.

Option **-Hnocomdat** turns off these toggles:

- `Inline_common`
- `RTTI_common`
- `Template_common`
- `Vtable_common`

A related toggle, which is off by default, is `Class_common_always`.

If you link with a linker that does not support COMDAT sections, you must specify option **-Hnocomdat** to disable COMDAT support, and adopt one of the manual instantiation techniques described in the *MetaWare C/C++ Language Reference*.

-Hnocopyr — Do not display copyright message

Option **-Hnocopyr** suppresses the display of the copyright message at compile time. The message includes a copyright statement, the compiler version number, and the serial number. By default, the copyright message is displayed when you invoke the compiler.

-Hnocplus — Do not include C++ .h files or run-time library

Option **-Hnocplus** instructs the compiler not to include C++ header files or the C++ run-time library when a program is compiled and linked. By default, the compiler uses option **-Hcplus**, which has the opposite effect.

If you are writing applications in C only, you can slightly increase the efficiency of your compilations by using this option.

For details about compiling C++ programs, see Compiling and Linking C++ Modules on page 30.

-Hnocpp — Use the inboard macro preprocessor

Option **-Hnocpp** directs the compiler to use the inboard macro preprocessor supplied with the compiler, rather than the outboard preprocessor supplied with the system. Compare to option **-Hcpp**, which has the opposite effect. For more information, see Using the Preprocessor on page 35.

-Hnocrt — Do not link in supplied run-time startup files

Option **-Hnocrt** suppresses the inclusion of the supplied `crt*.o` startup files during the link phase. Option **-Hnocrt** is off by default.

Use option **-Hnocrt** only when developing your own startup file(s) — using the supplied `crt1.s` as a starting point — or if your application runs on an operating system that does not need the services provided by the `crt*.o` files.

See Chapter 10 — Embedded Development for information about `crt*.o` files for embedded applications.

-Hnoinline — Do not invoke the outboard inliner, regardless of optimization level

Option **-Hnoinline** directs the compiler to skip the inlining optimization pass. Specifying **-Hnoinline** guarantees that code will not be inlined, even if the optimization level is 4 or higher. (By default, the compiler invokes the outboard inliner at optimization levels 4, 5, 6, 7, and s.)

Option **-Hnoinline** has no effect on the built-in C++ inliner. Also, if you specify option **-Hnoinline** and option **-Hi** together on the command line, option **-Hnoinline** takes precedence.

For information about the outboard inliner and the built-in C++ inliner, see Chapter 8 — Inlining.

-Hnolib — Do not link in MetaWare library

Option **-Hnolib** suppresses the inclusion of the MetaWare Run-Time Library during the link phase. Option **-Hnolib** is off by default.

-Hnonexcept_wrap=*filename* — Wrap specified include file to compile without exception handling

For C++ Only

Option **-Hnonexcept_wrap** directs the compiler to wrap the specified include file (those include files that are not exception-aware, such as `iostream.h`) with header files `exc_off.h` and `exc_pop.h`.

The wrapper files turn off toggle `Exception_aware_class` (so the specified include file compiles as non-exception-aware), and restore toggle `Exception_aware_class` to its previous setting at the end of the file.

See the *MetaWare C/C++ Language Reference* for details about C++ exception handling.

-Hnoobject — Do not generate object code or invoke the linker

Option **-Hnoobject** prevents the compiler from generating object code, and prevents the driver from calling the linker. With this option, you can check for compilation errors without the expense of generating object code.

-Hnopro — Do not read a .pro file

Option **-Hnopro** suppresses reading a `.pro` file. Compare to option **-Hpro**, which directs the compiler to read a `.pro` file before source files are read.

If you specify neither **-Hnopro** nor option **-Hpro**, the compiler reads the first file named `hc.pro` found on the directory search path. For more information, see Using .pro Files on page 39.

-Hnosdata — Generate code without use of a small-data area

Option **-Hnosdata** directs the compiler to generate code without use of a small-data area. The reserved small-data base register `%gp` is used as a general-purpose register.

NOTE Code compiled with option **-Hnosdata** specified is not compatible with code compiled without this option.

These are related compiler controls:

- options **-Hall_sdata** and **-Hsdata0**
- pragmas `Push_small_data` and `Pop_small_data`
- toggles `Unsized_arrays_in_small_data`, `Use_sdata`, and `Common_in_small_data`

Related Topic

- Using Small-Data Sections on page 214

Option **-Hnosdata** directs the compiler to generate code without attempting to place data in small-data sections for near addressing. Compare to option **-Hsdata0**, which has the opposite effect.

-Hnothread — Compile and link single-threaded programs

The standard MetaWare Run-Time Libraries are named `libct.a` and `libcct.a` for C and C++, respectively (the ‘t’ denotes thread safety). Option **-Hnothread** turns off thread safety by attempting to link single-threaded versions of the libraries (without the ‘t’). These versions are not provided, but you

can build them with option **-Hnothread** using the makefiles in the library source hierarchy. For more information, see *MetaWare Development Toolkit Getting Started*.

-Hobjdir=*path* — Specify an object-file directory

Option **-Hobjdir** specifies a path to an existing directory where object files and other output files are placed. You must create the storage directory before you can use it with **-Hobjdir**.

NOTE On Windows hosts, *path* can include a drive.

-Hoff=*Toggle_name* — Turn off a toggle

Option **-Hoff** turns the specified toggle off. For information about individual toggles, see Chapter 5 — Using Compiler Toggles.

-Hoffwarn=*n1*, *n2*, … — Disable specified warning messages

Option **-Hoffwarn** disables the specified compiler warning messages so they are not displayed. The arguments *n1*, *n2*, … are the numbers of the warning messages. This option is equivalent to pragma `Offwarn`.

-Hon=*Toggle_name* — Turn on a toggle

Option **-Hon** turns the specified toggle on. For information about individual toggles, see Chapter 5 — Using Compiler Toggles.

-Hpcc — Place the compiler in PCC mode to relax some ANSI C Standard restrictions

Option **-Hpcc** places the compiler in PCC mode, in which the compiler relaxes enough ANSI C Standard restrictions to emulate, to some degree, the AT&T Portable C Compiler. This mode permits old C programs to be compiled with little, if any, modification.

Specifically, option **-Hpcc** does the following:

- Turns on toggle `PCC`.
- Unreserves the keywords **signed**, **volatile**, and **pragma**; keyword **const** remains reserved because it is used in the library header files.
- Turns on toggle `Long_enums` so that **enum** types are mapped to full words, as is the PCC convention.
- Turns on toggle `Multiple_var_defs`.

Option **-Hpcc** also undefines the **__STDC__** predefined macro (see Using Predefined Macros on page 36).

-Hpfnoflt — Link in pfnoflt object file

Option **-Hpfnoflt** directs the linker to link in the pfnoflt object file, which eliminates the floating-point code used by the **printf()** family of functions. This reduces the size of a program.

Use option **-Hpfnoflt** only if your program is already using the **printf()** functions and does not use floating-point conversions.

-Hpictable — Generate position-independent code and data

Option **-Hpictable** directs the linker to generate a fixup table (`.pictable`) for relocating address constants so that code and data can be loaded independently at arbitrary addresses. It also links in the appropriate startup modules.

If you are invoking the linker without the driver, the corresponding linker option is **-Bpictable** (documented in the *ELF Linker and Utilities User's Guide*).

See Using Position-Independent Code and Data on page 244 for more information.

-Hpragma=*pragma_name*(*pragma_argument* ...) — Invoke a pragma on the command line

Option **-Hpragma** invokes the pragma named *pragma_name*. The effect is the same as executing the directive **#pragma** ***pragma_name*** in a source file. For more information about pragmas in general, see Chapter 6 — Using Compiler Pragmas.

For UNIX and Linux Hosts

Pragma names usually require parentheses to enclose their arguments. Because parentheses have special meaning to the shell, you must either precede parentheses with a backslash, or enclose the option or pragma in quotation marks. For example:

```
hcarc -Hpragma=Warning_level\(3\) hello.c
hcarc "-Hpragma=Warning_level(3)" hello.c
hcarc -Hpragma="Warning_level(3)" hello.c
```

> **NOTE** If you are invoking the driver command from a shell script, be sure to escape the backslash (\) and quotation-mark (") characters appropriately.

For Windows Hosts

You cannot use quotation marks within pragmas on the command line. However, if you are running an alternate command processor, you might need to enclose the whole option in quotes so that it is tokenized correctly:

```
hcarc "-Hpragma=Warning_level(3)" hello.c
```

-Hpro=*file* — Specify a .pro file

Option **-Hpro** specifies a `.pro` file (a file that the compiler reads before reading source files). If you specify neither **-Hpro** nor option **-Hnopro**, the compiler reads the first file named `hc.pro` found on the directory search path. For more information, see Using .pro Files on page 39.

> **NOTE** The compiler does not issue a warning if it does not find any `hc.pro` file.

-Hprofile — Instrument executable to display run-time profile information at exit

> **CAUTION** Profiling using option **-Hprofile** is not thread-safe. Using this profiler with a program designed to be thread safe is likely to produce erroneous profile results and might cause the program to fail. For information about a thread-safe alternative, see options **-p** and **-pg**, and Execution Profiling on page 293.

Option **-Hprofile** causes the compiler to generate calls to functions that time-profile each function call and build a profile tree. This profile tree is processed by the profiling functions called from **_atexit()** (just before program termination).

To produce a report, you must both compile and link with option **-Hprofile**. If you compile (but do not link) with option **-Hprofile**, the program runs normally but the does not produce a summary report at termination.

To profile using a timer extension, as on the ARCangel development board or equivalent, you must also link using option **-Xtimer0** or **-Xtimer1**. For more information, see Timer Extension Support on page 242.

-Hreg_remove=*reg*[*s*] — Remove a register from the processor

Option **-Hreg_remove** tells the compiler that the register (or registers) you specify in argument *reg*[*s*] is not available. The register then cannot be used for any purpose.

Argument *reg*[*s*] is a comma-separated list of register numbers and/or ranges, for example **-Hreg_remove**=`20,22-24`.

NOTE	The compiler might not permit a register to be removed if it deems that register essential.

CAUTION	If you use an option or pragma to remove a core register, you must compile your entire application with the same option or pragma. This includes the C run-time libraries. For information on rebuilding the libraries, see Rebuilding the C Run-Time Libraries on page 241.

To prevent the compiler from using a register and reserve the register for your own use, use option **-Hreg_reserve**.

Related Topics

- Pragma Core_register
- Pragma Reg_remove

-Hreg_reserve=*reg*[*s*] — Reserve a register for application-specific use

Option **-Hreg_reserve** reserves the register (or registers) you specify in argument *reg*[*s*]. Argument *reg*[*s*] is a comma-separated list of register numbers and/or ranges, for example **-Hreg_reserve**=`20,22-24`.

NOTE	The compiler might not permit a register to be reserved if it deems that register essential.

When it reserves the register, the compiler never generates code that uses that register. You may then use the register freely for application-specific purposes.

CAUTION	If you use an option or pragma to reserve a core register, you must compile your entire application with the same option or pragma. This includes the C run-time libraries. For information on rebuilding the libraries, see Rebuilding the C Run-Time Libraries on page 241.

CAUTION	Do not confuse this option with **-Hreg_remove**, which removes the register from any use.

Related Topics

- Pragma Core_register
- Pragma Reg_reserve

-Hrel — Omit absolute include files from dependency entries

Option **-Hrel** directs the compiler to omit absolute include files when the **-Hmake**, **-Hmakeof**, or **-Humake** option produces dependency entries. Option **-Hrel** also directs the compiler to ignore any absolute path names that start with "/" or "\".

An absolute pathname is any of the following:

- any include-file name enclosed in angle brackets ("<" and ">")
- any Windows path name beginning with *x*`:` (where *x* is the letter indicating a drive)
- any path name beginning with "/" (UNIX and Linux hosts)
- any path name beginning with "\" (Windows hosts)

For example, each of the file names specified in the following **#include** directives is an absolute pathname:

```
#include <stdio.h>
#include "c:\myfile.c"
#include "/dirname/myfile.c"
#include "\dirname\myfile.c"
```

-Hretwc — Add the warning count to the compiler's return code

Normally, the compiler returns the number of errors (but not warnings) in its return code. Option **-Hretwc** is provided so that, if you must have a completely clean compile with no warnings, you can check for success with a shell script or a batch file. For sample shell scripts and batch files, see Checking the Compiler Return Code on page 340.

NOTE When you specify option **-Hretwc**, compiler warnings take on the importance of errors. With this option specified, the linker will not be invoked if there are errors or warnings.

-Hrm=*xxx* — Enable right margin setting

Option **-Hrm** enables right margin settings; specifying **-Hrm**=*xxx* sets the right margin of the file being compiled at *xxx*, an integer. Text to the right of column *xxx* is ignored. This might be appropriate for files downloaded from 370 CMS or MVS systems; such files can have sequence numbers in columns 73 through 80, so **-Hrm**=`72` instructs the compiler to ignore those columns. See also option **-Hlm** (enable left margin setting).

-Hrtldir=*dir* — Specify an alternate run-time library directory

By default, the linker links libraries based on the processor series, endianness, and other configuration options you specify, using the directories below *install_dir*`/lib`. Use option **-Hrtldir** to specify an alternate directory, such as an additional library directory provided on your distribution or one you create.

When you specify option **-Hrtldir**, the linker searches for *dir* relative to *install_dir*`/lib` and then relative to the current directory, unless *dir* is an absolute path. For more information on rebuilding libraries and creating alternate library directories, see Rebuilding the C Run-Time Libraries on page 241.

-Hsdata0 — Generate code without use of a small-data area

Option **-Hsdata0** directs the compiler to generate code without using small-data sections but respecting the designation of register `%gp` as reserved so it may be used by other modules as the small-data base register. Compare to option **-Hnosdata**, which has a similar but different effect.

Code compiled with **-Hsdata0** specified is compatible with code compiled without this option — provided that the code compiled with option **-Hsdata0** does not define data that will be referenced as small data from other modules.

Related Compiler Controls

- options **-Hall_sdata** and **-Hnosdata**
- pragmas `Push_small_data` and `Pop_small_data`
- toggles `Unsized_arrays_in_small_data`, `Use_sdata`, and `Common_in_small_data`

Related Topic

- Using Small-Data Sections on page 214

-Hsfnoflt — Link in sfnoflt object file

Option **-Hsfnoflt** directs the linker to link in the sfnoflt object file, which eliminates the floating-point code used by the **scanf()** family of functions. This reduces the size of a program.

Use option **-Hsfnoflt** only if your program is already using the **scanf()** functions and does not use floating point conversions.

-Hsilent — Suppress screen output

Option **-Hsilent** inhibits the printing of output such as copyright messages by the compiler and assembler.

-Hstack=*size* — Specify the stack size

Option **-Hstack** overrides the default stack size of 64 K with the value specified in *size*.

-Hsuffix=.*suffix* — Specify the object file extension

Option **-Hsuffix** directs the compiler to produce an object file with the file-name extension `.`*suffix*.

The default file-name extension is `.o`.

-Hthread — Compile and link thread-safe programs

Option **-Hthread** defines preprocessor macros **_MT** and **_REENTRANT** and ensures that your program uses the thread-safe versions of the MetaWare Run-Time Libraries. Use option **-Hthread** when you are compiling and linking a program that uses multiple threads. Compare to option **-Hnothread**, which has the opposite effect.

> **NOTE** To generate code for a multi-threaded application, you must both compile and link with option **-Hthread**.

Option **-Hthread** is specified by default, and the MetaWare Run-Time Libraries are thread-safe by default. Non-thread-safe versions of the libraries are not provided, but can be built with option **-Hnothread** using the makefiles in the library source hierarchy.

CAUTION Do not confuse toggle `Thread` with the **-Hthread** option. Toggle `Thread` is meant to be applied at the source-code level, while option **-Hthread** is meant to be applied to an entire compilation.

If you apply toggle `Thread` to an entire compilation (using **-Hon**=`Thread`), the compiler tries to make every declaration in the source file into a **__declspec**`(thread)` declaration, which is erroneous in most cases.

-Htmpprefix=*temp_dir* — Set a directory for temporary files

Option **-Htmpprefix** specifies an alternate directory for temporary files required by the compiler. The directory you specify in *temp_dir* supersedes the setting of the environment variable `TMPPREFIX`. Ordinarily, the compiler creates temporary files in a default directory — typically `/tmp` or `/var/tmp` on UNIX or Linux hosts, or the current working directory on Windows hosts.

For information about setting `TMPPREFIX` directly, see Modifying Configuration Variables on page 310 and Setting an Alternate Directory for Temporary Files on page 35.

-Hturboerr — Use Borland error message format

Option **-Hturboerr** directs the compiler to format error and warning messages in the style of Borland error messages. Related options are **-Hefmat**, **-Hmscerr**, **-Hunixerr**, and **-Hwfmat**. For more information, see Changing the Message Format on page 337.

-Humake[=*macro*] — Write makefile dependencies to a .u file

Option **-Humake** directs the compiler to generate a list of dependencies that can be used to create a makefile, and to write the list to a file with the extension `.u`. The `.u` file is created in the source-file directory. The `.u` file contains a list of files included in the source file, as well as the name of the source file itself. These entries are the dependent components for the source file. If you define the optional parameter *macro*, its expansion is prefixed to each entry in the `.u` file.

Related controls are options **-Hmake**, **-Hmakeof**, and **-Hrel** and toggle `Normalize_file_names`. For more information, see Format Strings for Output-File Names on page 84.

-Hunixerr — Use AT&T Portable C Compiler error message format

Option **-Hunixerr** directs the compiler to format error and warning messages in the style of AT&T Portable C Compiler error messages. Related options are **-Hefmat**, **-Hmscerr**, **-Hturboerr**, and **-Hwfmat**. For more information, see Changing the Message Format on page 337.

-Hunroll=*Loop_count*,*Size* — Replace loops with loop-body code

Option **-Hunroll** directs the compiler to replace a loop with the code that makes up the body of the loop.

- *Loop_count* is the maximum number of times a loop is unrolled.
- *Size* is the maximum size of loops to unroll (approximate number of instructions).

You can use option **-Hunroll** multiple times with different loop parameters.

By default, option **-Hunroll** is specified at optimization level 7 (option **-O7**) and is not available at any other optimization level. Using this option does not guarantee unrolling of the targeted loops.

CAUTION Exercise caution when using this option. The ARC processor's zero-delay-loop mechanism generally gives better results, unless your code makes numerous memory

references. Best practice is to build with and without this optimization and compare size and performance.

-Husebp=*bpfile* — Recompile the application using branch-profiling data

Option **-Husebp** recompiles the application using the branch profiling data in *bpfile*. (Use option **-Hgenbp_data** to instrument the application with code to generate this data.) The profiling data helps the compiler decide the following for each branch:

- whether the given branch was visited during execution
- whether one target was executed more often than the other

For more information, see Branch Prediction and Branch Profiling on page 174.

-Hv4compat — Revert to version 4.x settings

For ARCtangent-A4 Only

Option **-Hv4compat** reverts to toolset version 4.*x* defaults. Specifically, it does the following:

- Turns on toggle Multiple_var_defs.
- Turns off toggles Uncached_in_own_section and Align_uncached_section.
- Reverts to DWARF-1 debugging format.
- Reverts to a default optimization level of **-O***1*.
- Reverts to the default core version **-core5**. However, it is best to specify your actual core version when using this option because **-core*** options for later core versions generate better code.

-Hwcsize=*value* — Specify the size of type wchar_t

Option **-Hwcsize** specifies the size of wide characters (represented as type **wchar_t**) to be *value*. To build Unicode applications (that is, applications using a two-byte character set), if the size of type **wchar_t** is not already 2 by default, compile with **-Hwcsize**=2 specified. For more information, see the Unicode appendix of the *MetaWare C Library Reference*.

-Hwfmat=*format* — Set the format of warning messages

Option **-Hwfmat** directs the compiler to format warning messages according to *format*, a quoted string containing a sequence of characters with special meanings. Related options are **-Hefmat**, **-Hmscerr**, **-Hturboerr**, and **-Hunixerr**. For more information, see Changing the Message Format on page 337.

-*Idir* — Specify an alternate directory to search for include files

Option **-I** specifies an alternate directory to search for an include file. You can specify this option more than once to name multiple directories to be searched. If a particular file is not found in the specified directories, the compiler also searches one or more standard directories.

A related option is **-Hipname**, which establishes the name of the environment variable to be used for specifying a directory search path for include files.

For information about another related option, see the listing for **-YI**.

-Lpaths — Specify paths to search for libraries specified with option -l

By default, the driver links the appropriate and necessary C++, C, and operating-system libraries from the current working directory and MetaWare C Library directory. Use option **-L** to specify other

directories that the linker should search for an additional library, which you name by specifying option **-l***name* after **-L***paths*.

paths is a list of directories separated by colons (:) on UNIX and Linux hosts, or by semicolons (;) on Windows hosts.

For more information about this linker option, see the *ELF Linker and Utilities User's Guide*.

Option **-YP** specifies default directories to be used for locating libraries.

-l*name* — Specify a library to include in the link

Option **-l** specifies that one of the following libraries (depending on the target operating system) is to be included in a link:

- `lib`*name*`.a`
- *name*`.lib`

You can specify option **-l***name* multiple times for multiple libraries. The driver searches for the library in these directories, in the order given:

1. your current working directory
2. any directories specified with the **-L***paths* option
3. the `lib` subdirectory of your distribution

If the driver does not find the library in any of these locations, it omits the library from the link and issues a warning message.

For more information about this linker option, see the *ELF Linker and Utilities User's Guide*.

-M*model* — Specify far data, near or far code

Option **-M***model* produces code with function calls designated as near or far, depending on the specified memory model (large, big, medium, or small) and the keywords used.

On ARCtangent-A4 and previous, near calls use the **BLcc** instruction (a four-byte instruction with a range of plus or minus 2 M from the current location); far calls use the **JLcc** instruction (eight-byte instruction with an absolute range of 0 to 64 M).

On ARCtangent-A5 and later, near calls use either the unconditional **BLcc** instruction (a four-byte instruction with a range of plus or minus 16 M from the current location) or the conditional **BLcc** instruction (a four-byte instruction with a range of plus or minus 1 M from the current location). Far calls use the **JLcc** instruction (eight-byte instruction with an absolute range of a full 32-bit address, 0 to 4 G).

If you attach the keyword **_Far** to an **extern** function declaration, the compiler generates far calls when calling the function.

If you attach the keyword **_Near** to an **extern** function declaration, the compiler generates near calls when calling the function.

Argument *model* can be *l*, *b*, *m*, or *s* for large, big, medium, or small, respectively. For ARCtangent-A4 and previous, *b* is the same as *m*.

Option **-Mm** is specified by default.

- Option **-Ml** directs the compiler to generate code in which calls are far by default, but **_Near** is respected. Calls to local static functions remain near. This option has the same effect as having toggles `Default_far` and `Respect_far` both on.
- For ARCtangent-A4 and previous, option **-Mb** is the same as option **-Mm**; for ARCtangent-A5 and later, option **-Mb** directs the compiler to generate code in which calls are near by default, but **_Far** is respected and no conditional calls are emitted. This option has the same effect as having toggle `Default_far` off and toggle `Respect_far` on while turning toggle `Conditional_calls` off.
- Option **-Mm** directs the compiler to generate code in which calls are near by default, but **_Far** is respected. This option has the same effect as having toggle `Default_far` off and toggle `Respect_far` on.
- Option **-Ms** directs the compiler to generate code in which calls are always near. The **_Far** keyword is ignored. This option has the same effect as having toggles `Default_far` and `Respect_far` both off.

-o *output_file* — Specify name of executable file

Option **-o** specifies the file name of the executable application generated by the compiler and the linker. Whitespace is required between the **-o** and the output file name.

- If you specify option **-c** and only one source file is being compiled, *output_file* is the name of the object file to be produced. By default, the name of the object file is derived from the name of the source file.
- If you specify option **-S** and only one source file is being compiled, *output_file* is the name of the assembly file to be produced. By default, the name of the assembly file is derived from the name of the source file.

The default *output_file* is `a.out`.

-O[*n*] — Set the optimization level

Option **-O*n*** sets the optimization level to *n*, where *n* is a number ranging from 0 (zero) through 7, or the letter ‘s’ (for generating smaller code).

- If you specify **-O** without an argument, level 6 is the default
- If you do not specify **-O** level 6 is the default, unless you specify option **-g**. When option **-g** is specified, level 1 is the default.

CAUTION Optimization level 7 inhibits the generation of zero-delay loops in favor of loop unrolling. The zero-delay loops built into ARC processors generally give better results, unless your code makes numerous memory references.

NOTE For easier debugging, we recommend that you not specify option **-O** when you use option **-g**. You can use option **-g** with any optimization level, but debugging is more difficult for code optimized at levels higher than level 1.

-p — Generate code to count calls to each function (enable statistical profiling)

Option **-p** generates code that counts the number of times each function is called during execution. When you invoke the linker, it includes run-time support for profiling in the executable.

This option defines preprocessor macro **__PPROFILE__**.

NOTE If you intend for your executable to run on an ARCangel™ development board or equivalent, you must also specify option **-Xtimer0** or **-Xtimer1** to link in support for the appropriate system timer extension module (if you specify both modules, the one specified last is included in the link). If you do not specify a timer, the linker issues a warning.

Run-time I/O support is required to write the output data file. The default hostlink run-time support provides the required I/O calls. For more information, see Providing System Calls for Debugging from a Host on page 229.

CAUTION Executables compiled using option **-p** might require a sizeable heap — between 50% and 100% of the size of the executable code, depending on the granularity and range of the sampling. For more information, see Using a Heap on page 213 and Providing Support for Statistical Profiling on page 242.

When you run the executable linked with **-p**, it generates a `mon.out` file with profiling information.

You can view the statistical profile using any of the following:

- the MetaWare debugger
- the GUI-based integrated profiling tool from ARC International
- the command-line utility **profarc** (**profac** for ARCtangent-A5 and later)

For more information, see Execution Profiling on page 293.

-P — Preprocess source file but do not compile; write output to a file

Option **-P** directs the driver to process the specified source file but not to compile it. Instead, the preprocessed result is written to a file with the same name as the source file, but with the file-name extension `.i` (or `.ii`). Compare to option **-E**, which also preprocesses without compiling but writes the result to standard output.

NOTE For C++ source files, the output extension is `.ii`. However, the system preprocessor you specify with **-Hcpp** might not support the `.ii` file-name extension.

For UNIX and Linux Hosts Only

If you use option **-Hcpp** in conjunction with option **-P**, the driver passes the **-P** option to the external (system) preprocessor **cpp** (see man page cpp(1)). For more information, see Using the Preprocessor on page 35.

-pg — Generate code to count calls to each function (enable statistical profiling with call graphing)

Like option **-p**, option **-pg** generates code that counts the number of times each function is called during execution. When you invoke the linker, it includes run-time support for profiling in the executable.

This option defines preprocessor macro **__PPROFILE__**.

NOTE You must both compile and link with option **-pg** to enable call graphing.

Run-time I/O support is required to write the output data file. The default hostlink run-time support provides the required I/O calls. For more information, see Providing System Calls for Debugging from a Host on page 229.

Run-time I/O support is required to write the output data file. The default hostlink run-time support provides the required I/O calls. For more information, see Providing System Calls for Debugging from a Host on page 229.

NOTE If you intend for your executable to run on an ARCangel™ development board or equivalent, you must also specify option **-Xtimer0** or **-Xtimer1** to link in support for the appropriate system timer extension module (if you specify both modules, the one specified last is included in the link). If you do not specify a timer, the linker issues a warning.

CAUTION Executables compiled using option **-pg** might require a sizeable heap — between 50% and 100% of the size of the executable code, depending on the granularity and range of the sampling. For more information, see Using a Heap on page 213 and Providing Support for Statistical Profiling on page 242.

The executable linked with **-pg** generates a `gmon.out` file. You can view the statistical profile using any of the following:

- the MetaWare debugger
- the GUI-based integrated profiling tool from ARC International
- the command-line utility **profarc** (**profac** for ARCtangent-A5 and later)

For more information, see Execution Profiling on page 293.

-S — Produce an assembly source file instead of an object file

For each source file, option **-S** directs the compiler to produce an assembly-language source file, instead of an object file.

When you specify option **-S**, the compiler writes the assembly source to a file with the same name as the C or C++ source, with the source file extension replaced by `.s`. The file is placed in the current working directory. The compiler does not generate an object file, and does not invoke the linker.

Use the **-o** option to specify an alternate name for the assembly file. If you specify **-Hanno** with **-S**, the compiler annotates the assembly file with lines from the source file.

-U*name* — Undefine a symbol (macro name)

Option **-U** removes any initial definition of macro *name*. See option **-D**.

-V — Display controls passed to the compiler; also display subprocesses

Option **-V** displays the name of each toggle and option being passed to the compiler, and each subprocess being executed.

-w*n* — Set the compiler diagnostic warning level

Option **-w***n* sets the compiler diagnostic warning level to *n*.

Warning levels are numbered from 0 (zero) to 4, according to the likely importance of the warnings. (Lower-level warnings are likely to be more important than higher-level warnings.) Appendix F — Diagnostic Messages explains in detail the process of managing diagnostic warning levels.

-X{a|c|m|t} — Specify the degree of conformance to the ANSI C Standard

The **-X** compiler options let you specify the degree of conformance to the ANSI C Standard by designating the mode in which the compiler runs. There are four modes:

- ANSI mode (**-Xa**)
- conformance mode (**-Xc**)
- MetaWare mode (**-Xm**)
- transition mode (**-Xt**)

The default setting is **-Xm**.

Predefined macro **__STDC__** is set to 0 (zero) in **-Xa** and **-Xt** modes; it is set to 1 (one) in **-Xc** and **-Xm** modes.

-Xa — Compile in "ANSI mode"

ANSI mode is the same as transition mode (**-Xt**) except that the compiler decides in favor of ANSI Standard C when it encounters a construct with multiple interpretations. This means the `Use_UP_rules` toggle is turned off.

Despite its name, **-Xa** does not enforce strict conformance to ANSI Standard C. The **-Hansi** option specifies conformance to ANSI Standard C.

-Xc — Compile in "conformance mode"

Conformance mode enforces most ANSI C Standard restrictions and supports all MetaWare C extensions to the C language. Standard header files are compliant with the ANSI C Standard and only ANSI-defined identifiers are visible in the standard header files. In this mode, the toggle `VP_UP_warn` is initialized to off unless the warning level is set to 4.

-Xm — Compile in "MetaWare mode"

MetaWare mode supports all MetaWare C-specific extensions to the language as documented in the *MetaWare C/C++ Language Reference* manual.

-Xt — Compile in "transition mode"

Transition mode is provided for compiling pre-ANSI Standard C programs. In this mode, if the compiler encounters a language construct that is specific to pre-ANSI C, it uses the pre-ANSI C interpretation by turning on toggles `Use_UP_rules` and `VP_UP_warn`.

When you use the **-Xt** option, the compiler generally supports all ANSI C Standard features and MetaWare C extensions that are compatible with pre-ANSI Standard C. It also relaxes enough of the ANSI rules to allow most pre-ANSI programs to be compiled without modification. The compiler warns about any construct that is invalid under the ANSI C Standard but permitted under pre-ANSI C.

-YI,*dir* — Specify a directory to search last for include files

Option **-YI** specifies a directory to be searched last for include files. For more information, see option **-I**.

-YP,*dir1*[:*dir2* [:*dir3*...]] — Specify default directories for locating libraries

Option **-YP** specifies default directories to be used for locating libraries.

Format Strings for Output-File Names

In these command-line options that generate output files, you can substitute format strings for parts of the explicit file-name arguments:

-Hbatch

-Hefile

-Hlist

-Hmakeof

These format strings, listed in Table 5 Format Strings for Output File Names, cause certain characteristics of the input file to be the default for the generated output file.

Table 5 Format Strings for Output File Names

Format String	Output-File Characteristic
@%d	Output file has the same directory pathname and file name as the input file.
@%e	Output file has the same extension as the input file.
@%f	Output file has the same full name as the input file: drive, directory pathname, file name, and extension.
@%n	Output file has the same file name as the input file.
@%p	Output file has the same prefix as the input file (drive and directory pathname), but not the same file name or extension.

Example 1

This example command line generates `myfile.lst`, a list file with the same file name as the input file (*%n*), with the extension `.lst`:

```
hcarc -Hlist=@%n.lst myfile.cpp
```

Example 2

This example command line generates `myfile.xyz`, a batch file with the same file name as the input file (*%d*), with the extension `.xyz`, and puts it in the same directory as the input file:

```
hcarc -Hbatch=@%d.xyz myfile.cpp
```

Chapter 5 — Using Compiler Toggles

In This Chapter

- Overview
- Using Toggles

Overview

This chapter documents MetaWare C/C++ compiler toggles and describes how to use them. For information about individual toggles, see Alphabetical List of Compiler Toggles on page 86.

Using Toggles

Compiler *toggles* are switches that turn compiler controls on or off. You can specify toggles on the command-line, in which case they affect the entire program being compiled. Alternatively, you can use them in your source code to affect only certain portions of your code.

Windows Hosts Only
If you are using a version of the tools that includes an IDE, you can set and reset toggles in the IDE without using the driver or command line directly. For more information, see the IDE documentation.

Toggle names are not case-sensitive.

To set or reset a toggle from the command line, use option **-Hon** or **-Hoff** (see Driver Option Reference on page 42). For example:

```
hcarc -Hon=Toggle_name ...
hcarc -Hoff=Toggle_name ...
```

To set a toggle in the configuration file, you specify the default value with the `ARGS` variable (see Appendix B — Configuring the Driver). For example:

```
ARGS+=-Hon=Toggle_name
```

To set a toggle in source code or in a `.pro` file, you use a compiler pragma (see Chapter 6 — Using Compiler Pragmas). For example:

```
#pragma On(Toggle_name)
#pragma Off(Toggle_name)
#pragma Pop(Toggle_name)
```

Pragma `On` turns the toggle on; pragma `Off` turns it off; and pragma `Pop` reinstates it to a prior value. Toggles operate in a stack-like fashion, where each on or off is a “push” of the previous state, and a pop “pops” the stack.

```
#pragma On (List) /* Turns On source listing   */
#pragma Off(Cleanup_spills) /*Register spill code not cleaned up */
#pragma Off(List) /* Turns Off source listing  */
#pragma On (List) /* Turns On source listing   */
#pragma Pop(List) /* Back to Off for listing   */
#pragma Pop(List) /* Back to On for listing    */
```

Alphabetical List of Compiler Toggles

This section lists all compiler toggles in alphabetical order. For a quick-access list, see Quick Toggles List on page 351.

Align_branch_targets — Ensure word alignment where possible

Default: On if optimization level = 6 — 7

For ARCtangent-A5 and Later Only
When this toggle is on, the compiler aligns all branch targets to remove the delay of one clock cycle that occurs when a branch is taken to a target that is not word-aligned. The compiler aligns the label by

finding a previous 16-bit instruction that it can widen to a 32-bit form. If no 16-bit instruction can be found (very rare), the label is left as is.

Align_uncached_section — Pad uncached data section to multiple of 256

Toggle `Align_uncached_section` pads the section `.ucdata` to a multiple of 256 bytes (the size of the largest cache line envisioned). This padding ensures that the linker does not combine uncached data with other data on an address boundary that might overlap a cache line boundary.

Turn this toggle off if the extra padding is not necessary for your application or if you are using a linker command file that explicitly places your uncached data.

For more information on using uncached **volatile** and **_Uncached** data, see Working with Volatile and Uncached Variables on page 222.

Asm_percent_reg — Require '%' on register names

Default: On

When this toggle is on, registers used in inline assembly macros must be specified with '%', for example, `%r0`, `%r13`. This matches the default assembly output of the compiler and avoids conflicts with program symbols that happen to have names like `r0` or `r13`.

Turn off this toggle to eliminate the '%' requirement and specify registers as simply `r0` or `r13`, for example.

When this toggle is off, the compiler still emits registers with a '%' prefix in generated assembly, but you need not do so in inline assembly macros.

Related Topic

- Chapter 9 — Inline Assembly Code

Auto_class_member_instantiation — Enable automatic instantiation of class templates

Default: On

For C++ Only

In a C++ program, a class template must always be instantiated when it is used. By default, the compiler generates instances of class members when the class is used in a source module, and when the definitions of those members are presented to the compiler. To disable automatic instantiation, turn off toggle `Auto_class_member_instantiation`. Related toggle is `Auto_func_instantiation`.

> **NOTE** Toggle `Auto_class_member_instantiation` has no effect if you use the COMDAT solution. Option **-Hcomdat** enables the COMDAT solution; **-Hnocomdat** disables it.

For more information, see the following:

- toggle `Defer_template_defn`
- templates in general, and the COMDAT solution, in the *MetaWare C/C++ Language Reference*
- Common Superfluous Warnings on page 340

AutoDebug — Place named SDI for a named struct in the struct's debug repository

Default: Off

When toggle `AutoDebug` is on and you specify option **-g**, the compiler places named symbolic debugging information (SDI) in the debug repository of any named C or C++ **struct** (including unions

and classes) declared within the toggle's scope. The debug repository of a **struct** is the compilation unit in which the **struct**'s first non-pure-virtual, non-inline function is defined. Related toggles are `ForceDebug` and `NoDebug`.

Auto_func_instantiation — Enable automatic instantiation of function templates

Default: On

C++ Only

When you use the same template in separate source modules in a C++ program, the linker returns an error complaining of a duplicate definition. To prevent duplicate definitions, disable automatic template instantiation by turning off toggle `Auto_func_instantiation`. Related toggle is `Auto_class_member_instantiation`.

> **NOTE** Toggle `Auto_func_instantiation` has no effect if you use the COMDAT solution. Option **-Hcomdat** enables the COMDAT solution; option **-Hnocomdat** disables it.

For more information about templates in general, and the COMDAT solution, see the *MetaWare C/C++ Language Reference*.

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Auto_reuse_warnings — Warn if auto variable declared twice in a function

Default: Off

Toggle `Auto_reuse_warnings` causes a warning to be issued whenever an **auto** variable is declared in a nested block and is already declared as an **auto** or parameter in an outer block of the same function.

Behaved — Specify a "well-behaved" program

Default: Off if optimization level = 0 — 3
On if optimization level = 4 — 7, *s*, or *s1*

Toggle `Behaved`, when on, specifies that the program is "well-behaved" in its use of pointers, so the optimizer can be less conservative in generating code to reference pointer-based objects. *Well-behaved* code follows these rules:

- The address of a **union** member is never assigned to a pointer.
- A value of a pointer type is never cast to an incompatible pointer type.

Option **-g** has no effect on toggle `Behaved`.

Bool — Enable support for the C++ bool type

Default: On

For C++ Only

When toggle `Bool` is on, the C++ compiler supports the **bool** data type. You might with to turn on this toggle if your C++ level (3 or less) does not support **bool** by default.

See also option **-Hcpplvl**.

Borland — Relax some C++ standards to conform to the Borland class library

Default: Off

For C++ Only

In C++ compilations, toggle `Borland` relaxes some compile-time error checking, and ensures that the type of **sizeof** is the same as **size_t**.

The following operations do not conform to the *Annotated C++ Reference Manual*, but they are relaxed when this toggle is on because they appear in Borland class-library source:

Type of sizeof

The ANSI C Standard requires the type of **sizeof** to be the same type as **size_t**. However, **size_t** is **unsigned**, and code such as this does not work if the type of **sizeof** is **size_t**:

```
if (amount_left - sizeof(item_2_allocate)) > 0
```

In MetaWare C/C++, **sizeof** is **unsigned**.

Type of Certain Conditional Expressions

In a conditional expression such as `X?Y:Z` or `X?Z:Y`, the `Y` can be a **const** structure type `T`, and `Z` a non-**const** `T`; the result type is `T`.

Branch_reductions — Generate compare-and-branch instructions

Default: On

For ARCtangent-A5 and Later

Toggle `Branch_reductions` enables generation of the compare-and-branch instructions introduced in the ARCompact instruction set.

Early variants of the ARCtangent-A5 processor had a restriction such that breakpoint instructions could not be placed adjacent to compare-and-branch instructions. This can cause problems when statement stepping with source code, as the debugger might attempt to place breakpoint instructions in illegal locations. If you have a processor with such a restriction, turn off toggle `Branch_reductions` when generating debugging information (**-g**) for source-level debugging. Because this toggle is necessary only when compiling for source-level debugging, turn it back on after debugging.

Call_trace — Generate information about a function being called

Default: Off

When toggle `Call_trace` is on, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is called. Related toggles are `Epilog_trace` and `Prolog_trace`.

For more information, see Tracing Function Calls on page 292.

Calls_preserve_flags — Preserve condition-code flags when calling functions

Default: Off

When toggle `Calls_preserve_flags` is on, the compiler generates code to preserve the condition-code flags when calling functions. With the knowledge that flags are not destroyed by calls, the compiler can perform better conditional execution of calls.

CAUTION Use of toggle `Calls_preserve_flags` prevents an application from setting and clearing interrupt-mask bits, because the interrupt bits are in the same register as the condition-code flags.

ARCtangent-A5 and Later

CAUTION Use toggle `Calls_preserve_flags` sparingly; it introduces extra overhead to save and restore the `status32` register. In addition, assembly-language source in the MetaWare C Run-Time libraries do not generally preserve flags; expecting flags to be preserved when calling such a function might cause your application to fail.

CFA — Generate DWARF 2 CFA information for call-stack tracing

For ARCtangent-A5 and Later

Default: On

Toggle CFA directs the compiler to emit canonical frame address (CFA) information so that the MetaWare debugger can perform call-stack tracing.

Char_default_unsigned — Make default representation of char unsigned

Default: Target-operating-system specific; see Example 11 Program for Determining Default Character Representation Setting to determine your default **char** representation.

Type **char** has one of two representations: **signed** or **unsigned**. When the representation of **char** is not specified explicitly using the keyword **signed** or **unsigned**, the default representation is used.

The setting of this default representation depends on whether toggle `Char_default_unsigned` is on or off:

- When toggle `Char_default_unsigned` is on, type **char** is **unsigned** by default.
- When toggle `Char_default_unsigned` is off, type **char** is **signed** by default.

The default setting of toggle `Char_default_unsigned` is ordinarily on (type **char** is **unsigned**), but the toggle can be altered from the configuration file.

You can use the following program to determine your default **char** representation.

Example 11 Program for Determining Default Character Representation Setting

```
/*
 * Test program for determining the default setting
 * of toggle Char_default_unsigned */

main() {
   printf("%d\n", (char) - 1);
   }
```

If the test program prints `-1`, the default for the toggle is off and the default **char** representation is **signed**. If the program prints `255`, the default for the toggle is on and the default **char** representation is **unsigned**.

Regardless of the default representation, type **char** is still not identical to either type **unsigned char** or type **signed char**. The compiler issues a diagnostic message if, for example, the default representation of **char** is **unsigned** and you try to assign a variable of type **unsigned char*** to type **char***.

To recognize type **char** as being identical to the default representation chosen for **char**, use toggle `Char_is_rep`.

For general implementation details about type **char**, see Characters on page 248.

Char_is_rep — Recognize type char as being identical to the default representation chosen for char

Default: Off

> **CAUTION** Turning on toggle `Char_is_rep` might decrease program portability.

Type **char** has one of two representations: **signed** or **unsigned**. For example, when **char** is **unsigned**, 0x80 represents 128, but when **char** is **signed**, 0x80 represents -128.

According to the ANSI C Standard, **char**, **signed char**, and **unsigned char** are three distinct types. Therefore, type **char*** cannot be assigned to type **signed char*** or type **unsigned char***. The compiler issues a diagnostic message if you attempt such an assignment.

When you do not explicitly specify the representation of **char** using the keyword **signed** or **unsigned**, the compiler uses a default representation. This default representation of **char** depends on the compiler implementation. However, even though the default representation of **char** is either **signed** or **unsigned**, the compiler does not automatically recognize **char** as being the same type as **signed char** or **unsigned char**.

When toggle `Char_is_rep` is on, the compiler recognizes type **char** as being identical to the default representation chosen for **char** — either **unsigned char** or **signed char**. For example, if toggle `Char_is_rep` is on and the default representation of **char** is **unsigned char**, a variable declared as type **unsigned char*** can be assigned to type **char*** without the compiler issuing a diagnostic message.

- To explicitly make the default representation of **char** be **unsigned char**, turn on toggle `Char_default_unsigned`.
- To explicitly make the default representation of **char** be **signed char**, turn off toggle `Char_default_unsigned`.

For general implementation details about type **char**, see Characters on page 248.

> **NOTE** If you intend to use toggle `Char_is_rep`, be sure to turn it on before you make any declarations.

Check_function_returns — Issue warning if function does not return

Default: On

When toggle `Check_function_returns` is on, the compiler issues a warning when it determines that a function might exit without returning a value. This is a useful diagnostic for detecting programming errors.

Class_common_always — Always generate virtual function tables for a class

Default: Off

For C++ Only

Toggle `Class_common_always` is a component of the COMDAT solution to the single-copy problem. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.)

By default, the compiler places a copy of certain classes' virtual function tables in any compilation unit (usually: file) that defines any non-pure-virtual, non-inline function of the classes.

To dispense with the notion of a seat and always generate the virtual function tables of a class, turn off toggle `Class_common_ok`. and turn on toggle `Class_common_always`. The disadvantage of this strategy is that more copies of the virtual function tables appear, so object modules are larger (though memory use is not).

The following are related toggles:

- `Class_common_ok`
- `Defer_template_defn`
- `Inline_common`
- `RTTI_common`
- `Template_common`
- `Vtable_common`.

NOTE Option **-Hcomdat** is on by default.

Class_common_ok — Generate virtual function tables when functions in the class are defined

Default: On

For C++ Only

Toggle `Class_common_ok` is a component of the COMDAT solution to the single-copy problem. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.)

When toggle `Class_common_ok` is on, the compiler emits duplicates of virtual function tables for any compilation unit (usually: file) that defines any non-pure-virtual, non-inline function of a class.

This helps guarantee that every class will be seated properly, i.e., can be referenced in its compilation unit, and also means that the compiler emits duplicates of every class that contains only constructors or destructors. This increases the size of object files (though not memory use); for smaller object files, turn off toggle `Class_common_ok` to generate a single copy of the tables.

Alternatively, you can turn off toggle `Class_common_ok` and turn on toggle `Class_common_always` to emit copies in every file where the class is declared.

The following are related toggles:

- `Defer_template_defn`
- `Inline_common`
- `RTTI_common`
- `Template_common`
- `Vtable_common`.

NOTE Option **-Hcomdat** is on by default.

Cmp_zero_elim — Remove compares with zero
Default: Off if optimization level = 0 — 3
Default: On if optimization level = 4 — 7, s, s1

When toggle `Cmp_zero_elim` is on, the compiler attempts to remove compares with zero by allowing a prior instruction to set the flags.

Command_line_ident — Put information in a comment section in generated code
Default: On

When toggle `Command_line_ident` is on, the compiler inserts information into a comment section in the generated code. This information includes the compiler's version number and the settings of certain options. Generating this information is not necessary, so it is safe to turn the toggle off (doing so merely reduces the object-file size). For related information, see Placing Comments in Object Code on page 162.

Common_in_small_data — Assign common variable declarations to small data section
Default: On

When toggle `Common_in_small_data` is on, the compiler assigns common variable declarations to the small-data section, removing the common semantics.

For example, this declaration is normally common, which means the linker can resolve multiple object files containing declarations of `i`:

```
int i;
```

When toggle `Common_in_small_data` is on and small-data allocation is configured to include variables of **sizeof(int)**, such multiple declarations result in multiply defined symbols.

This result is not a problem in most applications, because header files contain such declarations with the **extern** keyword. However, to avoid linker errors for some applications, you might need to turn off toggle `Common_in_small_data`, eliminating the common data from the small-data section.

NOTE The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** into small data unless you qualify them as **_Near**.

Using Common_in_small_data with Multiple_var_defs
When toggle `Multiple_var_defs` is off (the default), declarations like **int** `i;` are not common. Instead, they are "hard" definitions and have the same behavior as if they were put in a small-data section — multiple definitions or occurrences cause link errors. If toggle `Multiple_var_defs` is off, you probably do not need toggle `Common_in_small_data`.

Related Compiler Controls
- pragmas `Push_small_data` and `Pop_small_data`
- toggles `Unsized_arrays_in_small_data` and `Use_sdata`

Related Topic
- Using Small-Data Sections on page 214

Compress_asm_warn — Warn on use of inline assembly macros
For ARCtangent-A5 and Later Only
Default: On

By default, the ARCtangent-A5 compiler warns when it encounters inline assembly macros, because they inhibit optimization. Turn this toggle off to inhibit such warnings.

Conditional_calls — Emit conditionally executed function calls

Default: On unless option **-Mb** is specified

For ARCtangent-A5 and Later

When toggle `Conditional_calls` is on, the compiler may emit conditionally executed function calls. Turn this toggle off to suppress emission of such calls, which have a reduced range from the current location compared to unconditional calls. For more information, see the listing for option **-M**.

Debugger_handles_reg_vars — Allocate variables to registers

Default: On

When toggle `Debugger_handles_reg_vars` is on, the compiler allocates variables to registers, even when compiling in debug mode (that is, with option **-g** specified).

Default_far — Generate _Far calls for all function calls not declared as _Near

Default: Off

When toggle `Default_far` is on, the compiler generates **_Far** calls for all function calls, unless you have explicitly declared them to be **_Near**.

See the following for more information:

- toggle `Respect_far`
- option **-M***model* (in Driver Option Reference on page 42)
- type qualifiers **_Far** and **_Near** in the *MetaWare C/C++ Language Reference*

Defer_template_defn — Defer the definition of a template class until its definition is needed

Default:

On if option **-Hcomdat** is specified
Off if option **-Hnocomdat** is specified

For C++ Only

When toggle `Defer_template_defn` is on, the compiler defers the definition of a template class until its definition is needed.

When the toggle is off, the compiler defines a template class when the template is instantiated. This toggle is off by default, because it affects other template programming styles adversely.

When toggle `Auto_class_member_instantiation` is on, you should turn off toggle `Defer_template_defn` in all compilation units but one, then turn on toggle `Defer_template_defn` in a single defining compilation unit (one that contains **typedef**s for all the templates used in the program).

When toggle `Defer_template_defn` is on, this strategy does not work, because the defining compilation unit defines nothing — all the definitions are deferred. Toggle `Defer_template_defn` is turned on when you use the COMDAT solution, because COMDAT obviates the need for toggle `Auto_class_member_instantiation`.

The following are related toggles:

- `Class_common_always`
- `Class_common_ok`
- Inline_common
- `RTTI_common`
- `Template_common`
- `Vtable_common`.

For information about the COMDAT solution and templates in general, see the *MetaWare C/C++ Language Reference.*

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Define_static_members — Supply implicit definition of static class data members

Default: On

For C++ Only

According to the C++ language definition, static data members of a class must be defined outside the class.

```
class s {
   static int x;   /* This is NOT a definition */
   };
int s::x;   /* This is the required definition */
```

Turning on toggle `Define_static_members` relaxes this restriction, and the definition is implicitly supplied by the compiler. The toggle accommodates some existing practices. For information about compiling C++ code, see Compiling and Linking C++ Modules on page 30.

Define_unreferenced_template_members — Define all template members regardless of whether they are referenced

Default: On

For C++ Only

When a program instantiates a class template that was declared or defined with toggle `Define_unreferenced_template_members` turned on, each member function is defined.

When a program instantiates a class template that was declared or defined with toggle `Define_unreferenced_template_members` turned off, the only member functions that are defined are those member functions that are referenced.

A related toggle is `Defer_template_defn`.

For more information about templates in general, see the *MetaWare C/C++ Language Reference*.

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Direct_ldst_optimize — Improve memory addressing

Default:

Off if optimization level = 0 — 3
On if optimization level = 4 — 7, s, s1

By default, the compiler references memory by first loading the address into a base register, and then using the base register in subsequent loads and stores.

This produces smaller, faster code if the base register is used multiple times. However, if the base register is used only once, it is more efficient to substitute the address directly into the load or store instruction.

When toggle `Direct_ldst_optimize` is on, the compiler attempts to perform the back-substitution of addresses directly into load and store operations in cases where the address is used only once.

Double_constants — Use double precision for float constants

Default: On

Toggle `Double_constants` causes the compiler to generate code involving **float** constants using double precision unless such constants are suffixed with 'f.' Turn off this toggle to use single precision when other operands are also single-precision (type **float**).

Double_math_only — Perform double-precision floating-point arithmetic

Default: Off

When toggle `Double_math_only` is on, floating-point operations are performed in double precision.

When two operands of certain arithmetic operations are both of type **float**, the ANSI C Standard permits an implementation to do one of two things:

- Perform the operation using **float** arithmetic, in which case the result of the operation is of type **float**.

  Or

- Convert both operands to type **double** and use **double** arithmetic, in which case the result of the operation is of type **double**.

When toggle `Double_math_only` is turned off, the compiler uses the first option. When the toggle is turned on, the second is used instead.

NOTE Option -fsingle turns off toggle `Double_math_only`. Option -fdouble turns on the toggle.

Double_return — Make non-prototype functions return type double, not type float

Default: Off

When toggle `Double_return` is on, any non-prototype function returning type **float** returns type **double** instead. This convention conforms to some Portable C Compiler implementations.

Downshift_file_names — Convert include file names to all lowercase

Default: Off

When toggle `Downshift_file_names` is on, the file-name specification of any subsequent **#include** directive is interpreted as if it were in all lowercase. Toggle `Downshift_file_names` is useful when moving source code from an operating system in which file-name casing is not significant to a system in which it is significant.

Dpfp_div_em — Use DPFP extension when emulating floating-point division

For ARC 600 and Later when using option -Xdpfp

Default: Off

Toggle `Dpfp_div_em` uses a faster floating-point divide routine employing the double-precision floating-point extension instructions. Calculations performed within the scope of this toggle might yield a slightly less accurate result, but we recommend using it when using option **-Xdpfp**.

Dsf_hoist_from_target — Fill branch delay slot by hoisting instruction from target

Default:

Off if optimization level = 0 — 5, s, or s1
On if optimization level = 6 — 7

When toggle `Dsf_hoist_from_target` is on, the compiler attempts to fill a branch delay slot, if necessary, by copying an instruction from the target of the branch into the delay slot and adjusting the target of the branch to branch around the hoisted instruction.

Each_function_in_own_section — Map each function to a separate section

Default: Off

Toggle `Each_function_in_own_section` directs the compiler to map each function into a section with the same name as the function.

> **NOTE** This toggle does not apply to overlay sections, which are mapped to overlay sections. See the *Automated Overlay Manager User's Guide* for details.

You can use this toggle in conjunction with linker option **-zpurge** to reduce code size by eliminating unreferenced functions. See the *ELF Linker and Utilities User's Guide* for more information about linker option **-zpurge**.

Related Topics

- *Automated Overlay Manager User's Guide*
- *ELF Linker and Utilities User's Guide*

Each_var_in_own_section — Map each variable to a separate section

Default: Off

Toggle `Each_var_in_own_section` directs the compiler to map each variable into a section with the same name as the variable.

You can use this toggle in conjunction with linker option **-zpurge** to reduce code size by eliminating unreferenced variables. See the *ELF Linker and Utilities User's Guide* for more information about linker option **-zpurge**.

Ecpp_error — Generate error message when detecting certain non-EC++ features in Embedded C++ code

Default: On

When using C++ in embedded applications, you can use driver option **-Hecpp** to disable certain features and keywords of the C++ language. See the listing for option **-Hecpp** in General Driver Options on page 52 for a complete list of non-EC++ features and keywords, along with the toggles that enable or disable them.

When toggle `Ecpp_error` is on, the compiler generates an error message if your code uses one of the disabled keywords or features.

When this toggle is off, the compiler generates a warning — not an error — when it detects a disabled keyword or feature, but it does not actually disable the item.

With this error-versus-warning capability, you can use the following combination of options to direct the compiler to warn when your C++ code uses any of the keywords or features that option **-Hecpp** disables:

```
-Hecpp -Hoff=Ecpp_error
```

Emit_core_directives — Emit core-register-extension directives

Default: On

When toggle `Emit_core_directives` is on, the compiler automatically emits assembler directives to define extension core registers, condition codes, and extension instructions used in your program. The core directives are emitted as needed, such as when direct-register assignment and the **_core_read()** and **_core_write()** intrinsics use extension core registers.

> **NOTE** This toggle does not emit directives when you use core registers within inline assembly. In that case, use inline assembly directly or use command-line option **-Hextfile** (described in Driver Option Reference on page 42).

Emit_line_records — Write a line table in the generated object file

Default: Off

When toggle `Emit_line_records` is on, the compiler writes a line table in the generated object file. This line table can be used by debuggers to correlate object code with source lines.

> **NOTE** All **-g*** debug options automatically emit line records.

Empty_is_void — Force C++ (void) interpretation of function declarations with no arguments

Default: On

For C++ Only

In C++, a function declared **int f();** means the same as **int f(void);** — the function takes no arguments. In C this is not the case; a function declared **int f();** can take any arbitrary number of arguments.

When toggle `Empty_is_void` is off, the compiler interprets such function declarations as if they were C code, not C++ code. This capability is useful in "wrapping" old C header files that might contain such declarations, when compiling with C++:

```
extern "C" {
   #pragma Off(Empty_is_void)
   #include "old_hdr.h"
   #pragma Pop(Empty_is_void)
   }
```

Toggle `Empty_is_void` is turned off in `c_wrapl.h`.

Enforce_access_control — Allow compiler-enforced access control

Default: On

For C++ Only

In C++ compilations, turn off toggle `Enforce_access_control` to remove all compiler-enforced access control. This removal can be useful when, for example, you are in a debugging mode and want to print out some private members of a class. Normally, access control might deny you access to those members.

Ensure_instantiation — Force instantiation of all template types

Default: Off

For C++ Only

Toggle `Ensure_instantiation` forces instantiation of any template types used within its scope, regardless of whether you turn on toggle `Auto_class_member_instantiation`. Related controls are pragmas `Ensure_instantiation` and `Exclude_instantiation`.

For more information about templates in general, see the *MetaWare C/C++ Language Reference*.

NOTE Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Epilog_merge — Use the same epilog for multiple functions

Default:

Off if optimization level = 0 — 7
On if optimization level = s or s1

When toggle `Epilog_merge` is on, the compiler attempts to replace the epilog instructions of a function with a jump to a compatible epilog of another function.

Epilog_trace — Generate information about a function being exited

Default: Off

When toggle `Epilog_trace` is on, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is exited. Related toggles are `Call_trace` and `Prolog_trace`. For more information, see Tracing Function Calls on page 292.

Exception_aware_class — Compile exception-aware classes

Default: Off unless you specify option **-Hexcept**

For C++ Only

Toggle `Exception_aware_class` causes all classes compiled within its scope to be exception-aware. That is, whenever such a class is constructed, its constructor increments the global subobject count by

1 (one). When such a class is destroyed, its destructor decrements the global subobject count by 1 (one).

Related toggle is `Support_exceptions`.

Exception-aware classes permit MetaWare C++ exception-handling (as distinct from system exception-handling). See the *MetaWare C/C++ Language Reference* for details about exception handling.

NOTE The MetaWare Development Toolkit distribution does not include exception-aware run-time libraries, because developers of embedded applications rarely want to use them.

To obtain exception-aware libraries, you must rebuild the supplied C++ run-time library source code with option **-Hexcept** specified. You can then explicitly link your application with those libraries.

Export_vtabs — Make virtual function tables public and defined

Default: Off

For C++ Only

Toggle `Export_vtabs` affects C++ classes that contain only inlined and pure virtual functions. If `Export_vtabs` is on, the virtual-function tables for such classes are made public and defined, and can be referenced from any module compiled with toggle `Import_vtabs` turned on.

Use toggle `Export_vtabs` in conjunction with toggle `Import_vtabs` to force only one copy of virtual-function tables for these classes to be defined in a program.

Fast_virtual_bases — Speed access to members of virtual bases

Default: Off

For C++ Only

Turning on toggle `Fast_virtual_bases` can improve speed of access to members of virtual bases of a class when a class hierarchy contains virtual bases as bases of other virtual bases. When a virtual-base class *B* is accessible only as a base of another virtual-base class, access to *B* requires two pointer indirections. In general, multiple pointer indirections can be required for more complicated class hierarchies.

Turning on toggle `Fast_virtual_bases` is one way to improve the speed of access. With the toggle on, the compiler guarantees that a class has enough copies of virtual-base pointers directly accessible within the class so that access to any virtual base of the class requires only a single pointer indirection.

Toggle `Fast_virtual_bases` can be turned on and off around class definitions; turn it on just before the definition of a class for which you want fast virtual-base access.

You can modify your C++ source to use fast virtual-base access on a case-by-case basis. For example:

```
class B { int x; };
class X : virtual B {        };
class D : virtual X {        };
D *dp;
```

Here, `dp->x` would require two virtual-base pointer indirections; the first to access **X** from **D**, and the second to access **B** (and thus **X**) from **X**. You can use the toggle, as just described:

```
class B { int x; };
class X : virtual B {        };
```

```
#pragma On(Fast_virtual_bases)
class D : virtual X {        };
#pragma Pop(Fast_virtual_bases)
D *dp;
```

But you can also modify the class definition of **D** to specify **B** as a direct virtual base of **D**. Direct virtual bases are always accessible via one virtual-base pointer indirection.

```
class B { int x; };
class X : virtual B {        };
class D : virtual X, virtual B {        };
D *dp;
```

In each of the preceding two examples, `dp->x` requires only one virtual-base pointer indirection.

> **CAUTION** If you leave toggle `Fast_virtual_bases` turned on indiscriminately, your program requires more space.

For_dcln_local — Treat loop-initialization variables as local to the loop

Default: On

This toggle causes the compiler to treat variables declared in the initialization portion of a **for** statement as local to the loop in accordance with recent C++ standards. Turn off this toggle to have the compiler treat variables declared in the initialization portion of a **for** statement as if they were declared outside the **for** loop, in accordance with older standards.

ForceDebug — Place named SDI for any in-scope structure in current compilation unit

Default: Off

Toggle `ForceDebug` causes the current compilation unit to become the debug repository for any named structure with no natural debug repository. Related toggles are `AutoDebug` and `NoDebug`.

When toggles `ForceDebug` and `NoDebug` are both on and you specify option **-g**, the compiler places named symbolic debugging information (SDI) in the current compilation unit, for any named C or C++ structure (including unions and classes) declared within the scope of toggle `ForceDebug`. The current compilation unit becomes the debug repository for these structures.

Func_replacement_prolog — Generate a prolog for use in redirecting a function

Default: Off

When toggle `Func_replacement_prolog` is on, the compiler emits a replacement prolog. You can then use the replacement prolog to redirect a function to a replacement by modifying ELF section `.fixtable`, as described in Replacing One Function with Another on page 318.

> **NOTE** Turning on toggle `Func_replacement_prolog` generates a replacement prolog that redirects the function to itself. You must modify the replacement prolog as described in Replacing One Function with Another on page 318 to achieve the desired redirection.

Function_template_deduction — Use 10/96 ANSI C++ draft function template matching rules

Default: On

For C++ Only

When toggle `Function_template_deduction` is on, the compiler implements function template matching rules consistent with the October 1996 version of the ANSI C++ draft standard.

When toggle `Function_template_deduction` is off, the compiler implements function template matching rules consistent with the *Annotated C++ Reference Manual*.

> **NOTE** Turning on toggle `Function_template_deduction` will break some old C++ programs.

For more information about templates in general, see the *MetaWare C/C++ Language Reference*.

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Import_vtabs — Assume virtual-function tables to be external for classes with inlined, pure virtual functions

Default: Off

For C++ Only

In C++ compilations, toggle `Import_vtabs` applies to classes that contain inlined, pure virtual functions. If `Import_vtabs` is on, the virtual-function tables for such classes are assumed to be defined in some other module. Use toggle `Export_vtabs` to force this definition in the defining module.

Use toggle `Import_vtabs` in conjunction with toggle `Export_vtabs` to force only one copy of virtual-function tables for these classes to be defined in a program.

For more information about virtual-function tables, see the *MetaWare C/C++ Language Reference*.

Inline_common — Solve the single-copy problem for inline functions (COMDAT)

Default: On

For C++ Only

Toggle `Inline_common` is a component of the COMDAT solution to the single-copy problem. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.)

When toggle `Inline_common` is on, it solves the single-copy problem for the following types of inline functions:

- inline functions whose bodies must be generated (for example, when the function is too big to inline, or when its address is taken)
- inline functions whose bodies are defined by the user within the scope of the toggle

This includes inline functions within classes, and global functions declared **extern inline**.

The following are related toggles:

- `Class_common_always`
- `Class_common_ok`
- `Defer_template_defn`
- `RTTI_common`

- Template_common
- Vtable_common.

Related toggles are Class_common_always, Defer_template_defn, RTTI_common, Template_common, and Vtable_common.

> **NOTE** Option **-Hcomdat** turns on toggle Inline_common. (This option is on by default.) Option **-Hnocomdat** turns off the toggle.

Inline_divide — Generate inline code for integer division

Default:

Off if optimization level = 0 — 5, s, or s1
On if optimization level = 6 — 7

When toggle Inline_divide is on, the compiler generates code to perform integer division inline, which provides a performance benefit for some applications at the cost of increased code size.

Interprocedural_optimizations — Determine the best possible order for generating code for functions in a module

Default:

Off if optimization level = 0 — 5 or if you specify option **-g**
On if optimization level = 6 — 7, s, or s1

When toggle Interprocedural_optimizations is on, the compiler determines the best possible order for generating code for the functions presented in a source module; that is, it optimizes across function boundaries.

Int_function_warnings — Warn about missing and mistyped returns

Default: Off

When toggle Int_function_warnings is on, the compiler warns about missing returns or mistyped returns. Such warnings occur in the following cases:

- when a function returning an **int** contains no **return** *expression*; statement
- when a function returning an **int** contains a **return** statement with no argument (that is, just **return**;)

When this toggle is off, the compiler suppresses these warning messages.

> **NOTE** Old C programs that do not use the reserved word **void** to indicate a function returning no result often generate many warning messages. Leave toggle Int_function_warnings turned off if you do not want to see those warning messages.

Keep_global_refs_in_loops — Do not move expressions with global or external variables out of loops

Default: Off

When toggle `Keep_global_refs_in_loops` is on, the optimizer does not move expressions involving globally or externally defined variables out of loops.

CAUTION Turning on toggle `Keep_global_refs_in_loops` does not have the same effect as declaring all globally or externally defined variables with the **volatile** type specifier.

When you declare a variable with the **volatile** type specifier, the optimizer does not retain that variable's value in a register across statement boundaries.

The optimizer also updates **volatile** variables in memory as soon as an assignment occurs, and does not delete what appear to be "dead" or redundant assignments to these variables.

Turn on toggle `Keep_global_refs_in_loops` if your program relies on signal functions to modify static or global variables. You can also specifically declare such variables with the **volatile** attribute.

See Accessing a Volatile Object on page 258 for more information.

Ldst_from_code — Access instruction memory using load and store instructions

Default: On

Toggle `Ldst_from_code` allows generated executable code to use load and store instructions to access instruction memory (switch-table load labels are in TEXT instead of LIT sections). If your processor configuration does not permit load and store accesses of instruction memory (such as coupled memory or a Harvard architecture), or if you want to prevent the compiler from placing data including address constants in code, turn off toggle `Ldst_from_code`.

List — Generate a source listing

Default: Off

When toggle `List` is on, the compiler writes a listing to standard output. `List` is typically specified at the start of compilation, but it can appear in the source file to turn the listing on or off around a particular section of source. For more information, see Chapter 15 — Generating Listings.

List_includes — Display names of files included during compilation

Default: Off

When toggle `List_includes` is on, the compiler writes to standard error the name of each file included during compilation, one name per line. The equivalent option is **-H**.

List_macros — List all macros to stdout

Default: Off

When this toggle is on, the compiler dumps the full list of macro definitions known to it at the end of compilation to standard output. The list includes all predefined macros and user-defined macros not removed by a **#undef** directive.

Long_enums — Map type enum variables to an int-sized area of memory

Default: Off unless you specify option **-Hpcc**

When toggle `Long_enums` is on, any variable of an **enum** type is mapped to an **int**-sized region of memory. When this toggle is off, such a variable is mapped to the smallest of one, two, or four bytes, such that all values are representable.

Make_externs_global — Make objects with storage class extern be global

Default: On

When toggle **Make_externs_global** is on, any local declaration of an object with storage class **extern** is made global if there is not already a global declaration of the object. Early C compilers promoted an **extern** declaration within a function to the global scope. This toggle supports programs that depend on that feature.

Masking_analysis — Remove redundant bit-masking instructions

Default:

Off if optimization level = 0 — 4
On if optimization level = 5 — 7, s, or s1

When toggle `Masking_analysis` is on, the compiler attempts to locate and remove redundant instructions involving bit masking. For example, an **AND** instruction using a mask of "a power of two less one" might be eliminated if the compiler can determine that the source register can have no significant bits outside of the mask.

Microsoft — Relax some standards to conform with Microsoft Visual C/C++

Default: Off

Toggle `Microsoft` changes the following constructs to conform with Microsoft C/C++.

- Permits anonymous **struct** members in a **union** (anonymous **union** members in a **union** are permitted by default).
- Permits a pointer to **int** to be compatible with a pointer to **long**.

Msgno — Display error message numbers

Default: On

By default, most MetaWare C/C++ compiler warning and error messages are displayed with a three-digit message number. These numbers can also be used as arguments to the `Offwarn`, `Offwarn`, and `Popwarn` pragmas to selectively disable warnings (but not errors).

Turn off toggle `Msgno` when you do not want the message numbers to be displayed.

Multiple_var_defs — Use UNIX "common" name strategy to implement semantics

Default: Off

When toggle `Multiple_var_defs` is on, the compiler uses the UNIX "common" name strategy to implement semantics. A common practice on UNIX systems is to allow multiple definitions of a global variable, as in the following.

```
File f1.c:
int x;
File f2.c:
int x;
```

UNIX linkers "merge" the two definitions of `x` because UNIX compilers typically translate the two definitions of `x` into "common" symbols.

NOTE Toggle `Multiple_var_defs` must be off for toggle Uncached_in_own_section to work.

Nested_enum_mangling — Correctly handle the mangling of nested enumeration types
Default: On

C++ Only
When toggle `Nested_enum_mangling` is on, the MetaWare C++ compiler correctly handles the mangling of enumeration types when those **enum** types are nested inside a class and then referenced as a formal argument of a function member of the class. (Prior releases of the MetaWare C++ compiler incorrectly mangled such nested **enum** types.)

If you are linking against any third-party C++ library that uses nested **enum** types, you must do both of the following:

- Turn off toggle `Nested_enum_mangling` when you compile your code.
- Use an older version of the MetaWare C++ libraries.

For information about obtaining the older C++ libraries, contact Technical Support.

Nested_types — Make nested types invisible outside the class where they are declared
Default: On

For C++ Only
The C++ language includes nested types, meaning that types declared within the scope of a class remain within that scope only and are not visible outside the class.

```
class s { typedef int T; };
T x;      // Error:  T is undeclared
```

In implementations of the C++ language before version 2.0, nested types were made visible to code outside the block in which the nested type was defined.

In version 2.0 and later implementations of the C++ language, nested types are not visible to code outside the block.

To accommodate prior practice, turn off toggle `Nested_types`. Turning the toggle off causes any nested types to be declared outside the class as well as inside, in the first enclosing non-class scope. In this example, `T` would be declared within **s** as well as at the global scope, so both `s::T` (the nested-class syntax) and `T` alone (the old syntax) suffice to refer to `T`.

For information about compiling C++ code, see Compiling and Linking C++ Modules on page 30.

NoDebug — Do not generate SDI for any named structure within toggle's scope
Default: Off

When toggle `NoDebug` is on, the compiler suppresses generation of SDI (symbolic debugging information) for all named C and C++ structures within the scope of the toggle.

> **NOTE** If toggle `ForceDebug` is also on, it overrides toggle `NoDebug`, causing the compiler to generate the SDI.

Related toggles are `AutoDebug` and `ForceDebug`.

Normalize_file_names — Remove redundant relative path information from include file names
Default: Off

When toggle `Normalize_file_names` is on, the compiler normalizes all pathnames and file names associated with any of the include directives. When the path and file names are normalized, the compiler removes redundant `../` relative directories from the path and file name. For example, with `Normalize_file_names` turned on, this path and file name:

```
../proj_a/source/../../debug/wrx.c
```

becomes simply `../debug/wrx.c`

Toggle `Normalize_file_names` is beneficial when you use the make-dependency options **-Hmake**, **-Hmakeof**, and **-Humake**.

UNIX and Linux hosts only

> **CAUTION** Toggle `Normalize_file_names` works only with directories that are on the same file system. If the relative path crosses a file system, the compiler will probably not be able to find the file by its normalized file name.

Overload_info — Show function chosen during overload operation

Default: Off

C++ Only

If toggle `Overload_info` is turned on in C++ compilations, the compiler informs you which function was chosen during overload resolution. This can help if you are not sure at a particular point in your program which function (of a set of overloaded functions) is being called. The information is emitted as a warning.

Packed_means_unaligned — Make packed structs unaligned

Default: Off

When toggle `Packed_means_unaligned` is on, the compiler assumes that **struct**s that are packed are also not aligned. This applies both to **struct**s packed using a pragma and **struct**s packed using the **_Packed** keyword. See Aligning Functions and Data on page 149 for more information about aligning and packing **struct**s.

Parm_assignment_warnings — Warn when assignments are made into a routine's value parameter

Default: Off

When toggle `Parm_assignment_warnings` is on, a warning is issued upon assignment into a routine's value parameter.

This toggle is useful for developers familiar with PL/1-style parameter passing, where all parameters are passed automatically "by var" (that is, by reference). PL/1 developers expect the assignment in the routine to be reflected back to the call to the routine. But this is not C semantics; therefore, the warnings might point to places where "var parameters" were intended.

Parm_warnings — Warn when non-prototyped function arguments do not match declared parameters

Default: On

When toggle `Parm_warnings` is on, the compiler produces warnings whenever arguments to a non-prototype (old-style) function do not match the types of the declared formal parameters of the

function. Frequently, this inconsistency is a source of bugs that are disastrous or that are difficult to find. For example, in this code:

```
double square(x) double x; {return x*x;}
...
printf("%d\n",square(3));
```

`square` is passed the integer `3`, not double-precision `3.0`, and the compiler issues a warning. The C language definition prohibits the compiler from casting `3` to a **double** before passing it.

To eliminate the compiler warnings, turn off toggle `Parm_warnings`. We recommend, however, that the program text be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

PCC — Relax ANSI C Standard restrictions to compile PCC-compatible programs

Default: Off

Many UNIX C programs that compile with the AT&T Portable C Compiler do not compile with an ANSI compiler such as MetaWare C/C++. When the `PCC` toggle is turned on, many ANSI restrictions are relaxed so that such programs are more likely to compile with MetaWare C/C++, with appropriate warnings.

The `PCC` toggle, when on, affects these ANSI Standard C restrictions:

- Structure-member selection is permitted on a variable of any type; it does not need to be a **struct** or **union**.

  However, if the name of the member being selected is declared as a member in more than one **struct** or **union** definition, each occurrence must be declared with the same type and be mapped at the same offset.
- Any pointer type is considered compatible with any other pointer type. (Turning on toggle `Pointers_compatible` also has this effect.)
- Pointers are considered compatible with any integer type. (Turning on toggle `Pointers_compatible_with_ints` also has this effect.)

NOTE To use the unsignedness-preserving semantics employed by PCC, you must turn on toggle `Use_UP_rules`.

These PCC language features are *not* supported:

- old-fashioned assignment operators (for example, =+)
- old-fashioned initialization (for example, **int** `x 0;`)

The `PCC` toggle also makes the inboard macro preprocessor conform more closely to the UNIX C preprocessor. For more information, see option **-Hpcc**, which turns on toggle `PCC`.

PCC_msgs — Reduce compiler diagnostic capabilities to the PCC level

Default: Off

When toggle `PCC_msgs` is on, the compiler's diagnostic capabilities are reduced to the level of the AT&T Portable C Compiler. At this level, the following warnings are not emitted:

```
Function called but not defined.
"=" used where "==" may have been intended.
```

In addition, toggle `PCC_msgs` disables warnings about passing an **int**, for example, to a non-prototyped function expecting a **long int**, because **int** and **long int** are the same size.

When all the warnings are enabled in the MetaWare C/C++ compiler, code must be very "clean" to get through the compiler without a warning. Toggle `PCC_msgs` is provided because some developers have code that was designed with a compiler that is not so demanding, and these developers prefer fewer prods from the compiler.

Pointers_compatible — Assign pointer values to incompatible pointer variables

Default: Off

When toggle `Pointers_compatible` is on, the compiler permits a pointer value to be assigned to an incompatible pointer variable, with an appropriate warning.

Pointers_compatible_with_ints — Make any type pointer compatible with int types

Default: Off

When toggle `Pointers_compatible_with_ints` is on, the compiler allows pointers of any type to be compatible with **int** types, with an appropriate warning.

Although this compatibility is in violation of the ANSI C Standard and MetaWare C/C++ specifications, some C programs assign pointers to **int** types, and vice versa. Toggle `Pointers_compatible_with_ints` allows such programs to be compiled without modification. The ANSI C Standard and the MetaWare C/C++ compiler disallow this dangerous practice because pointers are not necessarily the same size as **int** types on all machines.

Postfix_takes_two_args — Force overloaded postfix operators to require two arguments

Default: On

For C++ Only

In C++ compilations, when you overload the postfix operators **++** and **--**, you must use two arguments to distinguish them.

Turn off toggle `Postfix_takes_two_args` to eliminate this distinction. Then you can define just one ++ operator in a class, and invoke it either with the syntax ++*x* or the syntax *x*++.

Pre_allocate_limms — Assign frequently used limms to unallocated machine registers

Default:
Off if optimization level = 0 — 3
On if optimization level = 4 — 7, s, or s1

When toggle `Pre_allocate_limms` is on, the compiler attempts to assign frequently used long-immediate constants (*limms*) to unallocated machine registers.

- For leaf functions, only unused volatile registers will be chosen.
- For non-leaf functions, only unused non-volatile registers will be chosen.

This optimization can improve code size and performance because a reference to a long-immediate constant can be replaced with a register, resulting in a 4-byte instruction instead of an 8-byte instruction.

Predicate_transformations — Transform conditional branches to conditional execution

Default:
Off if optimization level = 0 — 4

On if optimization level = 5 — 7, s, or s1

When toggle `Predicate_transformations` is on, the compiler attempts to replace conditional branches with conditional execution of instructions.

Print_lattice_members — Include a list of class members in the class lattice display

Default: Off

For C++ Only

Turn on toggle `Print_lattice_members` to include a list of class members along with the class-lattice display produced with toggle `Print_lattices`. The members are displayed graphically.

For example, given this program text:

```
class V { int v; };
class A { int a; };
class B : A, virtual V { };
class C : A, virtual V { };
class D : B, C { void f(); };
```

this is the graphical display for class **D**:

```
D {f}
---------------------------
B             C
------------- -------------
A {a} v:V {v} A {a} v:V {v}
```

See toggle `Print_lattices` for an explanation of this format.

Print_lattices — Generate a graphical view of any class with one or more bases

Default: Off

For C++ Only

Use toggle `Print_lattices` to obtain a graphical view of any class that has one or more bases. The output is simple character text that can be displayed on any device.

The lattice is printed in two forms.

- The first form is a one-dimensional linear representation that uses brackets to enclose classes. For example, class `some_class` with bases `b1 ... bn` is displayed as follows:

  ```
  [some_class b1 ... bn]
  ```

- The second form is a two-dimensional graphical form. A class `some_class` with bases `b1 ... bn` is displayed as follows:

  ```
  some_class
  -------------
  b1   ...   bn
  ```

In both forms, virtual bases are replicated in the display and are preceded by `v:` to identify them as virtual bases.

Example

Suppose you turn on toggle `Print_lattices` for this program text:

```
class V { int v; };
class A { int a; };
class B : A, virtual V { };
```

```
class C : A, virtual V { };
class D : B, C { void f(); };
```

The compiler produces the following linear lattice display for class **D**:

```
Class lattice for D at test.cpp, L5/C1, in the
format [D B1 ... Bn]: [D [B A v:V] [C A v:V] ]
```

The compiler also produces this graphical lattice display, with duplication of any virtual bases:

```
D
-----------
B     C
----- -----
A v:V A v:V
```

See also toggle Print_lattice_members.

Print_layout — Print layout of data structures

Default: Off

When toggle `Print_layout` is on, the compiler prints to standard output the layout of all data structures in the module being compiled.

Print_protos — Write prototype-style function headers to standard output

Default: Off

When toggle `Print_protos` is on, the compiler writes to standard output a new, prototype-style function header for each function definition. This toggle aids in the conversion of C programs to the ANSI Standard C prototype syntax.

For example, for this function definition:

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces the following output:

```
int f(int *x, double (*y)(), int z[]);
```

The old function definition can then be replaced with the generated one.

Print_template_usage — Display templates used in a C++ compilation

Default: Off

For C++ Only

Turn on toggle `Print_template_usage` to find out which templates were used in a C++ compilation. Both function templates and class templates are printed. For information about templates in general, see the *MetaWare C/C++ Language Reference*.

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Print_var_info — Display mapping of auto-, register-, and argument-class variables

Default: Off

When toggle `Print_var_info` is on, the compiler displays where each auto-, register-, or argument-class variable is mapped on a per-function basis.

One of the following formats is used:

- `N ==> R` specifies variable `N` has been mapped to machine register `R`.
- **N**`+offset ==> R` specifies a member of **struct N** has been mapped to machine register `R`.
- `N ==> FP+`*offset* specifies variable `N` has been mapped into the stack frame at *offset* from base pointer `FP`.
- `N ==>` <**eliminated**> indicates variable `N` was eliminated through copy propagation.
- `N ==>` <**unreferenced**> indicates variable `N` was not referenced (or all references were optimized away); the variable was not mapped.

Prolog_trace — Generate information about a function being entered

Default: Off

When toggle `Prolog_trace` is on, the compiler inserts code in each function call. This inserted code causes information about the function to be emitted at run time when the function is entered. Related toggles are `Call_trace` and `Epilog_trace`. For more information, see Tracing Function Calls on page 292.

Prototype_conversion_warn — Warn when a prototype declaration forces conversion of a function argument

Default:

Off if pragma `Warning_level` is not set or if `Warning_level` = 0 or 1
On if pragma `Warning_level` = 2, 3, or 4

When toggle `Prototype_conversion_warn` is on, the compiler generates a warning message when a function's argument is converted due to a prototype declaration.

When you use function prototypes, the compiler can automatically convert a function's argument so that the argument's type matches that of the formal parameter. Wherever such a conversion does not match what would happen in the *absence* of prototypes, such C code would probably not run correctly when compiled with older C compilers that lack prototypes.

Prototype_override_warnings — Warn when an ANSI Standard C declaration overrides an old-style function definition

Default: On

When toggle `Prototype_override_warnings` is on, the compiler produces a warning whenever a declaration (not definition) of a function using the ANSI Standard C prototype syntax overrides the semantics of an old-style function definition. For example:

File `header.h`:

```
int func(float f, long l);
```

File `prog.c`:

```
#include "header.h"
int func(f, l) float f; long l; {
   ...
   }
void sub() {
   func(3, 4.4);  /* Passes 3.0 and 4L via   */
   }              /*  automatic conversion   */
```

With the header file included, the interface for **func()** is changed to prototype style (`3` is converted to **float** and `4.4` to **long**).

If `header.h` were not included, the call to **func()** in **sub()** would pass the **int** `3` and the **double** `4.4`, and probably **func()** would not work properly. In this example, you can find out how the compiler treats **func()** only by searching all the header files.

We recommend writing function definitions with the ANSI Standard C prototype syntax for improved readability and reliability. For example:

File `prog.c`:

```
/* ANSI Standard C prototype syntax */
int func(float f,long l) {
   ...
   }
```

You can disable the warning message by turning off the toggle `Prototype_override_warnings`.

> **NOTE** Use toggle `Print_protos` to aid in converting non-ANSI C programs to the ANSI Standard C prototype syntax.

Read_only_strings — Store all string constants in the read-only data section

Default: Off

When toggle `Read_only_strings` is on, all string constants are considered read-only and are placed in a read-only data section. Multiple occurrences of identical string literals within a single module appear only once in memory.

> **CAUTION** Turn off toggle `Read_only_strings` if your program modifies the space occupied by a string literal.

For example, if you compile the following program with toggle `Read_only_strings` turned on, a protection fault or incorrect result might occur at run time:

```
static char *p = "a string literal";
...
/* Modifies the string literal */
p[0] = 'A';
```

Recognize_library — Inline code for certain functions in the ANSI Standard C library

Default: On

When toggle `Recognize_library` is on, the compiler can recognize calls to certain functions defined in the ANSI Standard C library. The compiler might inline the function, perform an arithmetic strength reduction, or treat the call as an operation that is subject to global optimizations.

See Inlining Certain ANSI Standard C Functions on page 178 for details about which ANSI functions the compiler inlines when this toggle is turned on.

Reference_all_functions — Create a reference to every non-inline function in the program

Default: Off

For C++ Only
Turn on toggle `Reference_all_functions` to create a reference to every non-inline function declared in your program. The reference is made within the function **main()**. The result is not intended to be executed, but only linked.

Successful linking means that each such function has been implemented. Use toggle `Reference_all_functions` to check that you have implemented all functions that you declared in, for example, a class library.

Relative_includes — Look up included files relative to the directory of the including file
Default: On

When toggle `Relative_includes` is on, the compiler first looks up the *filename* of the following statement in the directory of the file containing the **#include** line:

`#include` "*filename*"

That is, the **#include** is relative to the file being scanned when the request is made.

The compiler then looks in the directories specified by options **-I** and **-YI**, in that order. After that, the compiler looks in directories from the standard list.

This algorithm permits compilation of a source file that is in a directory other than the current working directory. This is UNIX C standard practice.

When toggle `Relative_includes` is off, the compiler first looks up *filename* in the current working directory.

Respect_far — Treat functions declared to be _Far as far calls
Default: On

By default, the compiler treats all function calls as **_Near**. When toggle `Respect_far` is on, the compiler treats function calls declared to be **_Far** as real **_Far** calls.

See the following for more information:

- toggle `Default_far`
- option **-M***model* (in Driver Option Reference on page 42)
- type qualifiers **_Far** and **_Near** in the *MetaWare C/C++ Language Reference*

RTTI_common — Solve the single-copy problem for RTTI structures (COMDAT)
Default: On

For C++ Only
Toggle `RTTI_common` is a component of the COMDAT solution to the single-copy problem. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.)

When toggle `RTTI_common` is on, it solves the single-copy problem for classes without seats declared within its scope, or for non-class types.

The following are related toggles:

- `Class_common_always`
- `Class_common_ok`
- `Defer_template_defn`

- Inline_common
- Template_common
- Vtable_common.

NOTE Option **-Hcomdat** turns on toggle RTTI_common. (This option is on by default.) Option **-Hnocomdat** turns off the toggle.

Run_time_type_info — Generate and store run-time type information (RTTI) for a class

Default: Off

For C++ Only

Toggle Run_time_type_info ensures that, for every class compiled within its scope, run-time type information (RTTI) is generated and stored in the class's virtual function tables.

If toggle Run_time_type_info is not turned on, RTTI is generated only as needed by **typeid**, and there is no guarantee that the virtual-function tables will contain a pointer to RTTI.

Related toggle is Support_run_time_type_info.

For more information about RTTI, see the *MetaWare C/C++ Language Reference*.

Scalars_in_bss — Place all uninitialized variables in the BSS section

Default: Off

By default, the compiler places only uninitialized variables eight (8) bytes or larger in the BSS section. Turning on toggle Scalars_in_bss causes the compiler to place *all* uninitialized variables, including small scalar variables, in the BSS section.

NOTE Because toggle Scalars_in_bss affects the entire file being compiled, you can specify it only on the command line (using option **-Hon**) not within source code.

Seat_funcs_in_dcln_order — Base seat of class on source order

Default: Off

For C++ Only

Toggle Seat_funcs_in_dcln_order alters the heuristic that the compiler uses when determining whether to generate a seat for a class. Normally the compiler uses a heuristic based on the ordering of its internal tables. Toggle Seat_funcs_in_dcln_order causes the compiler to use the order in which the declarations are encountered in the source, which is often more intuitive.

CAUTION Code compiled with Seat_funcs_in_dcln_order is not binary-compatible with code compiled without Seat_funcs_in_dcln_order.

Source_code_order — Produce code in source order where possible

Default: Off if optimization level = 2 — 7, s, or s1,
On if optimization level = 0 or 1 or if you specify option **-g**

When toggle Source_code_order is on, the compiler attempts to generate code in source order. This makes code easier to debug at the assembly language level.

Specify_override — Require _Override for overriding subclasses

Default: Off

C++ Only

When toggle `Specify_override` is on, the compiler requires you to use the keyword **_Override** whenever you override a base class function.

Use this toggle to find subclasses that you have not qualified with **_Override**, and add **_Override**. Then if you change the signature of a base-class method, the compiler alerts you when subclasses of that class no longer override the base-class method.

Support_cast_keywords — Enable cast keywords in C++ code

Default: On

When toggle `Support_cast_keywords` is on, the C++ compiler enables the C++ language's cast keywords **static_cast**, **reinterpret_cast**, **const_cast**, and **dynamic_cast**. For more information about casting, see the *MetaWare C/C++ Language Reference*.

Related toggle is `Try_static_cast`.

For Embedded C++ Only

When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for cast keywords.

Using keyword **dynamic_cast** bears a run-time cost because it requires the presence of RTTI. Using the other cast keywords does not bear a run-time cost.

Support_exceptions — Enable exceptions in C++ code

Default: On

When toggle `Support_exceptions` is on, the C++ compiler enables the exceptions feature of the C++ language. For more information about exceptions, see the *MetaWare C/C++ Language Reference*.

Related toggle is `Exception_aware_class`.

For Embedded C++ Only

When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for exceptions.

Using exceptions bears a run-time cost.

Support_multiple_inheritance — Enable multiple inheritance in C++ code

Default: On

When toggle `Support_multiple_inheritance` is on, the C++ compiler enables the multiple-inheritance feature of the C++ language. For more information about multiple inheritance, see the *MetaWare C/C++ Language Reference*.

For Embedded C++ Only

When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for multiple inheritance.

Using multiple inheritance bears a run-time cost.

Support_mutable — Enable mutable in C++ code

Default: On

When toggle `Support_mutable` is on, the C++ compiler enables the **mutable** feature of the C++ language. For more information about **mutable**, see the *MetaWare C/C++ Language Reference*.

For embedded C++ Only
When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for **mutable**.

Support_namespaces — Enable namespaces and using features in C++ code
Default: On

When toggle `Support_namespaces` is on, the C++ compiler enables the namespaces and **using** features of the C++ language. For more information about these language features, see the *MetaWare C/C++ Language Reference*.

Embedded C++ Only
When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for namespaces and **using**.

Using namespaces and keyword **using** bears no run-time cost. A larger executable file might result from using namespaces (because global symbols in the symbol table might be longer, due to name mangling), but the text of symbols does not appear in downloaded code or data.

NOTE For information about a utility that unmangles names, see the *ELF Linker and Utilities User's Guide*.

Support_run_time_type_info — Enable run-time type information (RTTI) and typeid in C++ code
Default: On

When toggle `Support_run_time_type_info` is on, the C++ compiler enables the run-time type information and **typeid** features of the C++ language. For more information about RTTI and **typeid**, see the *MetaWare C/C++ Language Reference*.

Related toggle is `Run_time_type_info`.

Embedded C++ Only
When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for RTTI and **typeid**.

Using run-time type information and keyword **typeid** bears a run-time cost.

Support_templates — Enable templates in C++ code
Default: On

When toggle `Support_templates` is on, the C++ compiler enables the templates feature of the C++ language. For more information about templates, see the *MetaWare C/C++ Language Reference*.

Embedded C++ Only
When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for templates.

Using templates bears no run-time cost.

NOTE Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Support_typename — Enable typename in C++ code

Default: On

When toggle `Support_typename` is on, the C++ compiler enables the **typename** feature of the C++ language. For more information about **typename**, see the *MetaWare C/C++ Language Reference*.

For Embedded C++ Only

When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for **typename**.

Using keyword **typename** bears no run-time cost.

Support_vbases — Enable virtual bases in C++ code

Default: On

When toggle `Support_vbases` is on, the C++ compiler enables the "virtual bases" feature of the C++ language. For more information about virtual bases, see the *MetaWare C/C++ Language Reference*.

For Embedded C++ Only

When you specify option **-Hecpp** for Embedded C++ language support, this toggle is turned off, disabling support for virtual bases.

Using virtual bases bears a run-time cost.

Switch_pic — Generate only position-independent switch tables

Default: Off

By default, the compiler might generate either position-dependent or position-independent `switch` statements. Turn on toggle `Switch_pic` to generate only position-independent switch tables.

Tail_jumping — Replace calls to certain functions with jumps to those functions

Default:

Off if optimization level = 0 — 5
On if optimization level = 6 — 7, s, or s1

Normally, functions containing calls to other functions are not considered "leaf" functions and thus require a prolog and epilog code sequence.

When toggle `Tail_jumping` is on, the compiler detects functions in which the only calls are immediately followed by a **return**, and replaces the calls with jumps. The compiler then treats the function as a "leaf" function and eliminates the prolog and epilog sequences, including the **return** instruction.

If the function requires any stack space for saving non-volatile registers or local variables, the compiler does not perform this optimization.

Tail_recursion — Optimize recursive functions where possible

Default: Off

when toggle `Tail_recursion` is on, the compiler attempts to optimize recursive functions. For more information, see Tail Recursion on page 179.

Template_common — Solve the single-copy problem for template functions and classes (COMDAT)

Default: On

C++ Only

Toggle `Template_common` is a component of the COMDAT solution to the single-copy problem. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.).

When toggle `Template_common` is on, it solves the single-copy problem for all virtual function tables, RTTI structures, inline functions, and out-of-line function bodies generated as a result of instantiating a template that was declared within the scope of this toggle.

The following are related toggles:

- `Class_common_always`
- `Class_common_ok`
- `Defer_template_defn`
- Inline_common
- `RTTI_common`
- `Vtable_common`.

NOTE Default option **-Hcomdat** turns on toggle `Template_common`. Option **-Hnocomdat** turns off the toggle.

NOTE Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Template_return_mangling — Allow overloading of template functions by return type

Default: Off

C++ Only

In previous versions of the C++ standard, template functions were treated the same as ordinary functions, that is, they could not be overloaded based solely on return type. However, in the current standard, template functions are treated differently from ordinary functions in this respect and may be differentiated based solely on return type.

For example, when performing explicit template-function specialization such as the following, a set of functions might be differentiated only by return type.

```
template <class T>  void foo(T *x) { ... }
....
foo<int>(55);
foo<double>(5.9);

//Another example:
void (*fp)(int) = foo<int>;
```

In such cases, turn on toggle `Template_return_mangling` so that the functions can be differentiated, in conformance with the C++ standard.

Template_trivial_conversions — Allow trivial conversions for matching arguments to function templates

Default: Off

For C++ Only

In C++ compilations, turn on toggle `Template_trivial_conversions` to allow so-called trivial conversions (as defined in the *Annotated C++ Reference Manual*) when matching arguments to function templates. For general information about templates, see the *MetaWare C/C++ Language Reference*.

> **NOTE** Some implementations of the C++ standard template library (STL) produce superfluous compiler warnings. For more information, see Common Superfluous Warnings on page 340.

Thread — Specify that variables are to be stored in thread-local storage

Default: Off

When toggle `Thread` is on, all data declared or defined within the toggle's scope is stored in thread-local storage (TLS).

Thread-local storage allows each thread in a multi-threaded process to allocate storage for data specific to itself, instead of sharing data with the other threads in the process. You use toggle `Thread` in place of manually applying the storage modifier **__declspec(thread)**. For more information, see Thread-Local Storage and the thread Attribute on page 254.

> **CAUTION** Do not confuse toggle `Thread` with the **-Hthread** option. Toggle `Thread` is meant to be applied at the source-code level, while option **-Hthread** is meant to be applied to an entire compilation.
>
> If you apply toggle `Thread` to an entire compilation (using **-Hon=***`Thread`*), the compiler tries to make every declaration in the source file into a **__declspec(thread)** declaration, which is erroneous in most cases.

Trace_global_ctors — Indicate construction and destruction of global objects

Default: Off

For C++ Only

Toggle `Trace_global_ctors` instructs the compiler to generate warning messages with the file and line number where C++ global objects are constructed and destructed. This toggle is useful in helping find instances of an object being referenced before it is constructed.

Treat_exports_as_locals — Change public functions to local static functions

Default: Off

To improve optimization, use this toggle with option **-Hmerge** to make all the code and data in multiple source files available to the optimizer at one time. When you specify **-Hmerge** and toggle `Treat_exports_as_locals` is on, the compiler changes all public functions (except **main()**) to local static functions and changes all global data to local static data. The optimizer then does not generate

code for functions not accessed within the module and does not emit data into the assembly file unless that data is referenced.

NOTE This toggle is useful only when compiling all of the necessary source files at once using option **-Hmerge**, for example:
`mcc *.c -Hmerge -Hon=treat_exports_as_locals`

When this toggle is on, the compiler assumes the following:

- All of the modules needed by the program are contained within the current source module.
- The only public entry point is function **main()**.
- Any non-local function is contained within a library.

NOTE Toggle `Treat_exports_as_locals` has effect only at optimization levels 6, 7, *s*, and *s1*.

Trigraphs — Enable trigraphs

Default: Off unless you specify option **-Hansi**

When toggle `Trigraphs` is on, the compiler recognizes *trigraphs* — sequences of three characters that are converted to a single character. Trigraphs are useful on machines that have limited character sets; they allow you to represent characters that would otherwise be unavailable.

On ASCII-based systems, trigraphs are not generally useful. All the characters that trigraphs represent are typically available on contemporary keyboards. However, trigraphs are required for ANSI Standard C compatibility, so option **-Hansi** turns on toggle `Trigraphs`.

Try_static_cast — Replace old-style type casts with static_cast

Default: Off

For C++ Only

Toggle `Try_static_cast` causes all old-style C++ casts within its scope of the form

`(`*Type*`)`*expression*

to be treated as the new type cast **static_cast**. This toggle also causes the compiler to issue a warning at each line containing a cast. For more information about the new C++ cast notation, see the *MetaWare C/C++ Language Reference*.

NOTE If you use toggle `Try_static_cast`, be sure toggle `Support_cast_keywords` is also turned on.

Unaligned_means_packed — Pack unaligned structs

Default: Off

When toggle `Unaligned_means_packed` is on, the compiler assumes that structs that are unaligned are also packed. See Aligning Functions and Data on page 149 for more information about aligning and packing **struct**s.

Uncached_in_own_section — Place uncached data in separate section

Default: On

Toggle `Uncached_in_own_section` causes the compiler to place all variables declared **_Uncached** in a separate ELF section called `.ucdata`. If toggle `Volatile_cache_bypass` is on (the default), toggle `Uncached_in_own_section` also causes the compiler to place all variables declared **volatile** in `.ucdata`.

Placing uncached variables in a separate section prevents coherency problems due to cached and uncached variables sharing a cache line.

> **NOTE** Toggle `Multiple_var_defs` must be off for toggle `Uncached_in_own_section` to work.

Turn off toggle `Uncached_in_own_section` if you do not want uncached variables to be placed in a separate `.ucdata` section. When toggle `Uncached_in_own_section` is off, **_Uncached** variables are placed in the same sections as other variables, as described in How the Compiler Maps Variables to Storage on page 252 (the placement of **volatile** variables depends on the setting of toggle `Volatile_cache_bypass`).

If you are placing your uncached data with a linker command file, you can save space by turning off toggle `Align_uncached_section`. For more information about using linker command files, see the *ELF Linker and Utilities User's Guide*.

For more information about using uncached data, see Working with Volatile and Uncached Variables on page 222.

Undefined_preprocessor_id — Warn on undefined #if and #elif identifiers

Default: Off

When this toggle is on, the compiler warns when identifiers used with **#if** and **#elif** are not defined (and thus use the default value of `0`). For example, **#if** `T` produces a warning where `T` is not defined.

Unroll_loops — Allow unrolling of loops

Default:

Off if optimization level = 0 – 6, s, or s1
On if optimization level = 7

Toggle `Unroll_loops`, when on, allows the compiler to unroll loops manually if requested or automatically at the appropriate optimization level.

> **NOTE** The compiler can usually generate faster code without loop unrolling if it can use the zero-delay-loop mechanism built into ARC processors. Loop unrolling inhibits zero-delay loops. For this reason, the compiler performs loop unrolling only at optimization level 7.

Unsized_arrays_in_small_data — Allocate external unsized arrays in small data

Default: Off

When toggle `Unsized_arrays_in_small_data` is on, the compiler assumes that external arrays declared with no size are in small data. Option **-Hall_sdata** turns on toggle `Unsized_arrays_in_small_data`.

NOTE The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** into small data unless you qualify them as **_Near**.

Related Compiler Controls

- options **-Hall_sdata**, **-Hnosdata**, and **-Hsdata0**
- toggles `Use_sdata` and `Common_in_small_data`
- pragmas `Push_small_data` and `Pop_small_data`

Related Topic

- Using Small-Data Sections on page 214

Use_millicode — Generate calls to millicode functions to reduce size of prolog and epilog

Default:

Off if optimization level = 0 — 7
On if optimization level = s or s1

When toggle `Use_millicode` is on, the compiler enables the calling of special-purpose library functions (*millicodes*) to implement function prolog and epilog code. This action can reduce code size significantly, but does so at the expense of execution speed.

Use_reciprocal_for_divide — Convert a floating-point divide-by-constant to a multiply-by-reciprocal

Default: Off

When toggle `Use_reciprocal_for_divide` is on, a floating-point divide by a constant is converted to a multiply by the reciprocal of the constant, because a multiply is typically faster than a divide.

CAUTION This conversion can introduce inaccuracy into a program, because the reciprocal might not be exact. In general, the last bit of the result might be different. For most programs, this does not matter; toggle `Use_reciprocal_for_divide` is provided for users who want the extra performance.

Even if toggle `Use_reciprocal_for_divide` is off, the compiler might choose to multiply by the reciprocal if the reciprocal has an exact binary representation; for example, the reciprocal of 2.

For more information, see Expression Simplification: Converting Division to Multiplication on page 176.

Use_sdata — Respect designation of %gp as the small-data base register

Default: On

When toggle `Use_sdata` is on, the compiler respects the designation of **reserved** applied to small-data base register `%gp`. When this toggle is off, the compiler uses `%gp` as a general-purpose register.

Related Compiler Controls

- Options **-Hall_sdata**, **-Hnosdata**, and **-Hsdata0**
- Toggles `Unsized_arrays_in_small_data` and `Common_in_small_data`
- Pragmas `Push_small_data` and `Pop_small_data`

Related Topic

- Using Small-Data Sections on page 214

Use_UP_rules — Widen unsigned char and unsigned short to unsigned int

Default: Off

In C, there are two rules for implicitly widening an **unsigned char** or **unsigned short** operand in an arithmetic expression: the *value-preserving* (VP) rule and the *unsignedness-preserving* (UP) rule.

The Value-Preserving Rule

The VP rule widens an **unsigned char** or **unsigned short** to **signed int**. This is the rule used in ANSI Standard C.

The Unsignedness-Preserving Rule

The UP rule widens such operands to **unsigned int**. This rule is used by older compilers, such as those based on the AT&T Portable C Compiler.

Program Behavior

Some programs behave differently depending on which rule is in force. For example:

```
unsigned char c;
...
if (c-1 > 0) fnc();
```

If the value of `c` is `0` (zero) when the **if** statement is executed, the relational expression is False if the VP rule is used, but True if the UP rule is used. The UP rule widens `c` to an **unsigned int**; therefore, the type of the expression `c-1` is also **unsigned int**.

When to Use This Toggle

If you are compiling code that was originally developed under a UP compiler, turn on toggle `Use_UP_rules`. See toggle `VP_UP_warn` for more information.

> **CAUTION** If toggle `Use_UP_rules` is turned on, some macros in the standard header files might not function properly, because they were written with the assumption that value-preserving rules apply.

Volatile_cache_bypass — Bypass the data cache when accessing volatile data

Default: On

Toggle `Volatile_cache_bypass` causes accesses to **volatile** data to bypass the data cache in the memory controller. The **.di** instruction suffix is used for all loads and stores of data declared **volatile**.

- If your target system uses a data cache, any data declared **volatile** must be aligned on a cache-line boundary. For more information, see Specifying the Alignment of Global or Static Variables on page 149.
- To place **volatile** variables in their own `.ucdata` section, turn on toggle `Uncached_in_own_section` as well (on by default for ARCtangent-A5 and later).
- ARCtangent-A5 and later have a type qualifier that loads and stores variables without involving the data cache and without the additional semantic implications of **volatile**; see Using the _Uncached Type Qualifier on page 224.
- See also Working with Volatile and Uncached Variables on page 222.

VP_UP_warn — Warn about expressions with more than one value

Default:

Off if pragma `Warning_level` is not set or if `Warning_level` = `0`, `1`, `2`, or `3`
On if pragma `Warning_level` = `4`

When toggle `VP_UP_warn` is on, the compiler issues a warning when it encounters an expression that could have more than one value, depending on whether value-preserving or unsignedness-preserving rules are used. For more information, see toggle `Use_UP_rules`.

Vtable_common — Solve the single-copy problem for virtual-function tables (COMDAT)

Default: On

For C++ Only

Toggle `Vtable_common` is a component of the COMDAT solution to the single-copy problem. Turning this toggle on solves the single-copy problem for virtual-function tables. (See the *MetaWare C/C++ Language Reference* for an explanation of the COMDAT solution.)

When toggle `Vtable_common` is on, the compiler places multiple copies of certain class's virtual-function tables in separate compilation units. However, the compiler does this only for classes without a "seat" declared within the scope of the toggle. (A *seat* is unique compilation unit in which to lay down the virtual-function tables.)

See also the complementary toggle `Class_common_always`, which dispenses with the notion of the "seat" of a class.

Other related toggles are the following:

- `Class_common_ok`
- `Defer_template_defn`
- Inline_common
- `RTTI_common`
- `Template_common`

NOTE Option **-Hcomdat** turns on toggle `Vtable_common`. (This option is on by default.) Option **-Hnocomdat** turns off the toggle.

For more information about virtual-function tables, see the *MetaWare C/C++ Language Reference*.

Wait_for_license — Wait until compiler license is available

Default: Off

Normally the compiler exits and issues an error message when informed by the FLEX*lm* license manager that no license is available. Turn on toggle `Wait_for_license` when such errors occur because another user has checked out the license and you want to wait for the license to become available.

Widen_float_args — Widen float arguments to double

Default: On unless you specify option **-fsingle2**

The compiler ordinarily widens arguments of type **float** to type **double** under the following conditions:

- The argument is passed to a non-prototyped function.
- The argument is passed as a variable argument to a prototyped function.

(In both cases, the compiler widens the argument *before* passing it. This is the ANSI Standard C convention.)

When toggle `Widen_float_args` is turned off, no such conversions are performed.

CAUTION Beware of passing a **float** argument to a function that was compiled with a different setting for toggle `Widen_float_args`. For example, when this toggle is off, the values of **float** expressions cannot be displayed with **printf()**.

Xmac_d16 — Support dual 16-bit XMAC extensions in compiler

Default: On

Turn this toggle off when you use the corresponding driver option to enable assembly support for the extension (for example for use within inline assembly) but you do not want the compiler to generate code using the extension.

Example

```
mcc -Xmul32x16 -Hoff=xmul_32x16 -Xxmac_d16 -Hoff=xmac_d16
```

Related Topics

- Driver option -Xxmac_d16 on page 50
- Toggle Xmul_32x16

Xmul_32x16 — Support 32x16 multiply instructions in compiler

Default: On

Turn this toggle off when you use the corresponding driver option to enable assembly support for the extension (for example for use within inline assembly) but you do not want the compiler to generate code using the extension.

Example

```
mcc -Xmul32x16 -Hoff=xmul_32x16 -Xxmac_d16 -Hoff=xmac_d16
```

Related Topics

- Driver option -Xmul32x16 on page 48
- Toggle Xmac_d16

Chapter 6 — Using Compiler Pragmas

In This Chapter

- Overview
- Setting Pragmas
- Pragma Reference
- Customizing Your Program with Pragmas

Overview

This chapter documents MetaWare C/C++ compiler pragmas and describes how to use them.

Compiler pragmas control toggle settings, linkage to external functions, data storage, inclusion and listing of source text, and other features.

Setting Pragmas

You can set compiler pragmas in an IDE, in your source code, or by using a command-line option

If you are using a version of the tools that includes an IDE, you can set pragmas in the IDE. For more information, see the IDE user's guide.

You can set compiler pragmas in your source code, using one of the following general forms:

ANSI C Standard Syntax

`#pragma` *Pragma_name* `/* or */`
`#pragma` *Pragma_name*`(`*Pragma_parameters*`)`

MetaWare C/C++ Syntax

`pragma` *Pragma_name*`;` `/* or */`
`pragma` *Pragma_name*`(`*Pragma_parameters*`);`

Pragma_parameters is a list of constant expressions separated by commas. The number and types of the expressions depend on the particular pragma. A pragma can appear anywhere a statement or declaration can appear. Pragma names are not case sensitive.

Although **#pragma** is the ANSI C Standard form, **pragma** existed in MetaWare C prior to the ANSI C Standard. Either form is acceptable. An advantage of the MetaWare C/C++ **pragma** syntax is that it can appear in macros; **#pragma** cannot. The MetaWare C/C++ **pragma** statement must be terminated with a semicolon, and can appear only in a statement or declaration context.

CAUTION Using pragmas within a function might not yield the expected behavior. Apply pragmas outside of function bodies for best results.

Each pragma has a default value. You can override the defaults and thereby customize the compiler to a local environment by modifying the driver configuration file, by specifying pragmas on the command line, or by supplying a `.pro` file with new pragma values specified.

Use option **-Hpragma** on the command line to specify individual pragmas:

`-Hpragma=`*Pragma_name*
`-Hpragma=`*Pragma_name*`(`*Pragma_parameters*`)`

For more information, see the following:

- option -Hpragma in Driver Option Reference on page 42
- modifying the driver configuration file in Appendix B — Configuring the Driver
- .pro files in Using .pro Files on page 39

Pragma Reference

Alphabetical List of Pragmas

This section lists all compiler pragmas in alphabetical order. For a quick-access list, see Quick Pragmas List on page 357.

Alias(*Internal_name*,"*External_name*") — Assign an external name to an internal identifier

Pragma `Alias` associates an external name with an internal name, for external or public purposes.

- *Internal_name* is a variable or function identifier.
- *External_name* is a string constant denoting the alternate or external name; it must be enclosed in double quotes.

External_name does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker). See Aliasing External Names on page 154.

Align_members([*n*]) — Set the maximum boundary for structure-member alignment

Pragma `Align_members` specifies the maximum byte boundary on which an (unpacked) structure member must be aligned. *n* is an integer constant denoting the maximum boundary. (The default alignment is the same as the default system alignment, which varies by operating system.)

Specified without any arguments, pragma `Align_members()` restores the default boundary.

Related pragmas are Pack and Push_align_members. For more information, see Aligning Functions and Data on page 149.

The default setting for pragma `Align_members` is `4`.

Align_to(*alignment*, *var*[, *var*…]) — Specify the alignment of global or static variables

Pragma `Align_to` specifies the alignment of global or static variables. The *alignment* specified must be a power of two between one and 32768.

Restrictions

- Pragma `Align_to` works only with global or static variables
- Pragma `Align_to` cannot change the alignment of types.
- Pragma `Align_to` does not handle complex identifiers such as templates.
- You must apply pragma `Align_to` at the first definition of the variable.
- To place variables that you align using pragma `Align_to` in small data, you must qualify them as **_Near**.
- Pragma `Align_to` does not affect stack-based data structures.

  To align storage on the stack, you must provide code as shown in Example 40 Aligning Data Storage on the Stack on page 213.

See Specifying the Alignment of Global or Static Variables on page 149 for an example of how to use this pragma.

Alloc_text(*seg_name*, *function*[, *function*…]) — Group functions in specified text segment

Pragma `Alloc_text` allows you to group in the text segment *seg_name* the functions you specify as the remaining arguments. You can specify any number of functions.

Asm("*assembly_string*") — Insert a line of assembly code

Pragma `Asm` places the string *assembly_string* in the assembly output of the compiler at the point where it is specified, then appends a newline character.

You can use pragma `Asm` either inside or outside of a function, but using it inside a function can compromise the performance of your program, because the compiler must be conservative with optimizations around the inserted assembly.

CAUTION The compiler does not attempt to check the string for validity.

For more information, see Chapter 8 — Inlining and Chapter 9 — Inline Assembly Code.

Asm_field_offsets(["*filename*",]["*naming_spec*",] *struct1* …) — Find field offsets

Pragma `Asm_field_offsets` produces an ASCII assembly listing of the offsets of the fields (members) of **struct** *struct1* and any additional **struct**s you name. You may use **typedef** or variable names, but they must refer to a **struct**.

The compiler sends the list to `stdout` unless you specify a *filename* as the optional first argument.

NOTE If the *filename* contains the sequence `%m`, the compiler interprets it as the *naming_spec*. If you want to use the sequence `%m` in the *filename*, specify the *naming_spec* first, and the *filename* second.

The optional second argument *naming_spec* is a quoted string that specifies how to name the symbols defined in the output file.

- *%t* is replaced with the name of the **typedef**, **struct**, or variable.
- *%m* is replaced with the name of the field (member).
- Other characters are echoed as you specify.

The default naming specification is `"%t.%m"`.

CAUTION Although argument *naming_spec* is optional, if used it must contain at least one instance of *%m*.

This pragma is useful when mixing assembly and C/C++ files in your application. For an example, see Finding Field Offsets on page 162.

For a pragma that produces the same output with **#define** directives for the preprocessor, see pragma `Define_field_offsets`.

Aux_register(*reg_num* [, *property*…]) — Define an auxiliary register

Pragma `Aux_register` defines an auxiliary register to the compiler.

Parameter *reg_num* is the auxiliary-register address value associated with the auxiliary register. It must be a compile-time constant.

Properties

The optional comma-separated *property* parameters provide the compiler with information about the auxiliary register. Specify each property using named-parameter-association syntax. The following properties are allowed:

`name => "`*ar_name*`"`

Declares the auxiliary-register name as *ar_name*. If you do not supply this property, the compiler uses a default name based on the auxiliary-register numeric value in the format *rreg_num*, where *reg_num* is the auxiliary register's numeric value.

`blocking_cycles =>` *n*

Specify this *property* if a write to the register stalls the processor for *n* cycles.

`latency cycles =>` *N*

Specify this *property* when a write to the register followed by a read of the register requires intervening cycles to avoid a stall. *N* is the number of intervening cycles.

`non_interlock_cycles =>` *N*

Specify this *property* if a write to the register *must* be followed by *N* cycles prior to a read of the register. The compiler adds **NOP** instructions if it cannot schedule other intervening instructions.

`effects => "`*register_effect*`[;`*register_effect*`...]"`

Specify this *property* when accessing the register causes accesses to other registers indirectly. Each *register_effect* substring describes a register side effect. The format of *register_effect* is as follows:

name`:`*effect*`[:`*effect*`...]`

- *name* is the name of the register affected by the side effect (with or without a leading '%'). *name* must be known to the compiler from a previous use of pragma `aux_register` or `core_register`, or from the base-case register set of the processor.
- *effect* may be one of the following:

`is_read`	Specify if register *name* is read as a side effect.
`maybe_read`	Specify if register *name* might be read as a side effect.
`is_written`	Specify if register *name* is written as a side effect.
`maybe_written`	Specify if register *name* might be written as a side effect.
`is_trashed`	Specify if register *name* is written with a value that is not intended to be used later.

As an alternative to *name*, the register may be specified numerically as follows:

`reg=`*N*`:`*effect*`[:`*effect*`...]`

or

`auxreg=`*N*`:`*effect*`[:`*effect*`...]`

- *N* is a compile-time constant identifying the number of the core or auxiliary register.

CAUTION If read or write operations on your auxiliary register have side effects and you access the auxiliary register from C, be sure to use pragma `Aux_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

`side_effects => "`*`rw`*`"`

Declares that the auxiliary register has side effects when read or written, or both. Specify *r* for side effects when the auxiliary register is read, *w* for side effects when the auxiliary register is written, and *rw* for side effects when the auxiliary register is either read or written.

CAUTION If read or write operations on your auxiliary register have side effects and you access the auxiliary register from C, be sure to use pragma `Aux_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

For an example of usage, see Defining Registers on page 163.

BSS([*"BSS_name"*]) — Specify an alternate BSS section name

Pragma `BSS` specifies the name of the BSS section. *BSS_name* is a string constant denoting the section name; it must be enclosed in double quotes. Specified with no arguments, pragma `BSS()` cancels the effect of a prior pragma `BSS("`*BSS_name*`")`.

NOTE Do not use as your section name any of the default section names listed in Table 13 Default Control Sections for Code and Data on page 252.

NOTE Pragma BSS is intended for local static variables. If you declare global variables within the scope of pragma BSS, leave toggle Multiple_var_defs off to prevent the compiler from mapping the global variables into a common section instead of the named BSS section.

Header **extern** declarations without this pragma bracketing override definitions that use them. Always apply this pragma to the header **extern** declarations. It is then not needed in the definition.

Related Topics

- Specifying the Name of the Default BSS Section on page 160
- Specifying Your Own BSS Section on page 158
- How the Compiler Maps Variables to Storage on page 252

Called_infrequently (*string* [, *string*]) — Designate as "called infrequently" all functions whose names contain a specified string

Pragma *Called_infrequently* directs the compiler to apply the **_CALLED_INFREQUENTLY** calling-convention attribute to any function whose name contains at least one of the *string* arguments as a substring, provided the function is declared within the scope of the pragma.

Pragma `Called_infrequently` must be specified before the declaration of the function(s).

For more information, see the following:

- Designating Certain Functions As "Called Infrequently" on page 176
- Appendix C — Calling Conventions

Calling_convention ([*Attributes* [, *_DEFAULT*]]) — Allow MetaWare C/C++ modules to call functions not compiled with MetaWare C/C++

Pragma `Calling_convention` controls linkage to subroutines by allowing calls to modules compiled with other C or C++ compilers, or in other languages such as assembly, Pascal, PL/M, or FORTRAN, from modules compiled with MetaWare C/C++.

- *Attributes* is a bit pattern in which each bit specifies a particular calling convention attribute.
- **_DEFAULT** sets the default calling convention.

Specified with no arguments, pragma `Calling_convention()` cancels the effect of pragma `Calling_convention(`*Attributes*`)`. See Appendix C — Calling Conventions for a complete discussion of calling conventions.

Code(["*Section_name*"]) — Specify the code section name

Pragma `Code` specifies the name of the code section. *Section_name*, a string constant, must be enclosed in double quotes. Specified with no arguments, pragma `Code()` cancels a previous pragma `Code("`*Section_name*`")`.

For more information, see Specifying an Alternate Code Section on page 159.

NOTE Header **extern** declarations without this pragma bracketing override definitions that use them. Always apply this pragma to the header **extern** declarations. It is then not needed in the definition.

Condition(*condition_code* [, *property*…]) — Define an extension condition code

Pragma **Condition** defines an extension condition code to the compiler.

Parameter *condition_code* is the numeric value of the condition code.

The optional comma-separated *property* parameters provide the compiler with information about the condition code using named parameter association syntax. The following properties are allowed:

`name =>` *"cc_name"*

Declares the condition code mnemonic to be *cc_name*. If you do not supply a name, the compiler invents a name based on the condition code numeric value.

`inverse =>` *inv_condition_code*

Declares *inv_condition_code* to be the inverse of *condition_code*. It must be the case that if *condition_code* is true, *inv_condition_code* is not true, and if *inv_condition_code* is true, *condition_code* is not true. We recommend declaring the *inv_condition_code* because knowing the inverse allows the compiler to more effectively optimize use of the **_condition()** compiler-intrinsic function.

By default, the compiler emits the directives required to define the extension condition codes to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle Emit_core_directives.

For more information, see the following:

- Using Extension Condition Codes in C/C++ on page 166
- Compiler-intrinsic function **_condition()** in Appendix A — Intrinsic Functions

Core_register(*reg_num* [, *property*...]) — Define an extension core register

Pragma `Core_register` defines an extension core register to the compiler.

Parameter *reg_num* is the numeric value of the core register.

Properties

The optional comma-separated *property* parameters provide the compiler with information about the core register. Specify each property using named-parameter-association syntax. The following properties are allowed:

`name => "`*cr_name*`"`

Declares the core register name as *cr_name*. If you do not supply this property, the compiler uses a default name based on the core register numeric value in the format *rreg_num*, where *reg_num* is the core register's numeric value.

`usage =>` "*usage*"

The *usage* parameter can be one of the following:

```
reserved
removed
scratch
saved
```

The usages are as follows:

`usage => "reserved"`

Use this property when you want the compiler to refrain from allocating this register when generating code. The register may then be used for any application-specific purpose. For example, a subset of registers could be reserved to implement fast interrupt handlers that don't need to save and restore any registers. Or a frequently accessed global variable could be removed and a register used instead, with the _core_read() and _core_write() intrinsics. You can also reserve a register for use as a small-data base register for your own small-data sections (see Adding Your Own Small-Data Sections on page 218).

NOTE Only selected registers not deemed essential by the compiler may be reserved.

See option **-Hreg_reserve** for a more convenient way to reserve a list of registers.

`usage => "removed"`

Use this property when a register or subset of registers has been removed from the ARC core. The compiler will not generate code which uses the register. It will not be possible to link code with removed registers with other objects that use those registers.

NOTE Only selected registers not deemed essential by the compiler may be removed.

See option **-Hreg_remove** for a more convenient way to remove a list of registers.

`usage => "scratch"`

Use this property when you wish to add an extension core register to the compiler. The core register must not have any unusual side effects or latencies — it must behave like a base-case core register. The compiler will use the register as a volatile scratch register, meaning that the register value is not preserved across function calls.

`usage => "saved"`

Use this property when you wish to add an extension core register to the compiler. The core register must not have any unusual side effects or latencies — it must behave like a base-case core register. The compiler will use the register as a non-volatile saved register, meaning that the register value is preserved across function calls.

`blocking_cycles =>` *n*

Specify this *property* if a write to the register stalls the processor for *n* cycles.

`latency cycles =>` *N*

Specify this *property* when a write to the register followed by a read of the register requires intervening cycles to avoid a stall. *N* is the number of intervening cycles.

`non_interlock_cycles =>` *N*

Specify this *property* if a write to the register *must* be followed by *N* cycles prior to a read of the register. The compiler adds **NOP** instructions if it cannot schedule other intervening instructions.

`effects => "`*register_effect*`[;`*register_effect*`...]"`

Specify this *property* when accessing the register causes accesses to other registers indirectly. Each *register_effect* substring describes a register side effect. The format of *register_effect* is as follows:

name`:`*effect*`[:`*effect*`...]`

- *name* is the name of the register affected by the side effect (with or without a leading '%'). *name* must be known to the compiler from a previous use of pragma `aux_register` or `core_register`, or from the base-case register set of the processor.
- *effect* may be one of the following:

`is_read`	Specify if register *name* is read as a side effect.
`maybe_read`	Specify if register *name* might be read as a side effect.
`is_written`	Specify if register *name* is written as a side effect.
`maybe_written`	Specify if register *name* might be written as a side effect.
`is_trashed`	Specify if register *name* is written with a value that is not intended to be used later.

As an alternative to *name*, the register may be specified numerically as follows:

`reg=`*N*`:`*effect*`[:`*effect*`...]`

or

`auxreg=`*N*`:`*effect*`[:`*effect*`...]`

- *N* is a compile-time constant identifying the number of the core or auxiliary register.

> **CAUTION** If read or write operations on your core register have side effects and you access the core register from C, be sure to use pragma `Core_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

`side_effects => "`*rw*`"`

Declares that the core register has side effects when read or written, or both. Specify *r* for side effects when the core register is read, *w* for side effects when the core register is written, and *rw* for side effects when the core register is either read or written.

> **CAUTION** If read or write operations on your core register have side effects and you access the core register from C, be sure to use pragma `Core_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

Reading and writing using intrinsics

By default, when you use compiler-intrinsic function **_core_read()** or **_core_write()** to access a register, the compiler emits the required directives to define the extension core registers to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle `Emit_core_directives`.

Related Topics

- Defining Registers on page 163
- Compiler-intrinsic functions **_core_read()** and **_core_write()** in Appendix A — Intrinsic Functions
- Pragma Reg_remove
- Pragma Reg_reserve

Data([[*Section_type*,]"*Sect_name*"[, align => *n*][, sda_base => *n*]]) — Allocate named blocks for data storage

Pragma `Data` specifies the use of named blocks for data storage allocation.

- *Section_type* is an optional constant that specifies the type of the section. The following are possible values for the *Section_type* argument:

Section Type	Description
BSS	Uninitialized data (similar to `.bss`)
COMMON	Uninitialized data with common semantics (similar to `.bss`)
COMMON_DATA	Initialized data that can be defined in multiple compilation units. Only one copy is retained by the linker (similar to `.data`)

DATA	Initialized data (similar to `.data` and `.data1`)
LIT	Read-only data (similar to `.rodata`)
TEXT	Executable code (similar to `.text`)

If you do not specify a *Section_type*, the section defaults to `DATA`.

- *Sect_name* is a string constant that denotes the name of the section; it must be enclosed in double quotes. Do not use as your section name any of the default section names listed in Table 13 Default Control Sections for Code and Data on page 252.

  Pragma `Data()` must be used either with a *Sect_name* or with no parameters. Specified with no *Sect_name*, pragma `Data()` cancels a prior pragma `Data(`*Section_type*`, "`*Sect_name*`")`. Each pragma `Data` must be terminated or "turned off" in the same scope in which it is turned on.

  The *Section_type* and `"`*Sect_name*`"` parameters may be entered in either order; "*Sect_name*" must be enclosed in quotes, and *Section_type* must *not* be enclosed in quotes.

TIP You can specify *Section_type* and *Sect_name* in either order.

- `align =>` *n* is the optional minimum section alignment of *Sect_name*. For an example of the use of this parameter, see Aligning User-Defined Storage Blocks on page 149.
- `sda_base =>` *n* is the optional small-data base register when allocating a small-data section. We recommend that registers reserved for additional small-data sections start from `25` and work downward as needed, as these registers are less frequently used by the compiler. You can also reserve a scratch register like `r8` or `r7` for this purpose. Enter the register number without 'r,' for example `sda_base => 25`. For more information and an example of the use of this parameter, see Adding Your Own Small-Data Sections on page 218

NOTE Header **extern** declarations without this pragma bracketing override definitions that use them. Always apply this pragma to the header **extern** declarations. It is then not needed in the definition.

Related Topic

- Specifying an Alternate Static Data Section on page 157.

Limitations of Pragma Data

The compiler places writable string literals used to initialize pointers in the default data section, not in the data section specified with pragma Data.

To place a string in the data section specified with pragma `Data`, you must initialize the string as an array:

```
char str[] = "This is a string.";
```

instead of as a pointer:

```
char *str  = "This is a string.";
```

You can use pragma `Data` for laying out data in the specified section, and for mapping variables into a named data section or a common block.

NOTE If an uninitialized **extern** variable is declared inside the scope of pragma `Data`, the compiler issues a warning.

Define_field_offsets(["*filename*",]["*naming_spec*",] *struct1* …) — Find field offsets

Pragma `Define_field_offsets` produces an ASCII preprocessor listing of the offsets of the fields (members) of **struct** *struct1* and any additional **struct**s you name. You may use **typedef** or variable names, but they must refer to a **struct**.

The compiler sends the list to `stdout` unless you specify a *filename* as the optional first argument.

NOTE If the *filename* contains the sequence `%m`, the compiler interprets it as the *naming_spec*. If you want to use the sequence `%m` in the *filename*, specify the *naming_spec* first, and the *filename* second.

The optional second argument *naming_spec* is a quoted string that specifies how to name the symbols defined in the output file.

- *%t* is replaced with the name of the **typedef**, **struct**, or variable.
- *%m* is replaced with the name of the field (member).
- Other characters are echoed as you specify.

The default naming specification is `"%t_%m"`.

CAUTION Although argument *naming_spec* is optional, if used it must contain at least one instance of *%m*.

This pragma is useful when mixing assembly and C/C++ files in your application. For an example, see Finding Field Offsets on page 162.

For a pragma that produces the same output using **.set** directives for the assembler, see pragma `Asm_field_offsets`.

Ensure_instantiation(*T1*, *T2*,…*Tn*) — Ensure instantiation of specified template types

For C++ Only

Pragma `Ensure_instantiation` ensures instantiation of the specified template types, regardless of the setting of toggle `Auto_class_member_instantiation`. Arguments *T1*, *T2*, … *Tn* are template type identifiers.

NOTE To use pragma `Ensure_instantiation` successfully, you must also specify option **-Hnocomdat**.

For more information about templates and COMDAT, see the *MetaWare C/C++ Language Reference*.

Exclude_instantiation(*T1*, *T2*,…*Tn*) — Prevent instantiation of specified template types

For C++ Only

Pragma `Exclude_instantiation` prevents instantiation of the specified template types, regardless of the setting of toggle `Auto_class_member_instantiation`. Arguments *T1*, *T2*, … *Tn* are template type identifiers. For more information about templates, see the *MetaWare C/C++ Language Reference*.

Exit_expr(*Expression* [,*Level*]) — Evaluate the Expression argument at program termination

Pragma `Exit_expr` evaluates the *Expression* argument when a program terminates. *Level* specifies the order in which expressions are evaluated. Level is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (zero), the default. For more information, see Setting Module Initialization Priority on page 153.

Function_alignment(*align*, *func*, ...) — Align functions to a power of two

You can align one or more functions to a power-of-two boundary with pragma `Function_alignment(`*align*`, `*func*`, ...)`, where *align* is the power of two and *func* is the function name.

Global_aliasing_convention(["*Form*"]) — Construct external object names for aliasing internal variable and function identifiers

Pragma `Global_aliasing_convention` specifies automatic construction of external object names for internal identifiers. *Form* is a sequence of substring substitutions and designators; it must be enclosed in double quotes.

Specified without arguments, pragma `Global_aliasing_convention()` cancels `Global_aliasing_convention("`*Form*`")`.

For more information, see the following:

- Arguments to Pragma Global_aliasing_convention in Aliasing External Names on page 154
- Function-Naming Conventions on page 266
- Sharing Variables across Modules on page 269

Ident("*Comment*") — Insert a comment in the generated object file

Pragma `Ident` specifies a string to be inserted into the generated object module in the `.comment` section. If you specify option **-S**, the string is added with the **.ident** assembler directive. The comment string must be enclosed in double quotes. For more information, see Placing Comments in Object Code on page 162.

Initialization_level(*Level*) — Set initialization priority of modules

Pragma `Initialization_level` sets the initialization priority for a module to *Level*, a constant integral expression whose value can be between 0 (zero) and 255, inclusive. The highest priority is 255; the default value of *Level* is 0 (zero).

In MetaWare C++ code, the item with the highest initialization level is constructed first and destructed last.

For more information, see Setting Module Initialization Priority on page 153.

Interrupt_ext_words(*num_words*) — Provide space for saving extension state

Pragma `Interrupt_ext_words` provides extra space in the interrupt frame to save extension state. Argument *num_words* is the number of 32-bit words to reserve. Use this pragma in conjunction with compiler-intrinsic functions as described in Providing Space for Extensions to Save State on page 226.

Intrinsic(function_name [, *property*…]) — Define a compiler intrinsic for an instruction

Pragma `Intrinsic` lets you generate processor instructions from C. The compiler translates each call to the intrinsic function into a single instruction. You can use this pragma to enable use of both base-case and extension instructions as if they were C functions. You can then optimize these instructions as you would C functions. For more information on optimizing, see Chapter 7 — Optimizing Performance.

For ARCtangent-A4, *function_name* identifies the name of the intrinsic function. It must refer to a previous declaration of a function (with prototype). The declared function must take one or two parameters and return a result. Each parameter and return type must be representable with a single core register. If the function takes one parameter, the compiler assumes that the instruction is a two-operand instruction encoded using a sub-opcode of opcode 3. If the function takes two parameters, the compiler assumes the instruction is a three-operand instruction.

For ARCtangent-A5 and later, *function_name* identifies the name of the intrinsic function. It must refer to a previous declaration of a function (with prototype). The declared function must take zero, one, or two parameters and may optionally return a result. Each parameter and return type must be representable with a single core register. Functions that take no parameters are emitted as zero-operand instructions. Functions that take one parameter and do not return a result are emitted as instructions with a single source operand and no destination operand. Functions that take one parameter and return a result are emitted as two-operand instructions with the first operand being the destination. If the function takes two parameters, the compiler assumes the instruction is a three-operand instruction.

Properties

The optional comma-separated *property* arguments provide the compiler with information about the instruction. Specify each property using named-parameter-association syntax. The following properties are allowed:

`blocking_cycles =>` *n*

Specify this *property* if the instruction blocks execution by the processor for *n* cycles.

`effects => "`*register_effect*`[;`*register_effect*`...]"`

Specify this *property* if the instruction accesses registers in addition to those specified directly as its operands. Each *register_effect* substring describes a register side effect. The format of *register_effect* is as follows:

name`:`*effect*`[:`*effect*`...]`

- *name* is the name of the register affected by the side effect (with or without a leading '%'). *name* must be known to the compiler from a previous use of pragma `aux_register` or `core_register`, or from the base-case register set of the processor.
- *effect* may be one of the following:

`is_read`	Specify if register *name* is read as a side effect.
`maybe_read`	Specify if register *name* might be read as a side effect.

`is_written`	Specify if register *name* is written as a side effect.
`maybe_written`	Specify if register *name* might be written as a side effect.
`is_trashed`	Specify if register *name* is written with a value that is not intended to be used later.

As an alternative to *name*, the register may be specified numerically as follows:

`reg=`*N*`:`*effect*`[:`*effect*`...]`

or

`auxreg=`*N*`:`*effect*`[:`*effect*`...]`

- *N* is a compile-time constant identifying the register number.

`latency cycles =>` *N*

Specify this *property* if the instruction requires more than one cycle to complete before the destination register can be referenced without a stall occurring. *N* is the number of cycles required.

`name => "`*instr_name*`"`

Declares the instruction mnemonic to be *instr_name*. Enclose in double quotes. If you do not supply this property the compiler assumes the mnemonic is the same as *function_name*.

`opcode => Opcode_val`

Declares the instruction opcode (ARCtangent-A5 and later: major opcode) to be *Opcode_val* (hexadecimal or other integer constant). If you provide this property (and, for ARCtangent-A5 and later, the `sub_opcode` property), the compiler automatically generates the required directive to declare the extension instruction to the assembler.

`sub_opcode =>` *Sub_opcode_val* (ARCtangent-A5 and later)

Declares the instruction minor opcode to be *Sub_opcode_val* (hexadecimal or other integer constant). If you provide this property and the `opcode` property, the compiler automatically generates the required directive to declare the extension instruction to the assembler.

`flags =>` `["`*z*|*c*|*n*|*v*`"]`

Declares that the instruction sets the given flags when the `.f` suffix is used. You can use any of the characters *z*, *c*, *n*, and *v* in any order, enclosed in double quotes.

`assume_volatile => 1`

Declares that the instruction has side effects that may interfere with surrounding instructions, such as when an extension instruction modifies state other than the destination register. Be sure to specify `assume_volatile => 1` if the instruction should not be moved past other extension instructions, auxiliary register accesses, and the like during scheduling and delay-slot filling.

`set_flags => 1`

Specifies that the intrinsic should always generate an instruction to set the flags (use the `.f` suffix with the assembly instruction).

If you do not want to set the flags every time you use the instruction, you must define two separate intrinsics for the instruction: one that sets the flags and one that does not. Even if you define an

intrinsic that does not set the flags, the compiler might cause the flags to be set if you specified the `flags` property and the compiler determines that it can make use of setting the flags.

NOTE We recommend you limit the use of pragma `Intrinsic` to extension instructions that have no side effects and store their results in the single register you provide as the first operand. For instructions with more complex behavior, it is important to specify the side effects and cycle requirements of the instruction using the provided *properties* arguments. If the instruction cannot easily be described using the provided *properties* arguments, enhanced inline assembly macros are recommended, although they inhibit optimization.

On ARCtangent-A5 and later, you can turn off toggle `Compress_asm_warn` to disable warnings about inline assembly macros inhibiting optimization.

For more information, refer to Defining Intrinsics for Extensions Using C/C++ on page 306 and Using and Optimizing Extensions in C/C++ on page 164.

Literals("*name*") — Name a section for storing literals and static variables

Pragma `Literals` specifies the name of the section to contain literals and read-only static variables declared with the **const** type qualifier. The name of the default literals section is operating-system dependent. The *name*, a string constant, must be enclosed in double quotes. Do not use as your section name any of the default section names listed in Table 13 Default Control Sections for Code and Data on page 252.

See Specifying an Alternate Literals Section on page 161. See also toggle `Read_only_strings` in Chapter 5 — Using Compiler Toggles.

Loop_factor(*Loop_count*, *Size*) — Unroll loops

Pragma `Loop_factor` optimizes code by unrolling loops; that is, it replaces a loop with the code that makes up the loop body, replicated some number of loop iterations.

- *Loop_count* is an integer constant specifying the maximum number of times a loop is unrolled; its default depends on the number of instructions in the loop.
- *Size* is an integer constant indicating the maximum size of loops to be unrolled (approximate number of instructions).

NOTE Using this pragma does not guarantee unrolling of the targeted loops. For more information.

CAUTION This pragma is available only at optimization level 7. Exercise caution when using this pragma. The ARC processor's zero-delay-loop mechanism generally gives better results, unless your code makes numerous memory references. Best practice is to build with and without this pragma and compare size and performance.

Naked(*func1*[,*func2*,...*funcN*]) — Do not save or restore prolog or epilog

Pragma `Naked` tells the compiler to omit prolog and epilog save and restore sequences for the function(s) specified. Instead, the compiler saves and restores the return address of the specified functions if they make a call.

The *func* arguments may be one of the following:

- Simple function names (for C++, the function or functions must not be overloaded)
- Fully qualified C++ function names

For more information, see Eliminating Prolog and Epilog Overhead on page 154. To disable warnings for functions that do not return, turn off toggle `Check_function_returns`.

Never_returns(*func*[,*func2*,...*funcN*]) — Designate function as never returning

Pragma `Never_returns` indicates to the compiler that the specified function is guaranteed not to return. The compiler can then save overhead by not saving and restoring the function's prolog and epilog (as with pragma `Naked`), and also its return address.

The *func* arguments may be one of the following:

- Simple function names (for C++, the function or functions must not be overloaded)
- Fully qualified C++ function names

For more information, see Eliminating Prolog and Epilog Overhead on page 154. To disable warnings for functions that do not return, turn off toggle `Check_function_returns`.

Off(*Toggle_name*) — Turn off a toggle

Pragma `Off` turns off the specified compiler toggle. For information about individual toggles, see Using Toggles on page 86.

Off_inline(*function_name*[, *function_name* ...]) — Turn off inlining for specific functions

Pragma `Off_inline` prevents calls to *function_name* from being inlined when they otherwise would be. Related pragmas are `On_inline` and `Pop_inline`.

For more information, see Using Pragmas to Inline Individual Routines on page 185.

Offwarn(*n1*, *n2*, ...) — Disable specified warning messages

Pragma `Offwarn` disables specified compiler warning messages so they are not displayed. The arguments (*n1*, *n2*, ...) are the numbers of the warning messages. Related pragmas are `Onwarn` and `Popwarn`. For more information, see Controlling Individual Warnings by Number on page 339.

To find the warning numbers, compile with toggle `Msgno` turned on (the default) and note them.

On(*Toggle_name*) — Turn On a toggle

Pragma `On` turns on the specified compiler toggle. For information about individual toggles, see Alphabetical List of Compiler Toggles on page 86.

On_inline(*function_name*[, *function_name* ...]) — Turn on inlining for specific functions

Pragma `On_inline` causes calls to *function_name* to be inlined even if *function_name* does not otherwise meet the criteria for inlining. Related pragmas are `Off_inline` and `Pop_inline`. For more information, see Using Pragmas to Inline Individual Routines on page 185.

> **NOTE** This pragma is available at optimization levels 4 to 7 and s, or if you manually invoke the outboard inliner (option **-Hi**). It does not work at an optimization level less than 4. For more information, see Chapter 8 — Inlining.

Option **-g** inhibits inlining.

C++ Only

Pragma `On_inline` is designed for simple C functions; it has no effect on C++ functions that require a modifier.

Onwarn(*n1*, *n2*, …) — Enable specified warning messages

Pragma `Onwarn` enables specified compiler warning messages so they are displayed. The arguments (*n1*, *n2*, …) are the numbers of the warning messages. Related pragmas are `Offwarn` and `Popwarn`. For more information, see Controlling Individual Warnings by Number on page 339.

Onwarn_error(*n1*, *n2*, …) — Enable specified warning messages as compiler error messages

Pragma `Onwarn_error` enables specified compiler warning messages, so they are displayed, and upgrades them to be compiler errors. The arguments (*n1*, *n2*, …) are the numbers of the warning messages. For more information, see Controlling Individual Warnings by Number on page 339.

To find the warning numbers, compile with toggle `Msgno` turned on (the default) and note them.

Warnings upgraded using pragma `Onwarn_error` have the full status of errors, including appearing in the error count and aborting the compilation.

> **NOTE** Pragma `Onwarn_error` affects only warning messages generated by the compiler front end, not the assembler or linker.

Pack([*n*]) — Control default alignment of data structures

Pragma `Pack` is synonymous with pragma `Align_members` — it controls the default alignment of data structures.

- Pragma `Pack(1)` packs a **struct** so that no padding is inserted between members.
- Pragma `Pack()` "unpacks" the data; that is, it restores the alignment to the default ("natural") alignment of data structures on the target machine, as specified by the application binary interface.

Related pragmas are `Align_members` and `Push_align_members`. For more information, see Aligning Data Structures on page 150.

Page(*n*) — Insert page ejects in a source-code listing

Pragma `Page` causes page ejects to be inserted into a source-code listing, where *n* is the number of form feeds (pages to eject). This pragma takes effect only when toggle `List` is on. For more information, see Customizing the Listing with Pragmas on page 298.

Pop(*Toggle_name*) — Reinstate prior status of a toggle

Pragma `Pop` reinstates a prior status of a compiler toggle. For information about individual toggles, see Alphabetical List of Compiler Toggles on page 86.

Pop_align_members — Restore alignment prior to earlier Push_align_members

Pragma `Pop_align_members` cancels a previous pragma `Push_align_members` and restores the prior alignment. For more information, see Restoring Structure Alignment on page 151.

Pop_ignore(*Pragma_name*(*Pragma_arg*), …) — Reinstate prior ignore status of listed pragmas

Pragma `Pop_ignore` restores the compiler's previous state of awareness of any pragma listed as one of its arguments. The names of pragma arguments can be listed with or without quotes. Once a pragma has been set, you can make the compiler ignore that pragma or become "unaware" of it by listing the pragma as an argument of pragma `Push_ignore`. The compiler issues a warning if you "un-ignore" a pragma when the compiler is already aware of that pragma.

Pop_inline(*function_name*[, *function_name* …]) — Pop top-most stack entry for each specified function

Pragma `Pop_inline` pops the top-most stack entry for each function specified, and restores the previous pragma `On_inline` or `Off_inline` state.

For more information, see Using Pragmas to Inline Individual Routines on page 185.

Pop_small_data() — Reinstate prior small-data specification

Pragma `Pop_small_data` pops the small-data specification set by the last `Push_small_data` pragma, and reinstates the previous specification.

These are related compiler controls:

- options **-Hall_sdata**, **-Hnosdata**, and **-Hsdata0**
- toggles `Unsized_arrays_in_small_data`, `Use_sdata`, and `Common_in_small_data`

For more information, see Using Small-Data Sections on page 214.

Popwarn — Cancel previous pragma Offwarn or Onwarn

Pragma **Popwarn** cancels the previous pragma `Offwarn` or `Onwarn`; these pragmas are "stacked." For more information, see Controlling Individual Warnings by Number on page 339.

Push_align_members(*n*) — Set maximum boundary for structure-member alignment and save state

Pragma `Push_align_members` specifies that *n* bytes is the maximum boundary for structure member alignment, then saves the state for a subsequent call to pragma `Pop_align_members`.

To save the current state for pop, then restore the default, specify the pragma as follows:

```
#pragma Push_align_members(_DEFAULT_MEMBER_ALIGNMENT)
```

Related pragmas are `Align_members` and `Pack`. For more information, see Aligning Structures and Saving Maximum Alignment on page 151.

Push_ignore(*Pragma_name*, *Pragma_name* …) — Ignore pragmas listed

Pragma `Push_ignore` causes the compiler to ignore (that is, lose awareness of) any previously set pragma listed as one of its arguments. It also suppresses any warning message the compiler would otherwise emit about the pragma.

Pragma `Push_ignore` is useful mainly for ignoring pragmas not supported by the MetaWare C/C++ compiler. The names of pragmas to ignore can be listed with or without quotes. For more information, see pragma `Pop_ignore` and Behavior on Encountering Pragma Directives on page 259.

Push_small_data(*n*[;*m*]) — Assign data to small-data section

> **NOTE** The default configuration of the compiler automatically turns on pragma `Push_small_data(4;4)`.

Pragma `Push_small_data` establishes a small-data specification for subsequently declared non-**auto** variables. Pragma `Push_small_data` specifies the small-data sections to which the compiler will assign variables declared in a given block of code and the maximum size of variables in the specified small-data sections.

> **NOTE** The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** into small data unless you qualify them as **_Near**.

The `Push_small_data` syntax allows you to set independent specifications for writable and read-only data.

- *n* is an integer representing the maximum size in bytes of writable (non-**const**) data.
- *m* is an integer that represents the maximum size in bytes of read-only data (such as **const** data and string literals).

Pragma `Push_small_data` "pushes" the specification for small data processing, and pragma `Pop_small_data` pops the current specification, reinstating the previous state of small-data processing.

> **CAUTION** Pragma `Push_small_data` can move **const** variables from a read-only data section to a read-write small-data section. If you prefer the **const** variables to remain in a read-only section, you must use 0 (zero) as the *m* argument. For example:
> `-Hpragma="Push_small_data(8;0)"`

Related Compiler Controls

- Options **-Hall_sdata**, **-Hnosdata**, and **-Hsdata0**
- Toggles `Unsized_arrays_in_small_data`, `Use_sdata`, and `Common_in_small_data`

Related Topic

- Using Small-Data Sections on page 214

Reg_remove(*regs*) — Remove one or more registers from use

Pragma `Reg_remove` removes the register (or registers) you specify in argument *regs*. Argument *regs* may be a comma-separated list of register numbers, or a range. If you wish to specify a range, you must place the entire argument in quotes, for example:

```
pragma reg_remove("20,22-24");
```

> **NOTE** The compiler might not permit a register to be removed if it deems that register essential.

Pragma `Reg_remove` is equivalent to the following:

- Pragma `Core_register` with *`usage => removed`*
- Option **-Hreg_remove**

CAUTION If you use an option or pragma to remove a core register, you must compile your entire application with the same option or pragma. This includes the C run-time libraries. For information on rebuilding the libraries, see Rebuilding the C Run-Time Libraries on page 241.

To prevent the compiler from using a register and reserve the register for your own use, use pragma `Reg_reserve`.

Reg_reserve(*regs*) — Reserve one or more registers for application-specific use

Pragma `Reg_reserve` reserves the register (or registers) you specify in argument *regs*. Argument *regs* may be a comma-separated list of register numbers, or a range. If you wish to specify a range, you must place the entire argument in quotes, for example:

```
pragma reg_reserve("20,22-24");
```

NOTE The compiler might not permit a register to be reserved if it deems that register essential.

When it reserves the register, the compiler never generates code that uses that register. You may then use the register freely for application-specific purposes.

Pragma `Reg_reserve` is equivalent to the following:

- Pragma `Core_register` with *`usage => reserved`*
- Option **-Hreg_reserve**

CAUTION If you use an option or pragma to reserve a core register, you must compile your entire application with the same option or pragma. This includes the C run-time libraries. For information on rebuilding the libraries, see Rebuilding the C Run-Time Libraries on page 241.

Example Uses

- To reserve a register for use as a small-data base register for your own small-data sections (see Adding Your Own Small-Data Sections on page 218).
- To use registers in assembly-language interrupt handlers without needing to save and restore them.

You can use this pragma to replace a frequently-accessed global variable, and access the "variable" using the **_core_read()** and **_core_write()** intrinsics. However, direct register assignment provides a means for doing so without using intrinsics. For more information, see Direct Register Assignment for Variables on page 270.

CAUTION Do not confuse this pragma with `Reg_remove`, which removes the register from any use.

Skip(*n*) — Insert blank lines in a source-code listing

Pragma `Skip` causes blank lines to be inserted into a source-code listing, where *n* is the number of blank lines to insert. This pragma takes effect only when toggle `List` is on. For more information, see Customizing the Listing with Pragmas on page 298.

Startup_expr(*Expression*[, *Level*]) — Evaluate expression at startup

Pragma `Startup_expr` evaluates the *Expression* argument at run-time startup, before **main()** is invoked. If you use pragma `Startup_expr`, the compiler generates an initialization routine when it generates the object file. *Level* specifies the order in which expressions are evaluated. *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (the default). For more information, see Evaluating Expressions out of the Scope of main() on page 153.

Static_segment("*Section_name*") — Name a section for storing initialized static variables

Pragma `Static_segment` alters the name of the default data section. (The name of the default data section is operating-system-dependent.) *Section_name*, a string constant denoting the section name, must be enclosed in double quotes. Do not use as your section name any of the default section names listed in Table 13 Default Control Sections for Code and Data on page 252.

STDC *trait* [*ON* | *OFF* | *DEFAULT*] — Toggle a trait

Pragma STDC allows you to specify three states for *trait*: *ON*, *OFF*, and *DEFAULT*. The preprocessor does no macro replacement after encountering the STDC token.

The argument *trait* can be any of the compiler toggles in Alphabetical List of Compiler Toggles on page 86. Compare pragmas `On` and `Off`.

Title("*string_c*") — Put title at the top of each page of the source listing

Pragma `Title` causes a title to appear at the top of each page of the source listing. This pragma takes effect only when toggle `List` is on. The title *string_c*, a string constant, must be enclosed in double quotes. For more information, see Customizing the Listing with Pragmas on page 298.

Warning_level(*n*) — Set the level of warning diagnostics

Pragma `Warning_level` sets the level of warning diagnostics to *n*, where *n* is a decimal integer between `0` (zero) and `4`, inclusive. For more information, see Controlling Groups of Warnings by Level on page 338.

Weak {x = y|(x,y)|x} — Designate a weak global symbol and assign it a value

Pragma `Weak` designates identifier *x* as a weak global symbol whose value is the same as identifier *y*. Identifier *y* must be a global symbol defined in the current module with *x* otherwise undefined.

These statements are all equivalent:

```
#pragma Weak x
#pragma Weak(x,y)
#pragma Weak x = y
```

You can use this pragma to access a function that may not exist and resolve to a default function if that is the case.

Customizing Your Program with Pragmas

This section provides more information about using compiler pragmas to perform the following types of customizing:

- Aligning Functions and Data on page 149
- Controlling Initialization, Startup, and Exit on page 153
- Eliminating Prolog and Epilog Overhead on page 154
- Aliasing External Names on page 154
- Specifying an Alternate Static Data Section on page 157
- Specifying an Alternate Code Section on page 159
- Specifying Your Own BSS Section on page 158
- Specifying the Name of the Default BSS Section on page 160
- Specifying an Alternate Data Section on page 160
- Specifying an Alternate Literals Section on page 161
- Accessing a Common Block from Assembly on page 161
- Placing Comments in Object Code on page 162
- Finding Field Offsets on page 162
- Defining Registers on page 163
- Using and Optimizing Extensions in C/C++ on page 164
- Using Extension Condition Codes in C/C++ on page 166

Aligning Functions and Data

You can use compiler pragmas to align functions and data as described in the following sections.

Aligning Functions

You can align one or more functions to a power-of-two boundary with pragma `Function_alignment`(*align*, *func*, `...`), where *align* is the power of two and *func* is the function name.

Aligning User-Defined Storage Blocks

You can specify minimum section alignment when you use pragma `Data` to define named blocks for storage. Example 12 Aligning a Named Block for Storage defines a section `my_data` and ensures it is aligned at no less than a 32-byte boundary.

Example 12 Aligning a Named Block for Storage

```
#pragma Data("my_data", align => 32)
```

Specifying the Alignment of Global or Static Variables

Use pragma `Align_to` to specify the alignment of global or static variables.See the listing for pragma `Align_to` in Alphabetical List of Pragmas on page 129 for important usage restrictions.

Example

The following example shows the usage of pragma `Align_to`:

```
int stack[1024];
#pragma align_to(8,stack)
```

Aligning Structure Members

Use pragma `Align_members` to specify how members of subsequently declared structures are to be aligned. This pragma has the following form:

```
#pragma Align_members(n)
```

n is a decimal integer constant denoting in bytes the maximum alignment boundary to be applied to a member.

The default setting for pragma `Align_members` is `4`.

Example

Here is an example use of pragma `Align_members`:

```
#pragma Align_members(2)
struct s {
   char c;     /* Offset 0, one-byte padding  */
   int x;      /* Offset 2                    */
   };
```

Member `x`, which is type **int**, would ordinarily be aligned to a four-byte boundary. But because of the pragma `Align_members(2)`, `x` is aligned to a two-byte boundary instead.

> **NOTE** The default alignment of data structures in a program is the natural alignment specified by the application binary interface.

Note that pragma `Align_members` denotes the maximum boundary; an **int** member is never aligned to greater than a four-byte boundary regardless of how large a value is specified for *n*.

You can also use the MetaWare C/C++ type qualifier **_Aligned** for aligning data structures.

Packing a Struct

To pack a **struct** so that no padding is inserted between members, specify a setting of `1`:

```
#pragma Align_members(1)
```

> **CAUTION** If you use pragma `Align_members` indiscriminately, you can severely impair run-time performance by trying to access **struct** members that are not properly aligned.

Aligning Data Structures

Pragma `Pack` is synonymous with pragma `Align_members` — it controls the default alignment of data structures.

- Pragma `Pack(1)` packs a **struct** so that no padding is inserted between members.
- Pragma `Pack()` "unpacks" the data; that is, it restores the alignment to the default ("natural") alignment of data structures on the target machine, as specified by the application binary interface.

> **NOTE** Pragmas `Pack(n)` and `Pack()` are not stackable — that is, `Pack()` does not revert to the previous `Pack(n)` — so these pragmas cannot be nested. For nesting, use pragma `Push_align_members` and `Pop_align_members` pairs.

For an example of when not to use pragma `Pack(1)...Pack()` pairs, see Aligning Structures and Saving Maximum Alignment on page 151.

Aligning Structures and Saving Maximum Alignment

Pragma Push_align_members(*n*) has the same effect as pragma Align_members(*n*) or pragma Pack(*n*), except that Push_align_members saves the current setting of the maximum **struct** alignment.

The following code sample illustrates why we recommend the use of pragma Push_align_members(1) and Pop_align_members pairs (because they are stackable) rather than pragma Pack(1) and Pack() pairs (which are not stackable).

```
                                / * Pack structure */
#pragma Push_align_members(1)           /* Push #1 */
/* --- Begin included header file ---  */
/* ... */
             /* Align to machine's natural boundary */
#pragma Push_align_members(4)           /* Push #2 */
struct st_1 { char a, int b, char c; }
                    /* Revert to previous alignment */
#pragma Pop_align_members()             /*  Pop #2 */
/* --- End included header file --- */
struct st_2 { char a, int b, char c; }
                    /* Revert to previous alignment */
#pragma Pop_align_members()             /*  Pop #1 */
```

In this code sample, structure st_1 is naturally aligned, while structure st_2 is packed (aligned to 1).

If you replace the Push_align_members(n) ... Pop_align_members() pairs with Pack(n) ... Pack() pairs, as shown in the following code sample, structures st_1 and st_2 will both be naturally aligned.

```
                                /* Pack structure */
#pragma Pack(1)                         /* Push #1 */
/* --- Begin included header file --- */
/* ... */
#pragma Pack(4)                         /* Push #2 */
struct st_1 { char a, int b, char c; }
           /* Does not revert to previous alignment */
#pragma Pack()                          /*  Pop #2 */
/* --- End included header file --- */
struct st_2 { char a, int b, char c; }
           /* Does not revert to previous alignment */
#pragma Pack()                          /*  Pop #1 */
```

This natural alignment occurs because the alignment setting is changed in the included header file, and Pack() does not revert to the previous alignment setting (but Pack() always restores the alignment setting to the natural alignment setting of the target machine).

Restoring Structure Alignment

Pragma Pop_align_members cancels Push_align_members and restores the alignment that existed before the preceding **Push_align_members**.

If there is no pending Push_align_members, the alignment is restored to the default. (The default alignment of a **struct** is defined in Structures, Unions, Enumerations, and Bit Fields on page 255.)

Accessing Misaligned Data

You can save space by packing structures (using pragma Pack, pragma Align_members, and the **_Packed** keyword). However, packing results in misaligned data, which can be expensive to access.

Processors made by ARC® International do not permit misaligned memory accesses, and the compiler does not generate misaligned references. The compiler inserts padding in structures to align data appropriately. If you want to suppress such padding, use pragma Pack. If you use pragma Pack, the

compiler generates whatever piecemeal aligned accesses it requires to handle the otherwise misaligned data.

TIP Misaligned accesses slow execution speed.

The MetaWare debugger simulator issues an error message if a program attempts to perform a misaligned access.

CAUTION The ARC architecture requires 4-byte alignment for 32-bit loads and stores and 2-byte alignment for 16-bit loads and stores. On hardware, your application might access incorrect data in the aligned location when the actual data is unaligned. When the compiler knows an access is unaligned, it uses load/store shift operations, which might be inefficient, to correct alignment.

The MetaWare C/C++ compiler can usually determine that an object is misaligned and generate code that takes the alignment into account. There are situations, however, where perfectly valid code obscures the alignment of the members of packed structures. In particular, passing the addresses of misaligned members to functions expecting aligned pointers, as in the following code snippet, can cause difficulty.

```
_Packed struct { char a; int b; } s;
void func(int *p)     { *p = 10; }
main()     { func(&s.b); }
```

This usually causes a memory fault because there is no way for the compiler to know that the argument to **func()** is misaligned without assuming that all pointer dereferences are misaligned. To provide a means to inform the compiler that a particular pointer should be assumed to be misaligned, MetaWare C/C++ provides the **_Unaligned** keyword, documented in the *MetaWare C/C++ Language Reference*.

Meaning of _Packed in Legacy Code

Some compilers have overloaded the **_Packed** keyword to mean **_Unaligned** as well. The two qualifications are better kept semantically distinct because you may encounter situations where you need one of the qualifications but not the other. For instance, if a structure type is packed yet is never embedded within another packed object, it is always aligned, as in the following example:

```
_Packed struct stype { char a; short b; char c; } s;
```

While this structure is packed, it is not necessarily misaligned. But if a pointer to this struct is passed to a routine and dereferenced, the compiler should be able to determine that even though the **struct** type is packed, it is not misaligned, and dereference the entire struct as an aligned word. This is not possible for compilers that have overloaded the meaning of **_Packed**. Such compilers must always assume that **_Packed** also means unaligned.

To facilitate porting code from compilers that have overloaded the meaning of **_Packed**, the MetaWare C/C++ compiler provides toggles `Packed_means_unaligned` and `Unaligned_means_packed` (see Alphabetical List of Compiler Toggles on page 86). You can use these toggles along with macros to redefine the **_Packed** keyword into either **_Packed** or **_Unaligned**, as appropriate, so that legacy code can take advantage of the distinction.

Controlling Initialization, Startup, and Exit

Setting Module Initialization Priority

Pragma `Initialization_level`(*Level*) sets the initialization level for the module to the specified value. This can be useful, for example, in ordering the construction of static instances of a class that you want to be constructed before **main()** is called.

The program startup code calls each of the initialization routines in order of decreasing initialization level, where *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. If you do not specify *Level*, the compiler assumes a value of 0 (zero) — the lowest priority.

NOTE Initialization priorities 120 to 130 and 250 to 255 are reserved for internal use.

Evaluating Expressions out of the Scope of main()

You can specify expressions to be evaluated at program startup or termination with pragmas `Startup_expr` and `Exit_expr`. If you specify `Startup_expr` or `Exit_expr`, the compiler generates an initialization or termination routine, respectively.

Use these initialization pragmas if you want some code to be executed before **main()** is called or after **main()** exits. Such code could perform initialization of data structures and could free memory to undo the initialization process.

If data items must be initialized at run time, or expressions must be evaluated at startup or termination, the compiler generates an initialization routine when it generates an object module.

Pragma Startup_expr

Pragma `Startup_expr`(*Expression*, *Level*) evaluates the expression argument at run time before **main()** is invoked.

The order in which expressions are evaluated is specified by the integer constant expression *Level*. If you do not specify *Level*, the compiler uses either the initialization level of the entire compilation unit, or the default value 0 (zero). (Use pragma `Initialization_level` to set a compilation unit's initialization level, which is zero by default.)

Typically, the *Expression* argument is a function call:

```
void initialize_this_module() { ... }
#pragma Startup_expr( initialize_this_module(), 10 )
```

With MetaWare C/C++, the full power of the expression is available, and can be used to disambiguate between two functions or to execute other non-function code at startup:

```
void init(int), init(double); int x, y;
/*
 * C++ code here
 */
#pragma Startup_expr( (init(3), init(2.71),
                       x = 4,   y = x*x), 10)
```

To specify more than one expression, enclose them in parentheses — or use multiple pragmas.

Pragma Exit_expr

Pragma `Exit_expr`(*Expression*, *Level*) evaluates the *Expression* argument when the program terminates, at the time **atexit()** gains control.

The integer constant expression *Level* specifies the order in which expressions are evaluated. If you do not specify *Level*, the compiler uses the default value 0 (zero).

Eliminating Prolog and Epilog Overhead

The compiler provides pragmas that can help optimize applications by eliminating the overhead associated with prolog and epilog, or prolog, epilog, and return address.

Pragma Naked

Pragma `Naked` eliminates the overhead associated with saving and restoring function prologs and epilogs. Candidates for use of this pragma include the following:

- **main()** in almost all cases. One exception might be for an RTOS that uses a startup file not supplied by ARC® International

> **NOTE** Modifying **main()** in this way might render a program non-ABI-conforming, because the program can no longer be executed from a free-standing environment with the guarantee that it follows the ABI.

- A C signal handler that calls an interrupt-service routine that itself saves and restores all registers
- A function that never returns, or has a call to a function that never returns (if the call to a function that never returns is guaranteed to be made)
- Functions for which you supply your own prolog and epilog using inline assembly code

Pragma `Naked` is identical to Microsoft's **`__declspec(naked)`** and GNU's **__attribute__(*func_name*,naked)**.

Pragma Never_returns

Pragma `Never_returns` specifies a function as never returning, eliminating prolog and epilog save/restore and also eliminating save/restore of the function's return address to allow additional optimizations. Pragma `Never_returns` is appropriate for functions such as **exit()** or **abort()**.

Processing and Placement of the Pragmas

The pragmas discussed in this section are processed at the end of the source file, after all function definitions have been seen. No warnings or errors are issued if the specified functions are not found, which allows the pragmas to be placed in many source files, regardless of whether or not the functions are in a specific file.

Disabling Warnings

To disable warnings for functions that do not return, see toggle `Check_function_returns`.

See Also

- Pragma `Naked`
- Pragma `Never_returns`
- Calling convention attribute **_NEVER_RETURNS**

Aliasing External Names

An *aliasing convention* is a method for aliasing internal names to external names. The names of variables and functions that are communicated across module boundaries are normally made global in the resultant object module. In large programs there can be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately, C compilers and most linkers do not provide for a structured name space — for named packages of resources, for example. Thus the internal names in a program might not be usable as

external names (those known to the linker). Some method of aliasing internal names to external names is needed, and MetaWare C/C++ provides it.

It is important to be able to alias such names to avoid conflicts in the linker's external symbol dictionary, rather than being forced to pervert the internal names themselves. The internal names must be well-chosen "containers of meaning," for program maintainability.

The MetaWare C/C++ compiler provides two pragmas for aliasing names: `Alias` and `Global_aliasing_convention`.

Pragma Alias

Pragma `Alias` specifies, for a given internal name, another name for external or public purposes, as follows:

`#pragma Alias(`*Internal_name*`, "`*External_name*`")`

External_name does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker).

NOTE The `Alias` pragma must appear in the scope of the declaration of the internal name.

Example

This example illustrates the use of pragma `Alias` to specify alternate, external names for function **Initialize()** and **int** `BA`:

```
void Initialize();
#pragma Alias(Initialize,"x_initialize")
...
int BA;
#pragma Alias(BA,"Ph.D.")
...
```

In this example, function **Initialize()** is referenced in the object-module symbol table as `x_initialize` and `BA` is referenced in the symbol table as `Ph.D`.

Pragma Global_aliasing_convention

Pragma `Global_aliasing_convention` specifies the aliasing of external names not already aliased by the `Alias` pragma, as follows:

`#pragma Global_aliasing_convention("`*format*`")`

The *format* argument is a string that specifies the way each external name is derived from its internal name.

The (derived) external name is the actual text of *format*, except for the following substitutions for substrings within *format*:

- `%r` (resource) denotes *Internal_name*
- `%%` denotes `%`
- `%C`, `%c`, and `%a` affect the casing of the external name

For example, if you specify pragma `Global_aliasing_convention` as follows, a function named **fred** will be known as **FRED** in the object module:

```
#pragma Global_aliasing_convention("%C%r")
```

See the section Arguments to Pragma Global_aliasing_convention on page 156 for additional information.

By default, the `Global_aliasing_convention` setting is `%r`; external names match internal names.

Changing the Default Convention

You can change the default aliasing convention by modifying the driver configuration file to supply a value for the **-Hgac** option on the `ARGS=` line. If you set the `Global_aliasing_convention` by configuring the driver, all external names (explicitly declared or otherwise) are affected.

NOTE Changing the default aliasing convention has implications for the use of existing libraries.

If you set the aliasing convention in the source code or in a `.pro` file, only subsequently declared names are affected. Functions that are used but not declared are not affected by the convention.

You can override the `Global_aliasing_convention` setting for a particular name by specifying pragma `Alias`; that is, pragma `Alias` takes precedence.

You can turn off pragma `Global_aliasing_convention` by omitting the *format* argument, like this:

```
#pragma Global_aliasing_convention()
```

This restores the default (`%r`).

Arguments to Pragma Global_aliasing_convention

The `%r` argument can be followed by substring designators of the form `:`*Start*`:`*Len* to indicate the substring starting at character *Start* and going for *Len* characters. The character positions are numbered starting at 1 (one).

- If *Start* is negative, it denotes the starting position from the right end of the string where the last character is numbered -1.
- If you omit `:`*Len*, the substring means "to the end of the string."

These substitutions within *format* affect the casing of the external name:

- `%C` means to shift subsequent letters in the external name to uppercase.
- `%c` means to shift them to lowercase.
- `%a` means to use subsequent letters as is, each with its given case (that is, do not shift case). This mode holds at the beginning of the specification.

Example

This example changes all global names (`%r`) to uppercase (`%C`) and truncates them to eight characters (`:1:8`):

```
#pragma Global_aliasing_convention("%C%r:1:8")
```

The same example would appear in the configuration file as follows:

```
ARGS= -Hgac=%C%r:1:8
```

See Chapter 4 — Using Driver Options and Appendix B — Configuring the Driver for details about **-Hgac** and the configuration file.

Specifying an Alternate Static Data Section

NOTE Read this section if you want to control the mapping of static storage or use a data-communication convention that does not conform to the ANSI C Standard.

Pragma Data explicitly specifies the name of a section or FORTRAN-style common block in which to map static variables. The following section types are supported:

- *BSS* indicates an uninitialized static-data section.
- *DATA* indicates an ordinary read/write data section. This is the default if no explicit section type is specified.
- *LIT* indicates a read-only data section. All variables mapped into a LIT section must be **const** qualified.
- *TEXT* indicates a read-only executable section.
- *COMMON* indicates a FORTRAN-style common block. All variables mapped into a COMMON section must be **static**. (See FORTRAN-Style Common Blocks on page 158.)

Pragma Data has two forms, which must be used in pairs to "bracket" the affected declarations, as shown here:

```
#pragma Data(section_type, "section_name")
/*
 * Static and global variable declarations go here,
 * such as:
 *
 *  static int a, b, c;
 * ...
 */
#pragma Data()          /* Terminates specification */
```

section_type is one of the supported types: *BSS*, *DATA*, *LIT*, *TEXT*, or *COMMON* (casing is insignificant). If you do not specify a *section_type*, the section defaults to DATA.

A Data pragma with no parameters terminates the specification. Each Data pragma must be terminated or "turned off" in the same scope in which it is turned on.

A Data pragma can be used within the scope of a function, in which case all affected variable declarations default to **static** (instead of to **auto**). A Data pragma that is active across a function definition does not apply to those variables that are local to the function. That is, the Data pragma is suspended during the scope of the function definition.

Data pragmas cannot be nested. That is, you cannot use a Data pragma to specify a new section while a preceding Data pragma is still active. The compiler issues a warning if you specify a Data pragma when a prior Data pragma is still active (in which case the subsequent pragma applies). The compiler also issues a warning if a Data pragma is active at the end of a function declaration or at the end of a compilation unit.

Examples

This example maps global **int** variables a and b and **static** array c into a user-defined DATA section named MY_DATA.

```
#pragma Data("MY_DATA")   /* Defaults to type DATA */
int a, b;
```

```
static char c[10];
#pragma Data()
```

This example maps two variables into a user-defined read-only LIT section named `MY_LIT`.

```
#pragma Data(LIT,"MY_LIT")
const char string[] = "A read-only string";
const int *p = &i;           /* A read-only pointer */
#pragma Data()
```

NOTE All variables mapped into a LIT section must be **const** qualified.

Specifying Your Own BSS Section

To specify your own BSS section for static variables that are not initialized at compile time, use pragma `Data` with *BSS* as the first parameter. Example 13 Specifying Your Own BSS Section creates user-defined section `my_bss`, which is of type BSS, or uninitialized static data.

Example 13 Specifying Your Own BSS Section

```
#pragma Data(BSS, "my_bss")
```

Compare pragma BSS, which specifies the name of the *default* BSS section that the compiler uses for mapping such data.

FORTRAN-Style Common Blocks

The section type `COMMON` defines FORTRAN-style common blocks.

Static variables defined with the COMMON block are mapped to consecutive offsets within the block, subject to boundary alignment. For example:

```
#pragma Data(COMMON, "BLOCK")
static  int   a;    /* Offset  0 in section "BLOCK" */
static  int   b;    /* Offset  4 in section "BLOCK" */
static  char  c;    /* Offset  8 in section "BLOCK" */
static  char  d;    /* Offset  9 in section "BLOCK" */
static  short e;    /* Offset 10 in section "BLOCK" */
#pragma Data()
```

If this block of code were included in separate modules, variables `a`, `b`, `c`, `d`, and `e` would be shared, even though none of these variables is explicitly declared to be **global**.

NOTE The name space of the common blocks is in the same name space as other symbols. (Section names are ordinarily in a different name space from the names of ordinary symbols.)

For more information, see Accessing a Common Block from Assembly on page 161.

NOTE Because variables defined within common blocks are actually implemented as offsets from the base of the block, the variables must be **static**. Global variables within COMMON blocks are not supported. Likewise, variables within COMMON blocks must not be initialized at compile time. The compiler issues error messages if code violates these rules.

Specifying an Alternate Code Section

Code segmentation means grouping code into named control sections that are manipulated by a linker. The concept of segmenting generated code is useful whether or not a program is modularized and separately compiled.

The concept applies only to top-level or "level-one" functions; level-two and greater functions (that is, nested functions) must be in the same section as their containing level-one function. Code for one function cannot be split across sections.

NOTE Nested functions are a MetaWare C language extension; ANSI Standard C permits only level-one functions.

By default, compiler-generated code is placed in the `.text` section. The default code section name can be overridden by using the `Code` pragma. Pragma `Code` explicitly specifies the name of the section in which the code of the subsequent functions is to reside.

Pragma `Code` has two forms that must be used in pairs to bracket the affected function definitions:

```
#pragma Code(Section_name)
/* ----- Affected function definitions go here ---- */
#pragma Code()                /* No parameters here */
```

Section_name is a constant string expression that denotes the name of the section.

The closing pragma `Code` has no parameters. You use it to terminate the effect of the previously specified `Code` pragma so that subsequent code is placed in the default section. Pragma `Code` applies only at the outer level of the program, so the pairs cannot be nested. The `Code` pragma that is active over the body of a function applies to that function's code. Any `Code` pragma active at a prior declaration of the function is irrelevant.

NOTE See toggle `Each_function_in_own_section` (in Alphabetical List of Compiler Toggles on page 86) for information about a toggle that directs the compiler (and the linker) to map each function into a section with the same name as the function.

Example

This example illustrates the use of pragma `Code` pairs to control the placement of function code:

```
/* ------ Assume default section is ".text" ------ */
void A(){ ... }
...
#pragma Code("SEC1")
...
void B(){ ... }
#pragma Code() /* Terminates pragma Code("SEC1")*/
...
void C(){ ... }
...
```

Assuming the default section for code is named `.text`, function code from this example is placed as follows:

- The code for function **A** is put in section `.text`.
- The code for function **B** is put in section `SEC1`.
- The code for function **C** is put in section `.text`.

Specifying the Name of the Default BSS Section

Pragma BSS specifies the name of the BSS section for static variables that are not initialized at compile time. Such sections are zero-filled at run time. Use this pragma as follows:

```
#pragma BSS("BSS_name")
```

BSS_name is a constant string expression that denotes the name of the BSS section. The default BSS section is `.bss`.

Pragma `BSS` can be specified multiple times. Each specification applies to subsequently declared static variables and is active until another occurrence of the pragma is encountered.

To revert back to the default BSS section, specify pragma `BSS` without parameters:

```
#pragma BSS()
```

NOTE Global uninitialized variables might be mapped into the default BSS section due to linking rules.

Compare pragma Data, which lets you specify your own BSS section.

Example

This example illustrates the use of pragma `BSS` to rename the default BSS section used by the compiler:

```
#pragma BSS("my_BSS")
static int x[100];
...
#pragma BSS("my_other_BSS")
static int y[100];
...
#pragma BSS()    /* Revert to default BSS section */
static int z[100];
```

In this example, the static variables are mapped as follows:

- `x` maps into alternate BSS section `my_BSS`.
- `y` maps into alternate BSS section `my_other_BSS`.
- `z` maps into BSS section `.bss`, the default.

For information about which variables the compiler maps to the BSS section by default, see How the Compiler Maps Variables to Storage on page 252.

Specifying an Alternate Data Section

Associated with every compilation unit is a default data section into which the compiler maps initialized static variables (unless overridden with a pragma Data specification). You can alter the name of the default data section with pragma `Static_segment`, as follows:

```
#pragma Static_segment("Section_name")
```

Section_name is a string constant that names the section.

Ordinarily, the compiler places static variables declared with the **const** type qualifier in the default literals section.

NOTE See Specifying an Alternate Static Data Section on page 157 for information about pragma `Data`.

Specifying an Alternate Literals Section

The literals section is where the compiler ordinarily places read-only static variables and read-only string constants. You can assign an alternate literals section name with pragma `Literals`, as follows:

`#pragma Literals("`*Section_name*`")`

Section_name is a string constant that names the section.

A variable is designated as read-only if it is declared with the **const** type qualifier. A string constant is considered read-only if it is cast to a `(const char *)` type (implicitly or explicitly), or if toggle `Read_only_strings` is on. A string constant is implicitly cast by assigning it to a pointer of type `(const char *)` or passing it as a parameter that is so typed.

See Alphabetical List of Compiler Toggles on page 86 for information about toggle `Read_only_strings`.

Accessing a Common Block from Assembly

The MetaWare C/C++ compiler provides the ability to map more than one variable into a named block; for example, a common block as in FORTRAN. This facility is provided by the `Data` pragma and is documented in Specifying an Alternate Static Data Section on page 157.

Example 14 Accessing a Common Block from Assembly illustrates how such a common block can be accessed from assembly language.

Example 14 Accessing a Common Block from Assembly

C Common-Block Definition

```
#pragma Data(COMMON, "BLOCK")
static  int   a;  /* Offset  0 in "BLOCK" */
static  int   b;  /* Offset  4 in "BLOCK" */
static  char  c;  /* Offset  8 in "BLOCK" */
static  char  d;  /* Offset  9 in "BLOCK" */
static  short e;  /* Offset 10 in "BLOCK" */
#pragma Data()
```

Assembly

```
        .comm   BLOCK,12
        .set    a,0
        .set    b,4
        .set    c,8
        .set    d,9
        .set    e,10
        ...
        mov     %r0,BLOCK
        ld      %r1,[%r0,a]     ; Load value of a
        ldb     %r2,[%r0,c]     ; Load value of c
        add     %r1,%r1,%r2     ; Add c to a
        st      %r1,[%r0,b]     ; Store result in b
```

NOTE Variables a, b, c, d, and e are not global. That is, they do not appear in the symbol table with the **global** attribute. The only name that appears in the symbol table is BLOCK.

Placing Comments in Object Code

Use pragma `Ident` to place notes in a comment section within a generated object module, as follows:

`#pragma Ident("`*comment string*`")`

You can use this pragma multiple times. The name of the comment section is typically `.comment`. When the compiler generates an assembly file, it typically translates the argument of pragma `Ident` to an **.ident** statement.

The compiler normally emits a comment string that identifies the version of the compiler and the selected command-line options used to produce the object file, even when you do not use the `Ident` pragma.

This string can be suppressed by turning off the `Command_line_ident` toggle.

Examples

These are some example uses of the `Ident` pragma:

```
#pragma Ident("Source version 3.4.1\n")
#pragma Ident("Date is " __DATE__)
```

Finding Field Offsets

You can use pragma `Asm_field_offsets` or `Define_field_offsets` to determine the offsets of **struct** fields (members) from the base of their **struct**s. Both pragmas write output to the file you specify in the format you specify. The only difference is that `Asm_field_offsets` uses **.set** directives for the assembler, while `Define_field_offsets` uses **#define** directives for the preprocessor. See Example 15 Using a Pragma to Find Field Offsets.

Using pragma `Asm_field_offsets` or `Define_field_offsets` is useful for writing assembly that accesses structs also accessed by C or C++ code. For example, if you create a **struct** representing a register save layout, you could use preprocessor defines in conjunction with pragma `Asm_field_offsets` to write assembly code that saves exactly the right registers at exactly the right place to keep the assembly and C code synchronized when saving context.

Example 15 Using a Pragma to Find Field Offsets

```
typedef struct {
   int   first;
   char  second;
   short third;
   int   fourth;
} myStruct;

struct {
   short first;
   short second;
   char  third;
} myOtherstruct;

pragma Asm_field_offsets("myOffsets.s", "%t_%m_OFF", myStruct, myOtherstruct);

...

cat myOffsets.s

.set   myStruct_first_OFF, 0
.set   myStruct_second_OFF, 4
.set   myStruct_third_OFF, 6
.set   myStruct_fourth_OFF, 8
```

```
.set   myOtherstruct_first_OFF, 0
.set   myOtherstruct_second_OFF, 2
.set   myOtherstruct_third_OFF, 4
```

Defining Registers

This section describes how to use pragmas to define extension core registers and auxiliary registers to the compiler.

Defining Extension Core Registers

Use pragma `Core_register` to define an extension core register to the compiler. You can access core registers thus defined using compiler-intrinsic functions **_core_read()** and **_core_write()**.

> **CAUTION** If read or write operations on your core register have side effects and you access the core register from C, be sure to use pragma `Core_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

By default, when you use compiler-intrinsic function **_core_read()** or **_core_write()** to access a register, the compiler emits the required directives to define the extension core register to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle `Emit_core_directives`.

For more information, see the following:

- Pragma `Core_register` in Alphabetical List of Pragmas on page 129
- Compiler-intrinsic functions **_core_read()** and **_core_write()** in Appendix A — Intrinsic Functions

Defining Auxiliary Registers

Use pragma `Aux_register` to define an auxiliary register to the compiler.

> **CAUTION** If read or write operations on your auxiliary register have side effects and you access the auxiliary register from C, be sure to use pragma `Aux_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

Example 16 Defining an Auxiliary Register to the Compiler defines an auxiliary register with the following characteristics:

- The name of the auxiliary register is `%myreg`.
- The auxiliary address of the auxiliary register is 0x1234.
- The auxiliary register has a two-cycle latency (the latency is associated with the register, not the instruction).
- The auxiliary register requires a two-cycle block when it is being updated.
- A write to the auxiliary register requires a read from `%r12` and trashes `%r12`.
- A use of the auxiliary register causes a read and a write to auxiliary register `0x88`.

Example 16 Defining an Auxiliary Register to the Compiler

```
#define MY_REG_ADDR 0x1234
pragma aux_register(MY_REG_ADDR, name => “myreg”, latency_cycles => 2,
blocking_cycles => 2, effects =>
“%r12:is_read:is_trashed;auxreg=0x88:is_read:is_written;”);
```

Using and Optimizing Extensions in C/C++

You can use pragma `Intrinsic` to generate extension instructions beyond those known internally to the compiler or to cause the compiler to use a specific known instruction. You can then compile using option **-O***n* to optimize the instructions to the level desired.

- Example 17 Using Extensions to Generate and Use Assembly Instructions illustrates the use of this feature to add instructions and use specific known instructions from C.
- Example 18 Generating an Instruction with Blocking and Latency on page 165 illustrates how to use the *properties* parameter of pragma `Intrinsic` to generate a **MULT** instruction that blocks for two cycles and has a four-cycle latency. For more about the parameters to pragma `Intrinsic`, see the listing on page 140.

Example 17 Using Extensions to Generate and Use Assembly Instructions

MetaWare C/C++ for ARCtangent-A4

```
typedef unsigned long ulong;
extern ulong normalize(ulong);
extern ulong bic(ulong, ulong);
extern long mul(long, long);
pragma intrinsic(normalize, name=>"norm",
                 opcode => 0xa, flags => "zn");
pragma intrinsic(bic, flags => "zn" );
// base-case instruction
pragma intrinsic(mul, opcode => 0x1c,
                 assume_volatile => 1);

main(int argc) {
  if (normalize(3))
     return normalize(bic(argc, 0x8000_0000));

   return mul(0x200, 0x201);
   }
```

Assembly for Generated for ARCtangent-A4 at Optimization Level -O

```
    .file            "x.c"
    .option          %reg
    .ident           "hc2.7e -O0 x.c"
    .seg             ".text",text
.L00TEXT:
;-------------| main |-------------------
    .global          main
main:
    .extInstruction norm,0x03,0x0a,\
      SUFFIX_COND|SUFFIX_FLAG,SYNTAX_2OP
    norm.f           0, 3
    bic.ne           %r0, %r0, -2147483648
    jne.d            [%blink]
    norm.ne          %r0, %r0        delay
    mov %r0, 512 ;0x200
    .extInstruction mul,0x1c,0x00,\
      SUFFIX_COND|SUFFIX_FLAG,SYNTAX_3OP
    mul %r0, %r0, 513
```

```
    j   [%blink]

    .type main, @function
    .size main, . - main

    .end
```

MetaWare C/C++ for ARCtangent-A5 and Later

```
typedef unsigned long ulong;
extern ulong normalize(ulong);
extern ulong bic(ulong, ulong);
extern long mul(long, long);
pragma intrinsic(normalize, name=>"norm",
                 opcode => 0x05, sub_opcode => 0x01, flags => "zn");
pragma intrinsic(bic, flags => "zn" ); // base case
                                       // instruction
pragma intrinsic(mul, opcode => 0x05, sub_opcode => 0x08);

main(int argc) {
   if (normalize(3))
      return normalize(bic(argc, 0x8000_0000));

   return mul(0x200, 0x201);
    }
```

Assembly Generated for ARCtangent-A5 and Later at Optimization Level -O

```
    .file           "intrin.c"
    .option         %reg
    .ident           "hc2.7e -Os intrin.c"
    .seg            ".text",text
.L00TEXT:
 ; --------------| main |--------------
    .align          4
    .global         main
main:
    .extInstruction norm,0x05,0x01,SUFFIX_FLAG,SYNTAX_2OP
    norm.f          0, 3
    beq_s          .LN04.5
    bic            %r0, %r0, -2147483648
    norm           %r0, %r0
    j_s            [%blink]
.LN04.5:
    mov            %r0, 512 ;0x200
    .extInstruction mul,0x05,0x08,SUFFIX_COND|SUFFIX_FLAG,SYNTAX_3OP
    mul            %r0, %r0, 513
    j_s            [%blink]

    .type          main, @function
    .size          main, . - main

    .end
```

Example 18 Generating an Instruction with Blocking and Latency

C/C++

```
extern int mult(int opd1, int opd2);
pragma intrinsic (mult, opcode => 0x6, sub_opcode => 0x1a, blocking_cycles => 2,
latency_cycles => 4);
```

Using Extension Condition Codes in C/C++

You can use pragma `Condition` and pragma `Intrinsic` together to generate extension instructions that set extension condition flags, and then test the conditions in C using compiler-intrinsic function **_condition()**.

For more information, see the following:

- Pragmas `Condition` and `Intrinsic` in Alphabetical List of Pragmas on page 129
- Functions **_condition()** and **_core_read()** in Appendix A — Intrinsic Functions

Example 19 Setting and Testing Extension Condition Codes is a C program that generates extension instructions and tests their conditions, with the resulting assembly code.

Example 19 Setting and Testing Extension Condition Codes

MetaWare C/C++

```
extern xyz_f(unsigned, unsigned);
pragma intrinsic(xyz_f, name => "xyz", opcode => 0x1d,
            set_flags => 1,
            flags => "nz");
enum { ML=0x17, NML=0x18, Y=0x19, NY=0x1a };
pragma condition(ML, name => "ml", inverse => NML);
pragma condition(NML, name => "nml", inverse => ML);
pragma condition(Y, name => "y", inverse => NY);
pragma condition(NY, name => "ny", inverse => Y);

int main(int one, int two) {
   xyz_f(one, two);
   if (_condition(ML) && _condition(Y)) {
      _core_read(32);    //also a compiler-intrinsic
      return two+1;
      }
   else if (_condition(NML))
      return 4;
   else
      return _core_read(32);
        }
```

Assembly for ARCtangent-A4

```
; Compiled with mcc -a4 -core7 -Os
    .extCoreRegister r32,32,r|w,can_shortcut
        .extCondCode    ml, 0x17
        .extCondCode    nml, 0x18
        .extCondCode    y, 0x19
        .extCondCode    ny, 0x1a
;-------------| main |-------------------
        .global main
main:
        .extInstruction xyz,0x1d,0x00,SUFFIX_COND|SUFFIX_FLAG,SYNTAX_3OP
        xyz.f   0, %r0, %r1
        bnml    .LN02.7
        mov.y   0, %r32
        jy.jd   [%blink]
        add     %r0, %r1, 1     ; delay
.LN02.7:
        jnml.d  [%blink]
        mov     %r0, 4  ; delay
```

```
        j.d     [%blink]
        mov     %r0, %r32       ; delay
```

Assembly for ARCtangent-A5 and Later

```
; Compiled with mcc -Os
        .extCoreRegister      %r32,32,r|w,can_shortcut
        .extCondCode          ml, 0x17
        .extCondCode          nml, 0x18
        .extCondCode          y, 0x19
        .extCondCode          ny, 0x1a
        ; --------------| main |--------------
        .global main
main:
        xyz.f   0,%r0,%r1
        bnml    .LN02.7
        bny     .LN02.7
        mov     0,%r32
        j_s.d   [%blink]
        add_s   %r0,%r1,1
.LN02.7:
        jnml.d  [%blink]
        mov_s   %r0,4   ; delay
        mov_s   %r0,%r32
        j_s     [%blink]
```

Chapter 7 — Optimizing Performance

In This Chapter

- Overview
- Changing the Optimization Level
- Applying Manual Optimizations

Overview

Optimizing a program means compiling it to run faster or require less space. As a globally optimizing compiler, the MetaWare compiler offers two types of optimizations:

- Optimizations that you can select by changing the optimization level
- Manual optimizations

You can choose your optimization level, but it is not possible to completely eliminate optimizations or optimization level.

NOTE The compiler always performs certain optimizations that cannot be disabled. For details of the optimizations performed, see the ARC Technical Support site at the URL listed in Technical Support on page 14.

Related Topics

- Default Optimization Level
- Changing the Optimization Level
- Applying Manual Optimizations

Default Optimization Level

If you specify nothing, the default optimization level is **-O**, the recommended speed optimization.

CAUTION Specifying **-g** changes the optimization level.

Debugging Optimized Code

When you specify optimization level 0 (zero) or 1 (one), the compiler tries to generate code in source order. This makes your code easier to debug at the assembly-language level.

In general, optimizations make code harder to debug. Debuggers cannot usually determine when a variable's current value is in a register rather than in memory. These are some of the features that have an impact on debugging:

- Inlining often prevents debuggers from stepping through a program correctly.
- Instruction Scheduling might cause erroneous line numbers to be reported for run-time errors.
- During debugging, Cross-Jumping can result in surprising jumps that are not reflected in the source code.

Changing the Optimization Level

To change the optimization level, use option **-O***n*, where *n* is the number of the level (*0* through *7*) or *s* or *s1*.

Table 6 Changing the Optimization Level

If you specify...	The optimization level is...
Nothing or **-O**	Level 6 (best for speed)
-g	Level 1 (debugging, faster, smaller)
-O*0*	Level 0 (debugging)
-O*1*	Level 1 (debugging, faster, smaller)
-O*2* to **-O***5*	Level 2 to 5 (progressively smaller and faster)
-O*6*	Level 6 (best for speed; same as **-O**)
-O*7*	Level 7 (smaller and faster with loop unrolling)
-O*s*	Level s (space optimizations)
-O*s1*	Level s1 (space optimizations with loop changes)

Optimization Levels

This section provides an overview of optimization levels. For details of the optimizations performed, see the ARC Technical Support site at the URL listed in Technical Support on page 14. In general (but not always), each optimization level includes the previous levels, and adds more optimizations of its own.

Best Levels for Speed and Size

- The best level for speed is usually level 6 (**-O***6* or **-O**)
- The best level for reducing code size is level s (**-O***s1*)

NOTE All levels include the built-in optimizations that cannot be disabled.

Level 0 — Debugging

Optimization level 0 (**-O***0*) aids debugging by generating assembly code in the same order as the source code. It performs no further optimizations beyond the built-in optimizations that cannot be disabled.

Benefits

- Easier debugging

Drawbacks

- None

Level 1 — Debugging, Smaller, Faster

Optimization level 1 (**-O***1*) includes the optimizations performed at level 0, and adds new optimizations to make code smaller and faster.

NOTE All levels include the built-in optimizations that cannot be disabled.

Benefits

- Easier debugging at the assembly level
- Smaller code
- Potentially faster code

Drawbacks

- Speed benefits might vary or even be negative
- Debugging at the instruction level might be harder

Levels 2 through 7 — Progressively Smaller and Faster

Optimization levels 2 through 7 (**-O***2* through **-O***7*) no longer generate assembly code in source-code order. They use different techniques to eliminate some of the drawbacks of level 1 while making code progressively smaller and faster.

NOTE All levels include the built-in optimizations that cannot be disabled.

Choosing a Level

Most applications perform best at level 6. Some applications perform best at level 5. Little difference exists between levels 2, 3, and 4. To profile your code for speed before rebuilding, you can use one of the profiling tools:

- execution Profiling (see the *MetaWare Debugger User's Guide*)
- the GUI-based integrated profiling tool from ARC International (see the *Integrated Profiler User's Guide*)

TIP Sometimes optimizations can be applied to one particular module to good effect.

Notes

- The speed optimization Conditional-Jump Minimization included at these levels requires use of the **Rarely()** and **Usually()** pseudo-functions in your code.
- The speed optimization Inlining Selected Functions included at levels 4 through 7 is most effective on functions type-qualified with **_Inline**.
- To disable the optimization Inline Division included at levels 6 and 7, turn off toggle `Inline_divide`.

CAUTION Exercise caution when using optimization level 7. The ARC processor's zero-delay-loop mechanism generally gives better results than the loop-unrolling optimization included at this level.

NOTE

Benefits

- Faster code
- Smaller code
- Better performance (levels 6 and 7 only)

Drawbacks

- Harder debugging
- Potentially tool-dependent if you use toggles and pseudo-functions the levels make available
- Slower compile

Levels s and s1 — Space

Level s and s1 (**-O***s* and **-O***s1*) do not continue the numbering scheme, but include many time- and space-saving optimizations that also appear at other levels, in addition to new ones, with more focus on space. Level s includes multiple iterations of some optimizations to further reduce code size. Level s1 includes most of the optimizations from level s and also avoids zero-delay loops to further reduce code size.

Notes

NOTE All levels include the built-in optimizations that cannot be disabled.

- The speed optimization Conditional-Jump Minimization included at these levels requires use of the **Rarely()** and **Usually()** pseudo-functions in your code.
- The optimizations included at level s1 prevent zero-delay loops.

Benefits

- Faster code
- Smaller code
- Better performance (levels s only)

Drawbacks

- Harder debugging
- Potentially tool-dependent if you use toggles and pseudo-functions the levels make available
- Slower compile

Applying Manual Optimizations

This section lists optimizations you can apply manually using options, toggles, pragmas, or special functions. Some manual optimizations require a specific optimization level to work. In that case, the optimization level is listed as well.

Addressing Global Variables and Objects As Near

Option

-Hall_sdata

When you specify option **-Hall_sdata**, the compiler aggressively attempts to use near addressing for almost all global variables and objects. This optimization benefits most applications by improving performance and decreasing code size.

However, the maximum size of the small-data area is limited, so many applications cannot use this option. It is often possible to address global variables and objects as near by using pragma `Push_small_data`.

NOTE The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** into small data unless you qualify them as **_Near**.

This optimization has two side effects:

- The common ANSI extension of allowing multiple declarations like **int** `fred;` results in multiply defined variables at link time. In this situation, variables such as `fred` are no longer Fortran common blocks, but instead are `.sbss` labels.
- The compiler assumes that **extern** unsized arrays such as **extern char** `b[];` are small enough to be placed in a small-data area. If `b` is actually quite large, link-time errors occur. Turn off toggle `Unsized_arrays_in_small_data` to reverse this assumption.

For more information, see the following:

- pragma `Push_small_data`
- toggle `Multiple_var_defs`
- toggle `Unsized_arrays_in_small_data`

Aligning Code and Data

Pragmas

`Align_members`
`Pop_align_members`
`Push_align_members`

References to structure members are faster when you use these pragmas to align the members on appropriate byte boundaries. Whether there is much of a space trade-off depends on how many structures your application contains and how they are organized.

Branch Prediction and Branch Profiling

Options

-Hgenbp_data
-Husebp

If you compile using option **-Hgenbp_data**, the compiler instruments the application with code that generates a file of branch-profiling information when you execute the application. When you subsequently compile using option **-Husebp=*bpfile***, the compiler reads in this profiling information from *bpfile*.

The compiler uses the branch profiling information to determine the frequency at which the “true” and “false” targets of a branch are visited, and makes node-ordering decisions accordingly.

For more information, see Appendix E — Branch Prediction.

NOTE Branch profiling can cause additional register usage, which can cause spilling, and might actually increase time in complicated expression-oriented loops.

Compiler-Intrinsic Functions

Compiler-intrinsic functions are functions recognized by the compiler without the presence of a prototype. For a listing of the compiler-intrinsic functions available, see Appendix A — Intrinsic Functions.

Conditional-Jump Minimization

Optimization Levels

2 — 7, s, s1

Pseudo-Functions

`_Rarely(`*Boolean_expression*`)`
`_Usually(`*Boolean_expression*`)`

At optimization level 2 and above, you can use pseudo-functions **_Rarely()** and **_Usually()** in your code to improve execution speed by telling the compiler how frequently a Boolean expression evaluates to true. This technique is useful when you have located a bottleneck in an application's performance.

NOTE These functions are not available at optimization levels below 2.

Pseudo-functions **_Rarely()** and **_Usually()** accept a Boolean expression as an argument and return the value of the expression, but also give the compiler a "hint" about the likelihood of that expression being true. Thus **_Usually(*expr*)** evaluates expression *expr* and informs the compiler that the value is usually true. Likewise, **_Rarely(*expr*)** evaluates expression *expr* and informs the compiler that the value is rarely true.

When a call to **_Rarely()** or **_Usually()** is the controlling expression of an **if** statement, the compiler can ascertain which control path is more likely to execute. Using this information, the compiler can order blocks to reduce the number of conditional jumps taken at run time.

The performance improvement is sometimes significant for **if** statements inside frequently executed loops.

Example

```
if (_Rarely(a || b) ){
  statement1();
      }
else
   statement2();
```

In this example, the compiler places the block containing **statement2()** so that the **if** expression "falls through" directly to it, and places the block containing **statement1()** out of line, as it is rarely executed.

Constants in Code

Toggle

`Read_only_strings`

When toggle `Read_only_strings` is on, all string constants are assumed to be read-only. This assumption permits the compiler to place the string constants in read-only sections, thus reducing system overhead when more than one process is executing the same code.

NOTE String constants qualified with type **const** are unconditionally placed in the read-only data section.

CAUTION Placing constants in code can reduce page writes on systems that implement virtual memory management.

Designating Certain Functions As "Called Infrequently"

Pragma
`Called_infrequently`

When pragma `Called_infrequently(`*string*`)` is specified, the compiler designates as "called infrequently" any function whose name contains *string*. Whenever a function designated as "called infrequently" is conditionally called within a loop construct, the compiler assumes that the basic block (node) containing the call is seldom executed. Based on that assumption, the compiler does the following:

- It reorders the nodes so the node containing the function call is moved physically out of the loop.
- It sets a jump from the node containing the function call back into the loop.

NOTE The configuration file — as distributed — automatically applies calling-convention attribute **_CALLED_INFREQUENTLY** to functions with names containing any of these strings: `abort`, `error`, `fail`, `fatal`, or `trace`.

Benefits of Using This Optimization

Using pragma `Called_infrequently` in your code usually benefits error checking within a loop construct. When the portion of the **if** that contains the infrequently called function is moved out of the loop, the remaining code usually runs better, because there are fewer jumps involved. Such jumps would normally bypass the portion of the **if** that contains the infrequently called function.

Using this pragma benefits database code and usually benefits code (like a benchmark) that has hot spots. *Hot spots* are portions of code, usually loops, that run most of the time the code is running.

Eliminating Unreferenced Functions and Variables

Toggles
`Each_function_in_own_section`
`Each_var_in_own_section`

Linker Option
-zpurge

Turn on compiler toggles `Each_function_in_own_section` and `Each_var_in_own_section` to map each function and variable into a separate section with the function or variable name. Then use linker option **-zpurge** to remove unreferenced sections.

See the *ELF Linker and Utilities User's Guide* for details on option **-zpurge**.

Expression Simplification: Converting Division to Multiplication

Toggle
`Use_reciprocal_for_divide`

A floating-point divide operation whose divisor is a constant can be converted to a multiplication by the reciprocal of the constant. In most cases, floating-point multiplication is significantly faster than division. However, if the reciprocal of the constant is not exactly representable, multiplication might introduce an inaccuracy in the last bit of the resulting mantissa. In most programs this does not matter; you must decide whether this level of accuracy is critical in your application.

Toggle `Use_reciprocal_for_divide` must be turned on to force multiplication by reciprocal; this toggle works at all optimization levels.

Law of Associativity

Applying the law of associativity can also reduce the number or complexity of instructions required. The compiler does not apply this optimization automatically, but you can increase the efficiency of your code by careful attention to how you order operations.

In this example, `(x + k1) + k2` becomes `x + (k1 + k2)`:

```
float x, y;
int k1, k2;
...
y = (x + k1) + k2;
```

Computing `(x + k1) + k2` requires the following steps:

1. Convert `k1` to a **float**.
2. Add `x` and `k1`.
3. Convert `k2` to a **float**.
4. Add intermediate result and `k2`.

Computing `x + (k1 + k2)` requires the following steps:

1. Add `k1` and `k2`.
2. Convert the result to a **float**.
3. Add this to `x`.

High-Level/Intermediate Language Merging

Option

-Hmerge

Toggle

`Treat_exports_as_locals`

High-level/intermediate language merging is an interprocedural optimization that gives the optimizer and code generator direct access to all of the source files being compiled. It is similar to wide inlining (**-Hiw**), but is more powerful as it concatenates the intermediate language for all of the source files and passes it to the optimizer as one file containing all code and data (the other generated object files are left empty).

> **NOTE** To benefit from this option, you must compile all source files at once, for example:
> `mcc *.c -Hmerge -Hon=treat_exports_as_locals`

Inline Division

Optimization Levels

6 — 7

Toggle

`Inline_divide`

At the appropriate optimization levels, the compiler generates code to perform integer division inline for better performance at the expense of code size.

Inlining C++ Inline Functions

Options
-Hinlsize
-Hinlfactor

C++ only
The inlining of C++ **inline** functions is performed automatically. The compiler limits the size and expansion factor of functions that are automatically inlined, to limit code-size expansion in compiled code. To change the maximum size of functions to inline, specify option **-Hinlsize**; to specify the maximum expansion factor of inlined functions, use option **-Hinlfactor**. For more information, see The Built-in C++ Inliner on page 188.

Inlining Certain ANSI Standard C Functions

Toggle
`Recognize_library`

When toggle `Recognize_library` is on, the MetaWare C/C++ compiler automatically replaces calls to certain ANSI Standard C library functions with fast inline code.

NOTE Do not confuse ANSI Standard C functions that are inlined when toggle `Recognize_library` is on with compiler intrinsic functions. See Appendix A — Intrinsic Functions for more information.

The MetaWare C/C++ compiler might choose to inline the following ANSI Standard C functions when toggle `Recognize_library` is on:

```
abs()
labs()
```

The compiler might also choose to inline the following math functions when the input arguments are compile-time constants and `Recognize_library` is on:

acos()	ceilf()	log()	sinhf()
acosf()	cos()	log10()	sqrt()
asin()	cosf()	log10f()	sqrtf()
asinf()	cosh()	logf()	tan()
atan()	coshf()	pow()	tanf()
atan2()	exp()	powf()	tanh()
atan2f()	expf()	sin()	tanhf()
atanf()	floor()	sinf()	
ceil()	floorf()	sinh()	

To disable automatic inlining of ANSI Standard C library functions, turn off toggle `Recognize_library`.

NOTE If you do not have a prototype for the function, but the argument types match the expected argument types (and the return type matches the expected return type), the compiler might inline the unprototyped function.

Inlining Selected Functions

Optimization Levels

4 — 7, s

Type Qualifier

_Inline

At the appropriate optimization levels, the compiler inlines functions type-qualified with **_Inline**, and also automatically inlines specific functions depending on their size and how many times they are called

This optimization is not performed if you specify option **-g**.

You can also use option **-Hi[modifier]** to invoke the outboard inliner for functions (routines) meeting specific criteria. See Outboard Inliner Option Reference on page 186 for more information.

CAUTION Inlining might make function definitions longer, resulting in increased compile time. Too much inlining can increase the size of a function beyond the ability to fit in a cache.

Tail Recursion

Toggle

`Tail_recursion`

If a function calls itself recursively, and the call is the last action the function performs, the code can be modified to branch to the beginning of the function instead of calling it again, with appropriate adjustments for parameter passing and return. In performing tail recursion, the compiler optimizes any recursively called function that can be modified this way.

Chapter 8 — Inlining

In This Chapter

- Overview
- The Outboard Inliner
- Outboard Inliner Option Reference
- The Built-in C++ Inliner
- Inlining across Modules (Wide Inlining)

Overview

This chapter describes the function inliners available with the MetaWare C/C++ compiler and how to use them.

NOTE If you want to write your own assembly code and direct the compiler to insert that code at its output, use **_ASM()** or **_Asm** as documented in Chapter 9 — Inline Assembly Code.

The MetaWare C/C++ compiler includes two inliners for converting C and C++ code to inline assembly code: an outboard inliner and a built-in C++ inliner.

- The outboard inliner is automatically invoked as a separate compiler optimization phase at optimization levels 4 to 7 and s. This outboard inliner can also be invoked manually through the use of compiler options and pragmas.
- The built-in C++ inliner — part of the MetaWare C++ compiler — automatically inlines C++ functions type-qualified as **inline**.

NOTE Generating debugging information (option **-g**) suppresses inlining.

What Is Inlining?

When a function or iterator (referred to collectively as *routines*) is inlined, calls to it are replaced with the logic contained in the routine. (See the *MetaWare C/C++ Language Reference* for information on iterators, a MetaWare C/C++ language extension.)

Both inliners back-substitute routine bodies where the routines are called, and replace parameters with local variables or constants where appropriate. The substitution saves at least the overhead of prologs, epilogs, and parameter passing, and exposes opportunities for additional optimizations — such as copy propagation and constant folding — especially when the arguments themselves are constants.

The primary benefit of the inlining optimization is that it speeds up execution. However, code size is potentially larger, unless the inlined routine contains no more code than would be generated for the replaced calling sequence.

Factors such as exportability, addressability, and recursiveness play roles in determining whether an inlined routine must also be defined. If the routine can be completely inlined, its definition is not emitted. See Rules for Automatic Inlining in Automatically Invoking the Inliner on page 183.

CAUTION Inlining might make routine definitions longer, resulting in increased compile time. Too much inlining can increase the size of a routine beyond the ability to fit in a cache.

Several Ways to Inline Code

There are several ways to direct the MetaWare C/C++ compiler to inline your code: automatically, semi-automatically, or manually.

- Automatically, as an optimization

  When you specify optimization level 4 to 7, or s, the compiler automatically invokes the outboard inliner to inline routines that meet criteria defined internally for those optimization levels. This optimization is described briefly in Inlining Selected Functions on page 179. For more information, see Automatically Invoking the Inliner on page 183.

- Automatically, as part of C++ compilation

  When you type-qualify C++ functions with the **inline** keyword, the compiler invokes the built-in C++ inliner to inline those functions. Additionally, you can specify the maximum size and expansion factor of inlined C++ functions with options **-Hinlsize** and **-Hinlfactor**, respectively. Use of this inliner is described in The Built-in C++ Inliner on page 188. For information about the **inline** keyword, see the *MetaWare C/C++ Language Reference*.

- Semi-automatically, for routines meeting specific criteria

  When you specify one of the **-Hi[modifier]** options, the compiler invokes the outboard inliner to inline those routines that meet the criteria defined by the particular option specified. Use of these options is described in Outboard Inliner Option Reference on page 186. For more information, see Semi-Automatically Invoking the Inliner on page 184.

- Manually, by explicitly invoking the inliner

  When you type-qualify C routines with **_Inline** and specify option **-Hi**, the compiler invokes the outboard inliner to inline those routines. Option **-Hi** is summarized in General Driver Options on page 52. For information about type-qualifier **_Inline**, see the *MetaWare C/C++ Language Reference*.

- Manually, using pragmas

  When you specify one of the inlining pragmas (`On_inline`, `Off_inline`, or `Pop_inline`) for a particular routine, the compiler does (or does not, as directed) inline that routine. For information about this technique, see Using Pragmas to Inline Individual Routines on page 185.

- Manually, using a calling-convention attribute

  MetaWare C/C++ provides a **_FORCE_INLINE** calling-convention attribute. See the listing in Calling-Convention Attributes on page 315 for details.

The Outboard Inliner

The MetaWare compiler provides an optional optimization phase referred to alternately as the *function inliner* or the *outboard inliner*. In addition to automatically invoking the inliner as part of certain compilations, you can manually specify individual routines to be inlined, or allow the compiler to select routines to be inlined according to evaluation criteria you provide.

NOTE Generating debugging information (option **-g**) suppresses inlining.

NOTE The outboard inliner described in this section is not necessary for compiling C++ functions, even those with inline function declarations. Those functions are automatically inlined by the C++ compiler's built-in inliner, which offers almost the same functionality as the outboard inliner.

Use this outboard inliner for C++ programs only if you want more inlining than is automatically provided by the C++ compiler. See The Built-in C++ Inliner on page 188.

Automatically Invoking the Inliner

As a compiler optimization phase, the outboard inliner is automatically turned on by the compiler driver at optimization levels 4 to 7 and s (that is, when you specify options **-O4** to **-O7** or **-Os**).

You can direct the compiler driver to inline certain routines at lower optimization levels by specifying option **-Hi** and using the **_Inline** type qualifier; for details, see Using Type Qualifiers to Inline Individual Routines on page 185.

Rules for Automatic Inlining

The compiler uses the following rules to determine if a routine can be automatically inlined (and whether the inlined routine must also be defined):

- Routines that are exported, recursive, or have their address taken are inlined but also defined.
- Routines that are not exported, are not recursive, and do not have their address taken, are inlined but the definition is discarded.
- Recursive routines are inlined only once and then a call is generated.
- Routines that contain nested routines can be inlined only if all nested routines are inlined.
- Routines that contain calls to the intrinsic function **_Alloca()** or to an external routine named **alloca()** are not inlined.
- Routines that use variable arguments are not inlined.

Semi-Automatically Invoking the Inliner

In addition to automatically inlining routines at optimization levels 4 and above, you can also direct the compiler to automatically inline routines that satisfy adjustable criteria you specify, such as the number of calls and the size of the routine.

When you specify one or more of the **-Hi[modifier]** options to invoke the outboard inliner (where the modifiers give selection criteria), the compiler inlines those routines that meet the criteria — even if the compiler optimization level is lower than required for automatic inlining and even if those routines are not type-qualified with **_Inline**.

The following **-Hi[modifier]** options specify criteria that the compiler can use to select routines for inlining.

-Hia	Address — Do not inline routines whose address is taken.
-Hib=*n*	Buffer — Specify temporary-file buffer size.
-Hic=*c*	Called — Inline routines called *c* or fewer times.
-HiC=*t*,*c*	Called — Inline routines with *t* or fewer tree nodes, called *c* or fewer times.
-Hih	Help — Show inlined and/or defined routines.
-Hir	Recursive — Do not inline recursive routines.
-His=*n*	Stack — Inline routines with a stack size less than *n* bytes.
-Hit=*n*	Tree — Inline routines with fewer than *n* tree nodes.
-Hiw	Wide — Inline in multi-module (wide inlining) mode.
-Hix	Exported — Do not inline exported routines.

These options are described in greater detail in Outboard Inliner Option Reference on page 186.

NOTE Type qualifier **_Inline** applied to a routine overrides any **-Hi[modifier]** options that might otherwise exclude that routine from being inlined.

Manually Invoking the Inliner for Specific Routines

You can specify individual routines to be inlined — or not inlined — by using type qualifier **_Inline** or pragmas `On_inline` and `Off_inline`.

Using Type Qualifiers to Inline Individual Routines

If you type-qualify individual routines in your source code with **_Inline** and then turn on option **-Hi** (with no modifier) to invoke the outboard inliner, the compiler inlines those routines.

Option **-Hi** is on by default at some optimization levels; refer to the listing in Chapter 4 — Using Driver Options for details.

For example, this line causes the compiler to inline **func()** when option **-Hi** is specified:

```
_Inline void func() { ... }
```

NOTE In order for option **-Hi** to cause the compiler to inline a routine, you must type-qualify the routine with **_Inline**.

Certain conditions can make a routine not inlinable; see Rules for Automatic Inlining in Automatically Invoking the Inliner on page 183.

For more information about type qualifier **_Inline**, a MetaWare C/C++ language extension, see the *MetaWare C/C++ Language Reference*.

Using Pragmas to Inline Individual Routines

You can control the inlining of individual functions with inlining pragmas `On_inline`, `Off_inline`, and `Pop_inline`). When activated, these pragmas are pushed onto a stack for each function specified. Use pragma `Pop_inline` to pop the top-most stack entry for the specified function and restore the previous state for that function.

Pragma Off_inline

Specifying pragma `Off_inline(`*routine_name*`)` prevents the compiler from inlining calls to *routine_name* when it otherwise would inline them.

This prevention occurs even if the routine is declared as an inline function with **_Inline** or **inline** or (in the case of C++ class member functions) if the function is inlined in the class definition.

The `Off_inline` state is pushed onto a stack for each function specified.

Pragma Pop_inline

Pragma `Pop_inline` pops the top-most stack entry for each function specified, and restores the previous pragma `On_inline` or `Off_inline` state.

Pragma On_inline

Specifying pragma `On_inline(`*routine_name*`)` directs the compiler to inline calls to *routine_name* even if the routine does not otherwise meet the criteria for inlining. For example, calls to *routine_name* are inlined even if *routine_name* is not declared as an inline routine with **_Inline** (C) or **inline** (C++), or if it does not fit the specified (or default) size or call-count requirements for inlining of C routines.

The `On_inline` state is pushed onto a stack for each routine specified in the argument list.

NOTE Even if a routine appears in the pragma `On_inline` argument list, it cannot be inlined if it contains calls to **Alloca()** or **alloca()**, or to **setjmp()**, or if it is a varargs routine.

Pragma `On_inline` does override inliner options that suppress inlining for recursive and exported routines, and for routines that have their address taken (options **-Hir**, **-Hix**, and **-Hia**, respectively).

C++ Only

Pragma `On_inline` has no effect on C++ functions that require a modifier.

Turning off the Outboard Inliner

Option **-Hnoinline** directs the compiler to skip the inlining optimization pass. Specifying **-Hnoinline** guarantees that code will not be inlined, even if the optimization level is 4 or higher. See General Driver Options on page 52 for information about **-Hnoinline**.

Outboard Inliner Option Reference

This section provides a complete reference of outboard-inliner options. For a quick-access listing, see Quick Inliner Options List on page 361.

NOTE Many of the options listed in this section are implicitly specified at optimization level 4, but can be explicitly specified at any optimization level to alter the compiler's default behavior.

NOTE Generating debugging information (option **-g**) suppresses inlining.

-Hi — Invoke the inliner during compilation to inline routines type-qualified with _Inline

Use this option when you want to inline only those routines you designate individually with type qualifier **_Inline**, and when you are not specifying any other **-Hi** option on the command line.

-Hia — Do not inline routines that have their address taken

Option **-Hia** directs the inliner to not inline routines that have their address taken. Such routines might be inlinable, but still must be defined so that they will have an address.

-Hib=*n* — Specify temporary-file buffer size

Option **-Hib** specifies a buffer size *n* for the compiler temporary files; the default size is set at 65,536 bytes. You can override this default setting by specifying a different value for *n*; for example, to save buffer space, you can specify a smaller value. The default buffer size is large because the inliner does extensive seeking and reading through the temporary files.

-Hic=*n* — Inline routines called *n* times

When you specify **-Hic** and *n* is not 0 (zero), routines that are directly called *n* times are automatically inlined. If you specify **-Hic** and **-Hit** (with non-zero values), they work together — that is, both conditions must be met before a routine is automatically inlined.

The compiler default value for *n* is **4**. The default value is used only when the optimization level is 4 or higher, and you do not specify a value for n on the command line. At optimization level s, the default is 1 (one).

NOTE The -Hit and **-Hic** options work together. If both options are specified and their specified arguments are not 0 (zero), both conditions must be met for inlining to occur. If only one of these options is specified, and the specified condition is met, the routine is inlined.

CAUTION Do not confuse option **-Hic** with option -HiC, which specifies inlining routines with fewer than a certain number of tree nodes and called fewer than a certain number of times.

-HiC=*t*,*c* — Inline routines with *t* tree nodes, called *c* times

Option **-HiC** specifies that a routine with *t* tree nodes should be inlined only if it is called *c* times.

You can use **-HiC** more than once to specify routines of different sizes and number of calls. For example:

```
hcarc -HiC=10,5 -HiC=20,4 -HiC=25,2 myprog.c
```

This example tells the compiler to inline the following:

- any routine with fewer than 10 nodes if it is called 5 times or fewer
- any routine with fewer than 20 nodes if it is called 4 times or fewer
- any routine with fewer than 25 nodes if it is called 2 times or fewer

To determine the compiler's default inlining criteria at a specific optimization level, invoke the compiler with the relevant **-O*level*** option and option -V to display all toggles and options being passed to the compiler, and search the output for the arguments to *t* and *c*.

CAUTION Do not confuse option **-HiC** with option -Hic, which specifies inlining only infrequently called routines.

-Hih — Show which routines have been inlined and/or defined

Option **-Hih** displays the name and status of each routine as the inliner encounters it. The status information tells whether the routine has been inlined, inlined and defined, or not inlined. If a routine is not inlined, the status information includes the reason.

Option **-Hih** also prints the following information for each routine:

- the size of the routine, in tree nodes (see note)
- an approximation of the stack-frame size required by the routine
- the number of calls made to the routine

NOTE There is approximately one tree node for each operator, operand, or other language construct such as a statement, loop, declaration, or function.

-Hir — Do not inline recursive routines

By default, if a directly or indirectly recursive routine meets the criteria for inlining, it is inlined only at the first level. A definition is then generated so the inlined routine can be called at subsequent levels. To suppress inlining of recursive routines, even at the first invocation level, use option **-Hir**.

-His=*n* — Inline routines with stack size less than *n* bytes

When you specify **-His** and *n* is not 0 (zero), only routines with a stack size of less than *n* bytes are automatically inlined.

-Hit=*n* — Inline routines with fewer than *n* tree nodes

When you specify **-Hit** and *n* is not 0 (zero), routines with fewer than *n* expression-tree nodes are automatically inlined. There is one tree node, roughly, for each operator, operand, or other language construct, such as a statement, loop, declaration, or routine.

The default value for *n* is system-dependent; see the configuration file. The default value is used only when the optimization level is 4 or higher and you do not specify a value for *n* on the command line.

> **NOTE** The **-Hit** and **-HiC** options work together. If both options are specified and their specified arguments are not 0 (zero), both conditions must be met for inlining to occur. If only one of these options is specified, and the specified condition is met, the routine is inlined.

-Hiw — Invoke inliner in multi-module (wide inlining) mode

When you specify option **-Hiw**, the inliner operates on all files being compiled as a collection rather than operating on each file individually. This is known as *multi-module inlining* or *wide inlining*.

Multi-module inlining permits the inlining of a routine from one module into another module. This can improve performance when you use separate files for data hiding, or when more than one programmer is working on a project.

Benchmark performance can also be improved in cases where a benchmark is split across multiple files. For example, you can realize a significant performance improvement in the Dhrystone 2.1 benchmark with the following options:

> **NOTE** **-Hiw** and **-Hmake** are mutually exclusive.

```
hcarc dhry1.c dhry2.c -Hiw -Hit=250
```

-Hix — Do not inline exported routines

Option **-Hix** directs the inliner not to inline exported routines. Exported routines might be inlinable, but they still must be defined in order to be exported.

The Built-in C++ Inliner

The MetaWare C++ compiler includes a built-in C++ inliner that provides most of the functionality of the outboard inliner. The inlining of C++ **inline** functions is performed automatically, so you do not need to invoke the outboard inliner to inline such functions.

> **NOTE** Generating debugging information (option **-g**) suppresses inlining.

The compiler limits the size and expansion factor of functions that are automatically inlined, to limit code-size expansion in compiled code. You can use option **-Hinlsize** to change the maximum size of functions to be inlined and option **-Hinlfactor** to change the expansion factor of functions to be inlined.

Inlining C++ Functions with the Outboard Inliner

You can use the outboard inliner to inline a function anywhere, even if the function is called prior to its own definition (normally, such forward referencing is an error in C++ programs).

This inlining is especially helpful with functions in template classes, because the expansion of separately defined class template functions must be delayed until the end of compilation, in case you have overridden a template function with a specific version of your own. Thus, calls to the template function are emitted by the compiler before the function definition is seen.

In most cases, you will not need to use the outboard inliner for C++ functions. Use the outboard inliner for C++ functions only if you want to inline the following:

- arbitrary functions that are not necessarily designated as **inline**
- functions whose bodies are presented after the compiler sees calls to those functions
- functions across compilation units (through wide inlining); see Inlining across Modules (Wide Inlining) on page 189 for more information

For information about using **-Hi[modifier]** options to inline functions based on compiler-evaluated criteria rather than on their type qualification, see Semi-Automatically Invoking the Inliner on page 184.

Inlining and throw Statements

The compiler does not inline C++ functions containing a **throw** statement. It places copies of all inline functions containing a **throw** statement in any module that calls the inline function.

If you want inlining to occur, you must place the **throw** in a static member function and call this static member function from the inline function.

Inlining across Modules (Wide Inlining)

You can direct the inliner to operate on a group of files being compiled as a collection, rather than on each file individually. This is known as *multi-module inlining* or *wide inlining*.

NOTE Generating debugging information (option **-g**) suppresses inlining.

When you use separate source files for data-hiding purposes or because more than one programmer is working on a project, inlining a routine from one module into another can markedly improve performance. Also, benchmark results can be much improved where a benchmark is split across multiple files.

For multi-module inlining, specify option **-Hiw**.

The outboard inliner constructs unique public names for variables and routines that can easily be traced back to the defining module

Unique Names in Wide-inlined Code

For multi-module inlining, the inliner invents public names to communicate private data or code from one module to another. You might encounter these names when you use a debugger (although debugging inlined code is not recommended).

The names are constructed as follows. Suppose function **f()** uses a data item, `D`, and is located in a single module, `m1`. If **f()** is inlined into a separate module, `m2`, data item `D` must be available to `m2`. If `D` is private to `m1`, the inliner must make `D` public so that `m2` can access it.

Here is an example of the inliner constructing unique names:

File `m1.c`:

```
static int i = 7;
sub(int j) { i += j; }
```

File `m2.c`:

```
main () {
   sub(1);
   }
```

In this example, `i` must be available to the body of **main()** in `m2` when **sub()** is inlined. The inliner constructs a unique name for `i` and makes it public; that name is then used within both `m1` and `m2`.

The inliner constructs a unique name that can be easily traced back to the defining module.

The constructed name consists of three components, in this order:

1. a double underscore "`__`" followed by the name of the original variable or function
2. a single underscore followed by the name of the module defining the variable or function
3. a single underscore followed by a checksum value

For example, the unique constructed name for the private variable **i** in module `m1` of the preceding example would be `__i_m1_Vbd9755ee3`.

In the case of private compiler-generated data, the string `inl` is used in place of the variable name.

Chapter 9 — Inline Assembly Code

In This Chapter

- Overview
- Using GCC-Style Inline Assembly
- Inserting an Inline Assembly Function Using the AT&T Facility
- Inserting a Single Line of Assembly with _ASM()

Overview

This chapter describes mechanisms for inserting your own inline assembly code into the compiler's code stream. Inline-assembly insertions are sometimes called *macros* even if they are not encapsulated in a macro.

For information about interrupt functions in embedded applications, see Writing Interrupt Handlers on page 224.

For information about using the MetaWare C/C++ inliner to generate inline code for certain C and C++ functions and iterators during compilation, see Chapter 8 — Inlining.

NOTE Hand-written assembly must conform to the run-time model described in Chapter 13 — Run-Time Organization.

We recommend that, whenever possible, you use intrinsic functions supplied by the compiler or defined with pragma `Intrinsic` instead of inline assembly. For more information, see Appendix A — Intrinsic Functions and pragma `Intrinsic`.

Addressing Registers with and without '%'

Registers within all forms of inline-assembly insertions (or *macros*) must be specified with a '%' prefix by default. To eliminate this requirement, turn off toggle `Asm_percent_reg`.

Using GCC-Style Inline Assembly

The MetaWare ARC compiler supports the GCC extended Asm feature. This inline assembly feature makes it easier to describe the specific registers affected so the optimizer does not have to assume the worst case.

Related Topics

- Syntax on page 192
- GCC Extended Asm: Usage Tips and Techniques on page 194
- Real-World Examples on page 199

Syntax

GCC-style inline assembly lets you specify the operands of an assembly instruction using C expressions so that you need not guess which registers or memory locations contain the data you want to use. This section explains the basic syntax; see also Example 20 Basic Example of How to Use GCC-Style Inline Assembly on page 194 and GCC Extended Asm: Usage Tips and Techniques on page 194.

Overview

The syntax consists of an assembly string (*instruction template* in GCC terminology), plus an optional operand-constraint string for each operand, followed by an optional list of registers killed by the assembly string.

Syntax Listing

`__asm__ ("`*assembly_string*`"[...]:[`*Outputs*`][:`*Inputs*`][: "`*kill_reg*`"[, ...]]);`

Use of underscores is optional; if you are writing a header file that might be included in an ANSI or ISO C program, write **__asm__** instead of **asm**.

A colon separates the assembly string from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The list of killed registers begins with a colon.

If your assembly has input operands but no output operands, you must place two consecutive colons around the place where the output operands (*Outputs*) normally go. Output-operand expressions must be lvalues. Input operands need not be lvalues.

Components of the Syntax Listing

- *assembly_string* is the assembly code to inline, in double quotes. You must include at least one string of assembly code to inline. The compiler does not check the assembly code for validity.
- *Outputs* and *Inputs* are one or more comma-separated strings of the form "*constraint_string*"(*operand*) where
  - — *operand* is the C expression for the operand to which to apply the preceding constraint. Enter `memory` if the operand is in memory. If the operand is a register, enclose the register number in double quotes.
  - — *constraint_string* is made up of one or more of the following constraint modifiers (always in double quotes):

Table 7 Constraint Modifiers for GCC-Style Inline Assembly

Modifier	Definition\
`=`	Output (*result*) operand. Required for all output operands.
+	Operand is both input and output. List the operand with the output operands.
`&`	Early clobber output: Output operand is set before input operands are used.
`0`, `1`, `2`, `...`	Input operand lives in the same register as the specified output operand; see Using Numeric Constraints to Specify Operand Co-Location on page 198.
`c`	Any constant
`g`	Any operand
`i`	Operand is an immediate constant.
I	Any integer operand
m	Operand is in memory.
`n`	Operand is an immediate constant (same a `i`).
`r`	Operand is expected in a register.
V	Any variable (operand, register, or memory)

- *kill_reg* is the register (if any) killed by the specific assembly instruction. Enclose the register number in double quotes. For more than one register, separate the registers by commas: `"%r0"`, `"%r1"`, `"%r2"` For memory, enter `memory`. See Example 25 Using GCC __asm__ with Instructions That Kill Specific Registers on page 197.

Order of Operands

The list of operands is implicitly numbered from left to right, starting with 0. The operands are substituted into the assembly string by the appearance of the operand number in the assembly string prefixed with a % character.

Example 20 Basic Example of How to Use GCC-Style Inline Assembly

This example shows how to use GCC-style inline assembly to perform a bitwise rotate on a variable:

```
asm ("ror %0, %1": "=r"(rot_var):"r"(var));
```

This example assumes `var` and `rot_var` are variables in the C program.

Analysis

This example performs a bitwise rotate on the contents of variable `var` and stores the result in variable `rot_var`. It uses one output operand and one input operand. Register operand `rot_var` is substituted into the assembly string for `%0`, and the next operand, var, is substituted for `%1`.

Symbolic Names for Operands

It is also possible to specify input and output operands using symbolic names that you can reference in the assembly string. You specify symbolic names in brackets preceding the constraint string, and can then reference them from the assembly string using `%[`*name*`]` instead of *%number*.

NOTE Symbolic operand names have no relation whatsoever to other C identifiers. Any name may be used, including names of existing C identifiers, but you may not use the same name to represent two different operands of a single inline-assembly insertion.

Example 21 Basic Example Repeated Using Named Operands

This example shows the same rotate operation as Example 20 but uses named operands instead of operand numbers.

```
asm ("ror %[dest], %[src]":[dest]"=r"(rot_var):[src]"r"(var));
```

Related Topics

- GCC Extended Asm: Usage Tips and Techniques on page 194
- Real-World Examples on page 199

GCC Extended Asm: Usage Tips and Techniques

This section describes techniques for using GCC-style inline assembly. It includes the following topics:

- Placing Your GCC Inline-Assembly Insertions in Macros on page 195
- Using volatile to Preserve Instructions from Optimization on page 195
- Including Multiple Assembly Instructions and Formatting Your Output on page 196
- Ensuring the Correct Data Type on page 196
- Specifying Instructions That Kill Specific Registers on page 197
- Working with Output Operands on page 197
- Using Numeric Constraints to Specify Operand Co-Location on page 198
- Handling Unpredictable Memory Accesses by Listing Memory As Killed on page 198
- Working with Explicit Register Variables on page 198

- Working with Condition Codes on page 199
- Jumps between Insertions Not Permitted on page 199

Placing Your GCC Inline-Assembly Insertions in Macros

Usually the most convenient way to use GCC inline-assembly insertions is to encapsulate them in C macros that look like functions.

Example 22 GCC __asm__ Insertion in a Macro

```
//An example from the Linux kernel
inline void
set_bit(unsigned long nr, volatile void * addr)
{
   unsigned long temp, flags;
   int *m = ((int *) addr) + (nr >> 5);


   __asm__ __volatile__(
   "      ld %0,%3\n\t"
   "      bset %0,%0,%2\n\t"
   "      st %0,%1\n\t"
   :"=&r" (temp), "=m" (*m)
   :"Ir" (nr), "m" (*m));

}
```

See also Including Multiple Assembly Instructions and Formatting Your Output on page 196.

Using volatile to Preserve Instructions from Optimization

You can prevent the contents of an **asm** instruction from being optimized away by writing the keyword **volatile** after **asm** or **__asm__**.

Example 23 Using volatile to Prevent Deletion of a GCC __asm__ Insertion

```
//An example from the Linux kernel

#define __cli()                                        \
      __asm__ __volatile__ (                           \
      "lr       r20, [status32] \n\t"                  \
      "and      r20, r20, %0 \n\t"                     \
      "flag r20      \n\t"                             \
      :                                                \
      :"n" (~(STATUS_E1_MASK | STATUS_E2_MASK))        \
      :"r20");
```

NOTE You can write **volatile** or **__volatile__** because the MetaWare compiler supports the GCC **__attribute__** syntax. For more on the MetaWare implementation of the **__attribute__** syntax, see the *MetaWare C/C++ Language Reference*.

The **volatile** keyword indicates that the instruction has important side-effects. The compiler does not delete volatile GCC inline-assembly insertions if they are reachable, though other optimizations are still applied.

Optimization and Assembly Output

The existence and final ordering of assembly instructions depends on actions taken by the optimizer. Even instructions inserted as **volatile** might be moved forward or back, even to unexpected locations some distance from the intended point of insertion.

The following are methods to obtain strictly consecutive output:

- Use a single inline-assembly insertion containing multiple assembly strings; see Including Multiple Assembly Instructions and Formatting Your Output on page 196.
- Use the output variable of one inline-assembly insertion as input to another.

A GCC inline-assembly insertion without any output operands is treated identically to a volatile GCC inline-assembly insertion.

Preserving Modified Operands

It is not permissible to specify that an input operand is modified without also specifying it as an output operand. Note that if all the output operands you specify are for this purpose (and hence unused), you must also specify **volatile** for the **asm** construct, to prevent the compiler from discarding the inline-assembly insertion as dead code.

Example 34 Using GCC __asm__ in an Atomic Exchange with Memory on page 200 specifies a modified input operand as an output operand as well.

Including Multiple Assembly Instructions and Formatting Your Output

You can put multiple assembly instructions together in a single assembly string, separated as desired by newlines to break the line and tabs to move to the instruction field (`\n\t`). This produces more readable assembly output.

You can place your assembly strings all on one line or consecutive lines, but if you place your strings on consecutive lines when writing the inline assembly insertion, the assembly strings must each be enclosed in double quotes:

```
"ld %0,%3\n\t"
"bset %0,%0,%2\n\t"
...
```

Example 24 shows multiple assembly instructions in a single inline assembly insertion. The inline assembly string is written on a single line with newlines and tabs to format the generated assembly.

Example 24 Multiple Instructions in a Single GCC Inline Assembly Insertion

This example assumes the subroutine **_func** accepts arguments in registers r0 and r1. Note the empty constraint string (:) corresponding to the first assembly string. Replace the … with `"r3"`, `"r4"`, etc., through `r12`.

```
asm (“mov r0,%0\n\tmov r1,%1\n\tbl _func"
   : : "r" (from) , "r" (to) : "r0" "r1", "r2", ... r12);
```

Ensuring the Correct Data Type

GCC-style inline assembly offers two methods for ensuring your assembly operate on the correct data types:

- Declaring a variable
- Using a cast

Declaring a Variable

Example 22 GCC __asm__ Insertion in a Macro on page 195 declares a variable to make sure the assembly instruction operates on the correct data type.

Using a Cast

Using a cast converts more types. For example, if the desired type is **int**, casting the argument to **int** accepts a pointer with no complaint, while simply assigning a pointer to an **int** variable results in a warning about using a pointer not explicitly cast by the caller.

Specifying Instructions That Kill Specific Registers

Some instructions kill specific hard registers. To describe this, use a third colon after the input operands, followed by the names of the killed hard registers (given as strings and separated by commas).

Example 25 Using GCC __asm__ with Instructions That Kill Specific Registers

```
//Example from the Linux kernel
      __asm__ __volatile__ (
   "lr   r20, [status32] \n\t"
   "and  r20, r20, %0 \n\t"
   "flag r20  \n\t"
   :
   :"n" (~(STATUS_E1_MASK | STATUS_E2_MASK))
   :r20");
```

The input operands and the output operands' addresses are guaranteed not to use registers you specify in the killed list, so you can read and write killed registers as many times as you like.

You may not write a kill description in a way that overlaps with an input or output operand. See Example 29 Using GCC __asm__ with Register Variables on page 198.

If you write to a particular hardware register from the assembly code, you generally must list the register after the third colon to inform the compiler that the register's value is modified.

Working with Output Operands

If a GCC inline-assembly insertion has output operands, the compiler 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 might eliminate, move, or replace them if the output operands aren't used. Also, if your instruction has a side effect on a variable that otherwise appears not to change, the old value of the variable might be reused later if it happens to be in a register.

Unless an output operand has the *&* ("early clobber") constraint modifier, the compiler might 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 might be false if the assembly code consists of more than one instruction. In such a case, use *&* for each output operand that must not overlap an input. See Table 7 Constraint Modifiers for GCC-Style Inline Assembly on page 193.

If the output expression cannot be directly addressed (for example, it is a bit field), your constraint must allow a register. The compiler then uses the register as the output of the inline assembly, and stores that register into the output.

Read, Write, and Read/Write Operands

The ordinary output operands must be write-only; the compiler assumes that the values in these operands before the instruction are dead and need not be generated. GCC-style inline assembly supports input/output (read/write) operands. Use the constraint character '*+*' to indicate such an operand and list it with the output operands. Use read-write operands only when the constraints for the operand (or the operand in which some of the bits are to be changed) allow a register.

You may, as an alternative, split the read/write operand into two separate operands, one input operand and one write-only output operand using numeric constraints to express the connection between them.

Using Numeric Constraints to Specify Operand Co-Location

You can use an operand number in the constraint string to ensure that one operand is in the same place as another. Consider the hypothetical assembly string in Example 26 which has `svar` as its read-only source operand and `dvar` as its read/write destination:

Example 26 Hypothetical Instruction without Numeric Constraints

```
asm("combine %0, %2" : "=r" (dvar) : "r" (dvar), "g" (svar)
```

Even though `dvar` is the value of both operands in Example 26 this is not enough to guarantee that they are allocated to the same register in the generated assembly code. Various optimizations or reloading might cause operands 0 and 1 to be in different registers.

Using numeric constraints, you can express the connection between operands in such cases so that they are in the same location when the instruction executes. Here is Example 26 modified to use numeric constraints:

Example 27 Hypothetical Instruction with Numeric Constraints

```
asm("combine %0, %2" : "=r" (dvar) : "0" (dvar), "g" (svar)
```

Using the constraint `0` for operand 1 specifies that this operand must occupy the same register as operand 0. You can use the same C expression for both operands, or different expressions. Note that a numeric constraint is allowed only on an input operand and is must refer to an output operand.

Forcing an Operand into a Specific Register

Sometimes you might need to make an inline-assembly operand be a specific register, but no constraint letter exists for the register. To force the operand into a register, use a local variable for the operand and specify the register in the variable declaration.

Example 28 Using GCC __asm__ to Force an Operand into a Specific Register

```
//An example from the Linux kernel
unsigned long SP asm("sp");
```

Handling Unpredictable Memory Accesses by Listing Memory As Killed

If your assembly instructions access memory in an unpredictable fashion, add *`memory`* to the list of killed registers. This tells the compiler not to keep memory values cached in registers across the assembly instruction and not to optimize stores or loads to that memory.

Add **volatile** as well if the memory affected is not listed in the inputs or outputs of the inline assembly, as the memory kill is not considered as a side effect of the **asm** insertion.

Working with Explicit Register Variables

In the case of register variables, use any register-constraint letter that matches the register as your operand.

Example 29 Using GCC __asm__ with Register Variables

```
register int *p1 asm (“%r0") = some_value;
register int *p2 asm (“%r1") = some_value;
register int *result asm (“%r0");
asm (“add\t%0,%1,%2" : “=r" (result) : “0" (p1), “r" (p2));
```

Analysis

This example declares three variables and assigns them to registers, then copies two values into the `result` register variable.

Note that any register that may be killed by the compiler is overwritten by any function call in the assignment, including library calls for arithmetic operators. If `%r0` is a register permitted to be killed by a call, it might be killed by computing *some_value* for the assignment to `p2`. If you have to use such a

register, use temporary variables for expressions between the register assignment and rewrite Example 29 as follows:

```
   int t1 = some_value;
   register int *p1 asm (“r0") = some_value;
   register int *p2 asm (“r1") = t1;
   register int *result asm (“r0");
   asm("add \tr0,%1,%2\n\trap\n\tmov \t%0,r0\n": "=r" (result) : "0" (p1), "r"
(p2));
```

Working with Condition Codes

It is not possible to reliably access the condition code from code prior to the inline-assembly insertion or leave a condition code for code following the insertion. Better practice is to use a branch instruction to generate a value for an output variable of the insertion, for example:

```
asm("add.f\t 0,%2,%1\n\tmov\t%0,0\n\tbreq\t1f\n\tmov\t%0,1\n1: nop\n"
   :"r" (result): "r" (var1), "r" (var2));
```

This snippet contains multiple assembly strings with tabs and newlines in a single insertion; see Including Multiple Assembly Instructions and Formatting Your Output on page 196 for more information.

Jumps between Insertions Not Permitted

Jumps from one **asm** construct to another are not permitted. The optimizer does not know about such jumps, so it cannot take account of them when deciding how to optimize.

Related Topics

- Syntax on page 192
- Real-World Examples on page 199

Real-World Examples

This section presents examples of how to use the GCC inline-assembly mechanism to accomplish real-world programming tasks on an ARC processor.

These examples share some common characteristics:

- They assume toggle `Asm_percent_reg` is off so that registers may be specified without a '%' prefix.
- They use **volatile** to preserve the instruction from being optimized away; see Using volatile to Preserve Instructions from Optimization on page 195.
- They use tabs and newlines to format the assembly output; see Including Multiple Assembly Instructions and Formatting Your Output on page 196.

Example 30 Using GCC __asm__ to Rename a Global Variable

This example makes the internal name of a variable different from the externally visible name.

```
int  count __asm__("__kernel_setup_counter__")=0;
```

Example 31 Using GCC __asm__ to Halt the Processor

This example halts the ARC processor. Note the use of tabs for clarity, and **volatile** to prevent the instruction being optimized away.

```
void halt_processor()
{
   asm volatile("\tflag\t1"); /* halt processor */
}
```

Example 32 Using GCC __asm__ to Load and Initialize Registers

This example loads the dcache build register and initializes the dcache control register.

```
__asm__ __volatile__(
"lr %0, [%1]\n\t"
"sr 0x40, [%2]\n\t" /* Set IM and set replacement policy as round robin */
      :"=r"(cpuinfo_arc700.dcache)
      :"i"(ARC_REG_D_CACHE_BUILD_REG), "i"(ARC_REG_DC_CTRL)
      );
```

Example 33 Using GCC __asm__ to Disable Interrupts

This example from the Linux kernel uses an **__asm__** insertion to disable interrupts. It encapsulates the inline assembly in a C macro (`__save_flags_cli`). It uses "*n*" to identify an argument as an immediate constant. It also uses **__volatile__** (with GCC underscores supported through the **__attribute__** syntax) to preserve the inline assembly from being reordered or optimized away. Finally, it lists `r20` and *`memory`* as "registers" that are killed by the instructions.

```
#define __save_flags_cli(x)                          \
   __asm__ __volatile__ (                            \
   "lr   r20, [status32] \n\t"                       \
   "mov   %0, r20 \n\t"                              \
   "and   r20, r20, %1 \n\t"                         \
   "flag       r20   \n\t"                           \
   :"=r" (x)                                         \
   :"n" (~(STATUS_E1_MASK | STATUS_E2_MASK))         \
   :"r20", "memory");
```

Note that this example lists a register and memory as killed; see Specifying Instructions That Kill Specific Registers on page 197 and Handling Unpredictable Memory Accesses by Listing Memory As Killed on page 198.

Example 34 Using GCC __asm__ in an Atomic Exchange with Memory

This example is an atomic exchange with memory used to implement a semaphore for multiprocessing.

NOTE You can write **volatile** or **__volatile__** because the MetaWare compiler supports the GCC **__attribute__** syntax. For more on the MetaWare implementation of the **__attribute__** syntax, see the *MetaWare C/C++ Language Reference*.

```
extern inline unsigned long __xchg (unsigned long with,
volatile void *ptr, int size)
{

__asm__ volatile (" ex               %0, [%1]"
               : "=r" (with),"=r"(ptr)
               : "0" (with), "1" (ptr)
               : "memory" );

return (with);
}
```

This example lists memory as a killed "register"; see Handling Unpredictable Memory Accesses by Listing Memory As Killed on page 198.

Example 35 Using GCC __asm__ to Set a Bit in a Shared-Memory System

This example sets a bit on a word. It uses **volatile** so that the code can be used to access shared memory. This example shows the use of various constraint modifiers; see Table 7 Constraint Modifiers for GCC-Style Inline Assembly on page 193.

```
static inline void set_bit(unsigned long nr, volatile void * addr)
{
unsigned long temp, flags;
int *m = ((int *) addr) + (nr >> 5);

save_flags(flags); cli();
__asm__ __volatile__(
   "   ld %0,%3\n\t"
   "   bset %0,%0,%2\n\t"
   "   st %0,%1\n\t"
   :"=&r" (temp), "=m" (*m)
   :"Ir" (nr), "m" (*m));

restore_flags(flags);
}
```

Example 36 Using GCC __asm__ to Copy Memory

This example from the Linux kernel copies a block from the user process to the kernel. It shows the use of labels within the inline assembly, and spreads the inline assembly code among several sections.

```
static inline unsigned long
__generic_copy_from_user(void *to, const void *from, unsigned long n)
{
long res;
const void *tfrom = from;
char val;

if (n == 0)
return 0;
__asm__ __volatile__ (
      "   mov   %0, %3                    \n"
      "1:   ldb.ab      %4, [%2, 1]       \n"
      "   stb   %4, [%1]                  \n"
      "   add   %1, %1, 1                 \n"
      "   sub.f   %0, %0, 1               \n"
      "   bnz   1b                        \n"
      "2:   nop                           \n"
      "   .section .fixup, \"ax\"         \n"
      "3:   j   2b                        \n"
      "   .previous                       \n"
      "   .section __ex_table, \"a\"      \n"
      "   .word   1b, 3b                  \n"
      "   .previous                       \n"

      :"=r"(res), "=r"(to), "=r"(tfrom), "=r"(n), "=r"(val)
      :"3"(n), "1"(to), "2"(tfrom)
   );

return (res);
}
```

Related Topics

- Syntax on page 192
- GCC Extended Asm: Usage Tips and Techniques on page 194

Inserting an Inline Assembly Function Using the AT&T Facility

The MetaWare C/C++ compiler supports the enhanced **asm** facility as defined in the *AT&T UNIX System V Programmer's Guide: ANSI C and Programming Support Tools*. In MetaWare C/C++, to avoid corrupting the ANSI Standard C name space, the keyword **asm** has been renamed to **_Asm**.

The **_Asm** type qualifier declares a set of assembly instructions to be a C function and inserts that set of instructions into the compiler output. A function qualified with **_Asm** can be defined or called with multiple arguments.

Here is an example of an **_Asm** macro:

```
_Asm int do_lr(unsigned short val) {
    % con val
            lr      %r0, [val]
    % reg val
            lr      %r0, [val]; statement comments OK
    % error
    }
```

Using **_Asm** can interfere with the compiler's ability to optimize surrounding code, because the compiler can make no assumptions about the **_Asm** code. An **_Asm** macro can modify only volatile registers. The optimizer assumes that non-volatile registers are not modified by an **_Asm** macro.

Volatile registers `%r10`, `%r11`, and `%r12` are guaranteed to be unused for passing arguments and thus can be freely used within the **_Asm** body. For an illustration using `%r12`, see _Asm Macro with Multiple Arguments on page 206.

CAUTION Be careful when using registers in your **_Asm** body: An input parameter might be contained in any machine register (except `%r10`, `%r11`, or `%r12`). For example, if you explicitly load a parameter into a register, you might overwrite the register used for a second parameter. Safest is to use only `%r10`, `%r11`, and `%r12`.

Similarly, do not modify an input parameter in your **_Asm** body, as the parameter might be in a non-volatile register that the compiler assumes your **_Asm** body preserves. Copy into `%r10`, `%r11`, or `%r12` if you need to modify the value of an input parameter.

The compiler assumes that an **_Asm** macro might modify any of the scratch registers (`%r0` to `%r12`). With this restriction, the compiler is forced to place variables that are alive across the **_Asm** function into saved registers, or on the stack. This can also cause additional prolog and epilog sequences that might not be necessary. To avoid such overhead, use calling-convention attribute **_SAVE_ALL_REGS**. When this calling convention is applied, the compiler assumes the following:

- Only registers `%r10`, `%r11`, `%r12` and `%status32` (or `%status` in the case of ARCtangent-A4) might be written.
- Register `%r0` is written if the **_Asm** function is not a **void** function.

CAUTION If your **_SAVE_ALL_REGS** macro writes any other register, your code might not work as expected.

Example 37 Using _SAVE_ALL_REGS with an _Asm Macro

```
_Asm int _CC(_SAVE_ALL_REGS) func() {
    /* body of _Asm macro here */
}
```

Related Topics

- _Asm Macro Syntax on page 203
- Sample _Asm Macros on page 204
- Calling-convention attribute **_SAVE_ALL_REGS**

_Asm Macro Syntax

MetaWare C/C++ programs use the enhanced **asm** facility by way of **_Asm** macros. This is the syntax of an **_Asm** macro:

```
_Asm [type_spec] identifier([formal_param_list]) {
   %storage_mode_spec  [; storage_mode_spec]
            asm_body
            ...
   [%storage_mode_spec [; storage_mode_spec]
            asm_body
            ... ]
   }
```

NOTE You must use a '%' character at the start of a line containing storage mode specifications.

Components of an _Asm macro on page 203 lists the meanings of the individual components of the **_Asm** macro.

Table 8 Components of an _Asm macro

Component	Meaning
type_spec	Any valid C type (optional).
identifier	Any valid C identifier (required).
formal_param_list	One or more valid C parameters (optional).
storage_mode_spec	Storage-mode specification line (one required); a pattern for the actual parameter list to match. See Table 9 Storage Modes for _Asm Macro Arguments.
asm_body	Portion of assembly code to be expanded when the actual parameter list matches the pattern in *storage_mode_spec*.

Storage-Mode Specification

This is the syntax of each *storage_mode_spec* (also known as a *pattern*):

storage_mode identifier[, *identifier*]

NOTE You must use a '%' character at the start of a line containing storage mode specifications.

Table 9 Storage Modes for _Asm Macro Arguments lists the different storage modes the compiler recognizes.

Table 9 Storage Modes for _Asm Macro Arguments

Mode	Meaning
con	The argument is a compile-time constant.
error	Generate a compiler error.
lab	The argument is a unique label generated by the compiler.
mem	Matches any allowed machine-addressing mode, including *con* and *reg*. Automatically moves the constant to a register.
reg	The argument is a *treg* or *ureg*.
treg	Temporary register selected by the compiler.
ureg	C register variable, allocated by the compiler in a machine register.

Use storage mode *%error* to flag errors at compile time if a set of arguments does not match any of the **_Asm** macro's specified patterns.

Any identifier with storage mode *%lab* must be unique. Such an identifier does not appear as a formal parameter in the definition of the **_Asm** macro.

CAUTION If you use **_Asm** to define a function that has more than one argument, you must specify all the argument declarations on one line. (See _Asm Macro with Multiple Arguments on page 206 for an example.)

Assembly functions you inline using **_Asm** must be defined prior to their use so that the compiler and linker know of their existence.

Passing Memory References

It is possible to pass memory references with the *%reg* storage mode. The compiler loads the memory into a register and uses the *%reg* body. The compiler also loads a constant into a register to match *%reg*, which you can use if it is undesirable to declare a separate body for both *%reg* and *%con*.

However, the compiler selects the first storage-mode specification that satisfies the arguments. Because of this selection order, if you want to use *%con*, you must place the storage-mode specification and body containing *%con* before similar specifications and bodies containing *%reg*.

A complete discussion of this enhanced **asm** facility is beyond the scope of this *Programmer's Guide*. For a more detailed explanation, refer to the "Enhanced asm Facility" appendix in the *AT&T UNIX System V Programmer's Guide: ANSI C and Programming Support Tools*.

Sample _Asm Macros

Example 38 _Asm macro with One Argument

_Asm macro **CLZ()** takes one argument, which must correspond to a value loaded into a register (*%reg*). This example uses calling-convention attribute **_SAVE_ALL_REGS** to tell the compiler to expect changes to no registers other than `r0`, `r10`, `r11`, and `r12`.

```
/* _Asm Macro CLZ() -- Count Leading Zeros */

_Asm _CC(_SAVE_ALL_REGS) int CLZ(long arg) {
   % reg arg; lab loop, done
           mov.f   %r10, arg
           mov     %r0, 32
           beq     done
   loop:
```

```
          lsr.f   %r10, %r10
          sub     %r0, %r0, 1
          bne     loop
   done:
   % error
   }
```

_Asm macro **CLZ()** can be called just like any other C function, as shown here:

```
int x = 20;
void func() {
   CLZ(40);
   CLZ(x + 2);
   }
```

The following examples show the assembly code that the compiler generates for function **func()**, with its two different calls to **_Asm** macro **CLZ()**:

ARCtangent-A4

```
; Assembly code for function calling _Asm macro CLZ()
;
;-------------| func |-------------------
        .global func
func:
        st      %blink, [%sp, 4]
        st      %fp, [%sp]
        mov     %fp, %sp
        sub     %sp, %sp, 16
        mov     %r0, 40
;
; -------- BEGIN _Asm inline code
        mov.f   %r10, %r0
        mov     %r0, 32
        beq     .LASM2.1
.LASM2.0:
        lsr.f   %r10, %r10
        sub     %r0, %r0, 1
        bne     .LASM2.0
.LASM2.1:
; -------- END _Asm inline code
;
        ld      %r0, [%gp, x@sda]
        add     %r0, %r0, 2
;
; -------- BEGIN _Asm inline code
        mov.f   %r10, %r0
        mov     %r0, 32
        beq     .LASM3.1
.LASM3.0:
        lsr.f    %r10, %r10
        sub      %r0, %r0, 1
        bne     .LASM3.0
.LASM3.1:
; -------- END _Asm inline code
;
        ld      %blink, [%fp, 4]
        j.d     [%blink]
        ld.a    %fp, [%sp, 16]
```

ARCtangent-A5 and Later

```
; Assembly code for function calling _Asm macro CLZ()
;
;-------------| func |-------------------
        push_s  %blink
        mov_s   %r0, 40
;
; -------- BEGIN _Asm inline code
        mov.f   %r10, %r0
        mov     %r0, 32
        beq     .LASM2.1
.LASM2.0:
        lsr.f   %r10, %r10
        sub     %r0, %r0, 1
        bne     .LASM2.0
.LASM2.1:
; -------- END _Asm inline code
;
        ld_s    %r0, [%gp, x@sda]
        add_s   %r0, %r0, 2
;
; -------- BEGIN _Asm inline code
        mov.f   %r10, %r0
        mov     %r0, 32
        beq     .LASM3.1
.LASM3.0:
        lsr.f   %r10, %r10
        sub     %r0, %r0, 1
        bne     .LASM3.0
.LASM3.1:
; -------- END _Asm inline code
;
        pop_s   %blink
        j_s     [%blink]
```

Example 39 _Asm Macro with Multiple Arguments

You can write **_Asm** macros that specify more than one argument and account for different types, as shown here:

```
/* _Asm Macro DIV_EX()                          */
/*                                              */
_Asm int DIV_EX(int dividend, int divisor) {
   % reg dividend; con divisor
   ; Assume div32, "divlo", and "div_aux"
   ;  have been defined elsewhere using the
   ;  .extInstruction, .extCoreRegister, and
   ;  .extAuxRegister assembler directives.
   ; Must load into %r10, %r11, or %r12, as
   ;  any other register could be "dividend".
           mov     %r12, divisor
           sr      %r12, [%div_aux]
           div32   0, dividend
           mov     %r0, %divlo
   % reg dividend, divisor
           sr      divisor, [%div_aux]
           div32   0, dividend
           mov     %r0, %divlo
   % error
   }
```

In this example, macro **DIV_EX()** can be called when `divisor` is either a register (`% reg`) or a constant (`% con`). This macro assumes that instruction **div32** has been implemented as an extension, which is not directly supported by the compiler.

_Asm macro **DIV_EX()** can be called as shown here:

```
int func(int a, int b) {
    a = DIV_EX(a, b);
    return DIV_EX(a, 2);
    }
```

The following example shows the assembly code that the compiler generates for function **func()**, with its two calls to **_Asm** macro **DIV_EX()**.

ARCtangent-A4

```
; Assembly for Function Calling _Asm Macro DIV_EX()
;
;-------------| func |-------------------
        .global func
func:
        st      %blink, [%sp, 4]
        st      %fp, [%sp]
        mov     %fp, %sp
        sub     %sp, %sp, 16
;
; -------- BEGIN _Asm inline code
        sr      %r1, [%div_aux]
        div32   0, %r0
        mov     %r0, %divlo
; -------- END _Asm inline code
;
; -------- BEGIN _Asm inline code
 ; Assume div32, "divlo", and "div_aux"
 ;  have been defined elsewhere using the
 ;  .extInstruction, .extCoreRegister, and
 ;  .extAuxRegister assembler directives.
 ; Must load into %r10, %r11, or %r12, as
 ;  any other register could be "dividend".
        mov     %r12, 0x2
        sr      %r12, [%div_aux]
        div32   0, %r0
        mov     %r0, %divlo
; -------- END _Asm inline code
;
        ld      %blink, [%fp, 4]
        j.d     [%blink]
        ld.a    %fp, [%sp, 16]
```

ARCtangent-A5 and Later

```
; Assembly for Function Calling _Asm Macro DIV_EX()
;
;-------------| func |-------------------
        .global func
func:
        push_s  %blink
;
; -------- BEGIN _Asm inline code
        sr      %r1, [%div_aux]
        div32   0, %r0
        mov     %r0, %divlo
```

```
; -------- END _Asm inline code
;
; -------- BEGIN _Asm inline code
 ; Assume div32, "divlo", and "div_aux"
 ;  have been defined elsewhere using the
 ;  .extInstruction, .extCoreRegister, and
 ;  .extAuxRegister assembler directives.
 ; Must load into %r10, %r11, or %r12, as
 ;  any other register could be "dividend".
        mov     %r12, 0x2
        sr      %r12, [%div_aux]
        div32   0, %r0
        mov     %r0, %divlo
; -------- END _Asm inline code
;
        pop_s   %blink
        j_s     [%blink]
```

Inserting a Single Line of Assembly with _ASM()

The MetaWare C/C++ compiler provides the **_ASM()** function for inserting a single line of inline assembly within a function:

_ASM() works only within a function. You can use pragma `Asm` inside or outside of a function. For details about pragma `Asm`, see the listing in Alphabetical List of Pragmas on page 129.

> **NOTE** Using **_ASM()** can compromise performance of your program, because the compiler must be conservative with optimizations around this construct.
>
> We recommend that, whenever possible, you use intrinsic functions supplied by the compiler or defined with pragma `Intrinsic` instead of inline assembly. For more information, see Appendix A — Intrinsic Functions and pragma `Intrinsic`.

A function call of the form **_ASM("*assembly_string*")** places the string *assembly_string* in the assembly output of the compiler at the point of the **_ASM()** call, then appends a newline character.

> **CAUTION** The compiler does not attempt to check the string for validity.

```
/*  MetaWare C/C++ function using _ASM()  */
int identity() {
   /*
    * Return the value of the IDENTITY
    * auxiliary register (Note: this is an
    * example only; use of the _lr() intrinsic
    * is preferred in practice)
    */
   _ASM("lr      %r0,[%IDENTITY]");
   }
```

This example shows assembly code similar to what the compiler generates for the preceding MetaWare C/C++ **_ASM()** function.

```
/* Assembly Code Generated - _ASM() Function func() */
identity:
 lr      %r0,[%IDENTITY]
 j       [%blink]
```

Limitations of _ASM()

The **_ASM()** function can appear only where a function call is permitted; that is, within a function definition. The **_ASM()** function cannot be used to insert assembly instructions outside of a function.

To insert inline assembly code outside a function, use pragma `Asm`.

The only machine registers that can be modified in an **_ASM()** statement are those that can be modified by a call. By default, registers used within inline-assembly insertions must be specified with a '%' prefix. To eliminate this requirement, turn off toggle `Asm_percent_reg`.

CAUTION Use **_ASM()** sparingly. Never use it to alter control flow across C constructs.

Branch Instructions and Function _ASM()

You cannot reliably use branch instructions with a MetaWare C/C++ **_ASM()** insertion, because you might introduce control paths into the program flow without the compiler being aware of those paths. Such control paths can cause the compiler to generate incorrect code.

Discarding Dead _ASM() Code

Based on control-flow analysis, the compiler might discard **_ASM()** statements that appear to be dead (never execute).

For example, the three **_ASM()** statements after the **return** statement in the following code are discarded because they are dead (never execute):

```
int func() {
   _flag   (0x06);
   return;
   _ASM("        mov     %r0, 0");
   _ASM("        mov     %r1, 1");
   _ASM("        jal     [%ilink2]");
   }
```

The following assembly code is similar to what the compiler generates for this example:

ARCtangent-A4

```
func:
        flag    0x06
        j       [%blink]
```

ARCtangent-A5 And Later

```
func:
        flag    0x06
        j_s     [%blink]
```

Chapter 10 — Embedded Development

In This Chapter

- Using a Stack
- Using a Heap
- Using Small-Data Sections
- Initializing Memory
- Working with Volatile and Uncached Variables
- Writing Interrupt Handlers
- Adapting the Libraries to an RTOS
- Determining and Eliminating Library Function Calls
- Working with Startup Modules
- Providing System Calls for Debugging from a Host
- Providing Low-Level System Functions
- Building Thread-Safe Applications
- Rebuilding the C Run-Time Libraries
- Providing Support for Statistical Profiling
- Using Closely Coupled Memory
- Using Position-Independent Code and Data

Overview

This section covers points of special interest for developing embedded applications with the MetaWare compilation toolchain.

Using a Stack

The processor requires a stack to run any compiler-generated code. By default, the compiler places the stack in ELF section `.stack`.

Using the Default Stack

The supplied run-time startup module `crt1.o` (which contains the program entry point) initializes the stack pointer (`%sp` register) to point to the end of the `.stack` section (symbol `_estack`). If you receive an error stating that `_estack` is an undefined symbol, you need to define a `.stack` section in your linker command file.

The stack grows from high address to low address, toward the start of the `.stack` section. For more information about stack structure and mechanics, see Stack Conventions on page 276.

Specifying Stack Size

The stack size required varies widely. The default linker command file sets the stack size to 64 K. If you are using an RTOS where the initial stack is not used after application tasks or threads are created, you might require considerably less stack.

CAUTION Make sure the stack size is large enough for your application. The compiler does not provide run-time stack checking.

You can specify the size of the stack section using driver option **-Hstack**.

If you link using driver option **-Hstack** and want to write your own linker command files, link once using the default command file and use linker option **-m** (documented in the *ELF Linker and Utilities User's Guide*) to see the default command file. Then copy the line that creates the `.stack` section into your own command files.

You can also locate and size the `.stack` section using the linker command file. For more information about working with linker command files, see the *ELF Linker and Utilities User's Guide*.

Aligning Stack-Based Variables

Pragma `Align_to` does not affect stack-based data structures. To align storage on the stack, you must provide code to allocate stack using **alloca()**, test for alignment, and realign your structure as needed, as shown in Example 40

Example 40 Aligning Data Storage on the Stack

```
#include <alloca.h>
...
...
stack_align_struct_type * tsp;
char * tempp = alloca(sizeof(stack_align_struct_type ) + MIN_ALIGN);
   tempp = tempp + MIN_ALIGN - ((unsigned)tempp % MIN_ALIGN);
    tsp = (stack_align_struct_type * ) tempp;
```

Using a Heap

This section describes methods for specifying a memory heap. Your application might require a heap or a larger heap for any of several reasons:

- Some functions in the MetaWare C Run-Time Libraries use a heap.
- Memory-allocating function **malloc()** uses a heap.
- Statistical profiling using driver option **-p** or **-pg** uses a potentially large heap.

The MetaWare C/C++ compiler does not generate a heap by default; it generates a heap if you use driver option **-Heos**. The size of the heap depends on the RTOS given as an argument to **-Heos** (see the listing for **-Heos** for sizes). If you want to use a heap other than the default size for your RTOS, you must specify a heap using one of the methods described below.

For more information on profiling, see Execution Profiling on page 293; the concept of a heap is described in general in Appendix D — The Heap Manager.

Specifying a Heap

You can specify use of a heap in the following ways:

- Use the value of symbol `_end` as the start of the heap (default method).
- Use a separate ELF section (`.heap`) to contain the heap.
- Implement your own version of low-level memory allocator **_sbrk()** using RTOS memory-allocation functionality.

Using Symbol _end

The default implementation of **_sbrk()** in the MetaWare C Run-Time Libraries assumes that the heap starts at the first address after the address space used by the program (the first address after `_end`) and ends at the end of memory. When using this method, you might need to tell the debugger how much total memory is available on your system.

Using a Separate ELF Section

For more control over the size and location of heap memory, you can restrict the heap to ELF section `.heap`. Driver option **-Hheap** specifies a heap of the size you request.

If you compile using driver option **-Hheap** and want to write your own linker command files, link once using the default command file and use linker option **-m** (documented in the *ELF Linker and Utilities User's Guide*) to see the default command file. Then copy the line that creates the `.heap` section into your own command files.

You can also manually locate and size the `.heap` section in your command file so that the run time confines the heap to the ELF section you specify. The default implementation of **_sbrk()** in the MetaWare C Run-Time Libraries allocates from the `.heap` section if that section exists.

Implementing Your Own Low-Level Memory Allocator

The MetaWare C/C++ interface between system-specific memory allocation and the run-time library is function **_sbrk()**. You can implement any means of dynamic memory allocation you need by supplying your own version of **_sbrk()**. This might be a suitable solution if, for example, your RTOS already manages free memory the way you need and provides API calls to allocate and free memory. See also the description of function **_sbrk()** in Providing Low-Level System Functions on page 230.

Using Small-Data Sections

The MetaWare C/C++ compiler supports small-data sections. When the compiler allocates storage for data in a small-data section, references to the data are smaller and faster, because such data can be addressed with small signed offsets from the base of the small-data section. By default, the compiler creates small-data sections `.sdata` and `.sbss`. You can also create your own small-data sections as described in Adding Your Own Small-Data Sections on page 218.

Normally, the compiler either uses a two-instruction sequence to access data, or loads data from memory using direct access, which is an 8-byte instruction. Each of these code sequences involves more overhead than small-data code.

Variables placed in a small-data section can be referenced by smaller code because the base address of each small-data section is contained in register `r26`, also named `%gp`. Register `%gp` is initialized during run-time startup to point to a location in the `.sdata` section, and thereafter is never modified by the compiler or run time. Therefore, when a variable is referenced, only its offset needs to be loaded. This can be done using a single four-byte instruction (ARCtangent-A4) or possibly a two-byte instruction (ARCtangent-A5 and later).

Because the size of a small-data section is limited, the compiler places only variables four bytes and smaller in small-data sections. You can change this default using pragma `Push_small_data`.

The compiler assumes that variables of suitable size that are declared but not defined (for example, variables declared **extern**) are placed in one of the small-data sections. That is, small-data code is generated for such a variable, even though the variable is not defined. **auto** variables cannot be placed in small-data sections.

Related Topics

- Placing Data in Small-Data Sections on page 214
- Scope of Small-Data Specifications on page 216
- Limitations of Small-Data Specifications on page 217
- How Small-Data Sections Are Accessed on page 217
- Handling Link Problems Due to Size of Small Data on page 218
- Adding Your Own Small-Data Sections on page 218
- Adding Your Own Small-Data Sections: Example on page 220

Placing Data in Small-Data Sections

This section describes how to place data in small-data sections.

- The Easiest Way to Place Data in Small-Data Sections
- Compiler Controls for Small-Data Sections

NOTE Ensure that you place variables in small data in the same compilation unit where each variable is defined. At link time, the linker ensures that variables referenced with small-data code are, in fact, in a small-data section.

NOTE The compiler does not place variables that you align using pragma `Align_to` or **__attribute__ aligned** in small data unless you qualify them as **_Near**.

The Easiest Way to Place Data in Small-Data Sections

The easiest way to use small-data sections is to employ the same setting for pragma `Push_small_data` for all compilation units, and invoke it at the command line rather than in the code.

For example, this command line applies the `Push_small_data` setting to modules `f1` and `f2`:

```
hcarc f1.c f2.c -Hpragma="Push_small_data(8;12)" -c
```

For finer-grained control, you can bracket blocks of source code with pragmas `Push_small_data` and `Pop_small_data`.

NOTE To avoid link errors, put pragma `Push_small_data` in any header file that contains the declaration of variables shared among compilation units.

Pragma `Push_small_data` places data in small-data sections `.sdata` and `.sbss`. Instructions referencing these sections use a dedicated base register whose contents remain unchanged throughout application execution. The compiler uses section `.sbss` for uninitialized non-**const** variables and section `.sdata` for initialized **const** and non-**const** variables.

Compiler Controls for Small-Data Sections

You can manage the assignment of data to a small-data section with the compiler controls listed in this section.

NOTE You must compile with the same small-data specifications all modules that contain the definition and use of small-data variables.

Use This Control	To Do This
Pragma `Push_small_data`	Specify size constraints for small-data variables. The first number is the size in bytes of writable data; the second is the size in bytes of read-only data. The default configuration of the compiler automatically turns on pragma `Push_small_data(4;4)`.
TIP You can place a **struct** containing several member fields in small data by using pragma `Push_small_data` and increasing the size of the first number accordingly.	
Pragma `Pop_small_data`	Pop the most recent `Push_small_data` specification and restore the previous small-data specification.
Option **-Hall_sdata**	Place all data in small-data sections.
Toggle `Common_in_small_data`	Assign common variable declarations to small-data sections, removing the common semantics.
Toggle `Unsized_arrays_in_small_data`	Assume that external arrays declared with no size are in small data.

Use This Control	To Do This
Toggle `Use_sdata`	Respect the designation of `reserved` applied to small-data base register `%gp`.
Option **-Hsdata0**	Generate code that does not use small-data sections but respect the designation of register `%gp` as reserved so it may be used by other modules as the small-data base register.
Option **-Hnosdata**	Generate code that does not use small-data sections and use the small-data base register `%gp` as a general-purpose register.

Related Topics

For details about the syntax and usage of these controls, see the following:

- Alphabetical List of Pragmas on page 129
- Driver Option Reference on page 42
- Alphabetical List of Compiler Toggles on page 86
- Adding Your Own Small-Data Sections on page 218

Scope of Small-Data Specifications

Until it is explicitly respecified, a specification for a given small-data section applies for all subsequent variable definitions in the same compilation unit, even if a different `Push_small_data` specification (or no specification) is in effect at the point where you declare a variable with **extern**.

For example:

```
#pragma Push_small_data(8)
/* Setting is (sdata,8). Assign all non-const variables of size <= 8 to section
.sdata.  */
extern double i, j;
#pragma Pop_small_data();
/* Setting is still (sdata,8) for i and j. */
...
#pragma Push_small_data(4)
/* Setting is (sdata,4) */
...
double i = 7.0;      /* Put in section .sdata. */
double j;            /* Put in section .sbss.  */
```

Note that the `Push_small_data` specification at the external declaration of `i` and `j` was overridden in favor of the specification that was in effect when `i` and `j` were defined.

The compiler assigns a variable to the small-data section that was in effect when the variable was defined. The current small-data section has no effect on variables that are declared but not defined (for example, variables declared **extern**). This is because the small-data section, if any, to which they are assigned must be specified in the compilation unit where they are defined.

For example, a module could contain the following code:

```
#pragma Push_small_data(8)
extern const a,b,c;
extern char *p;
extern int **q;
#pragma Pop_small_data()
```

even though `a`, `b`, and `c` might be assigned to one small-data section and `p` and `q` to another.

Limitations of Small-Data Specifications

At global scope, you cannot define variables subject to small-data specification more than once. For example, after a statement **int** `i;` defines `i`, you cannot make another such definition if a small-data specification is in effect. For information about how to preserve "common" semantics for such declarations, see toggle `Common_in_small_data` in Alphabetical List of Compiler Toggles on page 86.

The small-data sections occupy the boundary between initialized and uninitialized data in the data segment of a file.

For executables, data in both global and local scopes can be assigned to a small-data section.

In the ARCtangent-A4 architecture, the maximum size of a small-data section is 512 bytes. In the ARCompact architecture, the maximum size is 2 K bytes.

However, access to one-byte variables is still limited to a 512-byte maximum, with half-word access limited to 1 K maximum and 32-bit word access spanning the full 2 K maximum. For ARCompact applications the linker groups byte variables together to try to keep them within 512 bytes of the small-data base register. It also groups half-word variables to try to keep them within the 1 K maximum distance from the base register.

How Small-Data Sections Are Accessed

In the ARC architecture, a direct access to load or store a variable's contents requires the use of a four-byte long-immediate (LIMM) instruction field to contain the address of the variable. For example:

```
ld   r0, [my_var]
```

This instruction is an eight-byte instruction with four bytes for the instruction word and an additional four bytes of long-immediate data following the instruction word.

A small-data section is a data section accessed relative to a fixed base register anchored to either the beginning or middle of the section. When accessing a variable in a small-data section, load and store instructions can use a base register and displacement instead of the full address. For example:

```
ld  r0, [gp, my_var@sda]
```

The attribute `@sda` tells the assembler to interpret `my_var` not as a full 32-bit address requiring a total of 8 bytes for the load instruction, but as a nine-bit displacement requiring only a 4-byte instruction. Thus, using a small-data section tends to reduce the code size and improve the performance of your application.

Using Symbol _SDA_BASE_ to Access Small-Data Sections

The base address of the small-data area for sections `.sdata` and `.sbss` is represented by the symbol `_SDA_BASE_`. Register `r26` (`gp`) contains the value of `_SDA_BASE_`.

If you are supplying your own replacement for the run-time startup module `crt1.o`, you must initialize the `gp` register to the value of `_SDA_BASE_` if you are using small data.

The existing code looks like this:

```
mov   gp, _SDA_BASE_   ; initialize small-data base register
```

For more information on how to write your own replacement run-time startup module, see Working with Startup Modules on page 227.

Related Topics

- Working with Startup Modules on page 227
- Placing Data in Small-Data Sections on page 214

Handling Link Problems Due to Size of Small Data

If the small-data sections available are not large enough for your small data, the linker fails to link your application and reports an error such as `Size of .sdata+.sbss segments exceeds ... byte limit.` In this situation, your options include the following:

- Eliminating Small Data Entirely
- Reducing the Size of Small-Data Variables
- Adding Your Own Small-Data Sections

Eliminating Small Data Entirely

If you wish to build your application without small data, you can turn off small data using the **-Hsdata0** or **-Hnosdata** compiler options.

- **-Hsdata0** causes the compiler to refrain from placing variables in small data, but still keeps the `%gp` register reserved so that your application remains compatible with other code in your application that uses small data.
- **-Hnosdata** turns off small data and frees the `%gp` register for use as a general-purpose register.

Reducing the Size of Small-Data Variables

The default setting of the compiler is to place global and static variables four bytes and smaller into a small-data section. You can use pragma `Push_small_data` to change this default size. The following example reflects the default setting:

```
pragma push_small_data(4;4);
```

The first number is the size in bytes of writable data; the second is the size in bytes of read-only data such as string literals and **const** data. If you are only a little over the small-data limit, you might benefit from changing the read-only number to zero and leaving the size of writable data as it is.

After altering the small-data criteria, you can restore the previous settings with the following:

```
pragma Pop_small_data();
```

Related Topics

- Adding Your Own Small-Data Sections on page 218
- Adding Your Own Small-Data Sections: Example on page 220

Adding Your Own Small-Data Sections

You can use pragmas to create additional small-data sections and store variables in them regardless of size.

NOTE Your application must initialize any additional small-data base registers that you define before it can access any of the data in the associated small-data sections. See Initializing Your Small-Data Base Register on page 219.

To create small-data sections, you must reserve an additional small-data base register for each small-data section. The default small-data base register is `r26` (`%gp`). We recommend that registers reserved for additional small-data sections start from `r25` and work downward as needed, as these

registers are less frequently used by the compiler. You can also reserve a scratch register like `r8` or `r7` for this purpose.

The following example shows how to reserve a small-data base register and create a small-data section using that register as the base register:

Example 41 Creating a New Small-Data Section

```
pragma reg_reserve(25);
pragma data("my_sdata", COMMON_DATA, sda_base => 25);
unsigned var1, var2, var3;
struct my_struct s;
...
pragma data(); /* Terminates specification */
```

Analysis

This example does the following:

1. Uses pragma `Reg_reserve` to reserve register r25.
2. Uses pragma `Data` to allocate a new storage block `my_sdata` of type `COMMON_DATA` and associates `r25` as its small-data base register.
3. Declares three **unsigned** variables and a **struct** to be placed in `my_sdata`.
4. Terminates the specification with `pragma data();`

The main difference from other uses of pragma `Data` is use of the property *`sda_base =>`* to define the section as a small-data section and indicate its base register, r25 in this case.

Note that when you use pragma `Data` in this way, pragmas `Push_small_data` and `Pop_small_data` cannot be used to control the inclusion of the data in the new small-data section; all data declared within the scope of pragma `Data` are placed in the new small-data section regardless of size.

Specifying the Data Type

You can specify *`DATA`* or *`COMMON_DATA`* as the data type.

- Specify type *`COMMON_DATA`* if you want to access the data from more than one compilation unit (source file), and place pragma `Data` and the subsequent data declarations in a header file. When you specify *`COMMON_DATA`*, each compilation unit that uses the data defines the data, but the linker includes only one copy of the data in the linked executable.
- Specify type *`DATA`* if the declarations are local to a compilation unit.

Initializing Your Small-Data Base Register

During application startup, all registers reserved as small-data base registers must be initialized with the correct address. Register `%gp` is initialized by the run-time file `crt1.o`. Your application must initialize any additional small-data base registers that you define before it can access any of the data.

For each small-data section created with pragma `Data`, the linker defines a symbol that contains the correct address to use for initializing the small-data base register for the section. The name of the symbol is the same as the section name you specify, but with the prefix `_b` (if your section name starts with a leading '.', the '.' is omitted from the base symbol). The correct way to initialize the base register for the small-data section created in Example 41 is shown in Example 42 (C) and Example 43 (assembly). You can modify this code to match the symbol name assigned to your own new small-data section.

Example 42 Initializing the Base Register for a Small-Data Section (C)

```
extern char _bmy_sdata[];
_core_write((unsigned)_bmy_sdata, 25);
```

Example 43 Initializing the Base Register for a Small-Data Section (Assembly)

```
mov r25 _bmy_sdata
```

Note that symbol `_bmy_sdata` does not necessarily point to the beginning of section `my_sdata`. Because the load and store instructions that use small-data references have a signed displacement, the base register is often positioned in the middle of the section so that both positive and negative displacements can be used to map variables into the section.

Related Topics

- Placing Data in Small-Data Sections on page 214
- Adding Your Own Small-Data Sections on page 218
- Adding Your Own Small-Data Sections: Example on page 220

Adding Your Own Small-Data Sections: Example

Your distribution includes an example that adds and tests four small-data sections.

Example Files

install_dir`/examples/small_data` contains the following files:

`a4sda.c`	Test program for ARCtangent-A4
`makefile`	Makefile for building the test program
`readme.txt`	Information on using small data, much like this topic
`sda1.c`	Test program for ARCtangent-A5 and later (one of two files); initializes, reads, and verifies each variable in each small-data section defined in `sda.h`.
`sda2.c`	Test program for ARCtangent-A5 and later (two of two files); initializes, reads, and verifies each variable in each small-data section defined in `sda.h`.
`sda.h`	Defines small-data areas and places variables in them for the ARCtangent-A5 and later test program.

Building and Running the Test Program

1. `cd` *install_dir*`/examples/small_data`
2. `make -f makefile`

   The makefile builds the test program `a4sda.out` for ARCtangent-A4 and `sda.out` for ARCtangent-A5 and later, if both licenses are available. If one is not, you receive an error for that target.
3. `runac sda.out` (ARCtangent-A4: `runarc a4sda.out`)

The test program prints the following to `stdout`:

```
Testing r22_sec
Testing r23_sec
Testing r24_sec
Testing r25_sec
Testing complete
```

What the Test Program Does

The test program does the following:

1. In the header file, reserves four registers for small data using pragma `Reg_reserve`. For example: `pragma reg_reserve(22);` reserves `%r22`.
2. Uses pragma `Data` to place variables in small data and uses the `sda_base =>` argument to set the small-data base register for each section, for example:

   ```
   pragma data("r22_sec", COMMON_DATA, sda_base => 22);
       char b_r22_sec1;
       char b_r22_sec2;
       char b_r22_sec3;
       ...
   pragma data() /* Terminates specification */
   ```

   Note that the test program for ARCtangent-A5 and later specifies `COMMON_DATA` as the section type because the data are accessed from different source files. The test program for ARCtangent-A4 specifies `DATA` as the section type because it consists of a single source file.
3. Initializes small-data base registers using compiler-intrinsic function **_core_write()**. For example: `_core_write((unsigned)_br22_sec, 22);`
4. Stores values into the registers reserved in the header file.
5. Loads each value from the registers if they are what was stored there.
   — Prints an error message if the values are not equal.
6. Compares each value loaded from each register with what was stored there.
   — Continues, or prints an error message if the values are not equal.
7. Prints out `Testing complete` when done.

About the Makefile

The makefile builds test programs for both ARCtangent-A4 and ARCtangent-A5 and later.

It uses driver option **-c** to produce object files in the first step, then combines the object files into an executable, specifying the name with driver option **-o**.

Related Topics

- Placing Data in Small-Data Sections on page 214
- Adding Your Own Small-Data Sections on page 218

Initializing Memory

You might need to place pre-initialized data in a read-only section (such as ROM) and have the run-time startup code copy it to the location in RAM where the executable expects to find it. The MetaWare C/C++ toolset meets this need with the **INITDATA** feature of the linker. The **INITDATA** feature is documented more completely in the *ELF Linker and Utilities User's Guide*.

The **INITDATA** feature stores the data to be copied in compressed form by default. It typically compresses code by about 40%, and literals by about 50%. It often compresses data by over 70% because of frequent zeroes.

If your code runs faster in volatile memory (RAM) than it does in non-volatile memory, you can store the executable code in compressed form as well and copy it to RAM during startup (leaving a small

amount of startup code uncompressed so that execution can proceed to the **_initcopy()** function prior to decompression). See the *ELF Linker and Utilities User's Guide* for example linker command files.

NOTE Compressing code is not compatible with closely coupled memory (a Harvard architecture).

You can also use **INITDATA** to zero out uninitialized data sections such as `.bss` and `.sbss`. See the *ELF Linker and Utilities User's Guide* for example linker command files.

Working with Volatile and Uncached Variables

This section discusses issues involving the data cache and how to safely bypass the data cache to ensure that the correct values are accessed with volatile and uncached variables. If you are interfacing to hardware using C/C++, you must declare the registers and variables involved in memory-mapped peripheral-register accesses as **_Uncached**. As long as memory-mapped data pointers are declared correctly, the compiler generates the appropriate code for all versions of the ARCtangent processor, whether or not they include a data cache.

CAUTION Declaring a stack-based variable as **_Uncached** or declaring it as **volatile** when toggle `Volatile_cache_bypass` is on (the default) causes cache-coherency problems.

Data-Cache Issues on ARC Processors

Because ARC processors do not go to the overhead of comparing cache data with its source each time it is read, variables type-qualified as **volatile** can result in cache-coherency problems in a processor's data cache. By default, the **volatile** qualifier applied to a variable causes that variable to be written to main memory, not cache. Subsequent reading of the cache then results in unexpected values (*cache incoherency*). This is especially a problem with memory-mapped peripherals.

Cache Incoherency: Example of a Volatile Bit Field

Because of the semantics involved in loading and storing bit fields, declaring them **volatile** is inefficient and not portable and is not the recommended means to access data on hardware. This situation is particularly common when using memory mapped to an external device (memory-mapped peripheral registers).

Bit-field operations cannot safely handle **volatile** variables, because the use of the **volatile** type qualifier implies accesses to the surrounding bit fields that might, for example, change the behavior of a read-once-and-clear status port next to a general I/O data port that is reading a stream of bits.

Avoid using a volatile bit field for an I/O device, as shown in Example 44 Danger of Using Volatile Bit Fields.

Example 44 Danger of Using Volatile Bit Fields

```
struct IOstr {
   int a;
   volatile int b: 1; //Danger of incorrrect data
   int c;
} x;
```

If you compile an application containing the example **struct IOstr** with toggle `Volatile_cache_bypass` turned on, you might encounter cache coherency problems when **x.a** is

written, **x.b** is written using a **.di** instruction suffix, and the cache line containing **x.a** is then flushed back. A safer solution is to make the entire **struct _Uncached**.

Using Library Functions for Volatile Copies

The MetaWare C Library contains functions for copying data to and from volatile destinations and sources, using the **.di** suffix for loads and stores as appropriate.

_vdmemcpy() — Copy to a volatile destination
_vmemset() — Duplicate a character to a volatile destination
_vsdmemcpy() — Copy from a volatile source to a volatile destination
_vsmemcpy() — Copy from a volatile source

See the *MetaWare C Library Reference* for listings.

Using the Volatile Type Qualifier

Compiler toggle `Volatile_cache_bypass` is on by default to cause accesses to **volatile** variables to bypass the data cache in the memory controller by using the **.di** instruction suffix for all loads and stores of data declared **volatile**. For added safety, turn on toggle `Uncached_in_own_section` (on by default for ARCtangent-A5 and later) to place **volatile** variables in a separate `.ucdata` section and reduce the possibility they might be loaded into a cache line.

> **NOTE** The Precise/MQX™ RTOS turns off toggle `Volatile_cache_bypass` because MQX guarantees cache coherence through its own macros. Use of the **_Uncached** type qualifier is unaffected.

You can also turn on toggle `Align_uncached_section` (on by default) to pad the `.ucdata` section to a multiple of 256 bytes (the size of the largest cache line envisioned). This padding ensures that the linker does not combine uncached data with other data on an address boundary that might overlap a cache line boundary.

Example 45 Using the Volatile Qualifier to Generate the .di Suffix shows the code generated for a variable declared **volatile**.

Example 45 Using the Volatile Qualifier to Generate the .di Suffix

C/C++

```
#include <stdio.h>
volatile int a;
int b;

int main(){

   a= 5;
   b= a+3;
}
```

Assembly

```
mov   r0,5
st.di r0,addressA

ld.di r0,addressA
add   r0, r0, 3
st    r0,addressB
```

Using the _Uncached Type Qualifier

MetaWare C/C++ for ARC provides the **_Uncached** qualifier, described in the *MetaWare C/C++ Language Reference*. Declaring variables **_Uncached** allows you to exclude data-cache participation in load and store operations on those variables without involving the additional semantic implications of **volatile**. The **.di** instruction suffix is used for all loads and stores of data declared **_Uncached**.

You can turn off toggle `Volatile_cache_bypass` to restrict **volatile** to its standard C/C++ meaning, and use **_Uncached** to keep variables out of cache.

Toggle `Uncached_in_own_section` (on by default for ARCtangent-A5 and later) places **_Uncached** variables in the separate ELF section `.ucdata`, keeping them separate from other data for safer access.

Toggle `Align_uncached_section` (on by default) pads the `.ucdata` section to 256 bytes to ensure that the linker does not locate uncached data near other data on an address boundary that might overlap a cache-line boundary. You can also locate the `.ucdata` section at a safe distance from other data by using a linker command file; see the *ELF Linker and Utilities User's Guide* for an example.

If your intent is to always load or store a certain variable without using the cache but you do not wish to treat all volatile variables this way, type-qualify the variable as **_Uncached** and turn off toggle `Volatile_cache_bypass`.

NOTE The **volatile** qualifier for ARCtangent-A5 and later is similar to the **volatile** qualifier for ARCtangent-A4. Both generate cache-bypass instructions by default. However, for ARCtangent-A5 and later, the compiler also places **volatile** variables in the `.ucdata` section by default, while for ARCtangent-A4, you must turn on toggle Uncached_in_own_section to place **volatile** variables in the `.ucdata` section.

For More Information

For more information, see the following:

- How the Compiler Maps Variables to Storage on page 252
- Toggle `Align_uncached_section` on page 87
- Toggle `Uncached_in_own_section` in page 122
- The *Cache Reference*, *Programmer's Reference*, and *System Reference* for your ARC processor
- The *ELF Linker and Utilities User's Guide*

Writing Interrupt Handlers

You can write interrupt handlers in assembly or C/C++. This section discusses writing interrupt handlers in C/C++ when using the MetaWare C/C++ toolset.

MetaWare C/C++ provides calling convention attributes **_INTERRUPT1** and **_INTERRUPT2** to type-qualify a function as an interrupt handler.

Interrupt handlers differ from other functions in the following ways:

- Interrupt handlers should not return a value or take any input parameters.
- All registers used by the handler are saved and restored (not just the non-volatile registers).
- Interrupt handlers return using the `ilink1` or `ilink2` link registers (as opposed to the `blink` register used for other calls).

Interrupt handlers are most efficient if they do not make any calls to other functions. Then the compiler can save and restore only the minimum number of registers required to execute the handler. If interrupt handlers make calls, the compiler must save and restore all volatile registers (unless the calls also contain the **_SAVE_ALL_REGS** calling convention attribute, which instructs the compiler to save both volatile and non-volatile registers).

If your interrupt-handler entry point is written in assembly language but you want to call into C or C++, use calling-convention attribute **_SAVE_ALL_REGS** for your C or C++ function.

If you do not use **_SAVE_ALL_REGS** for your C or C++ function, your assembly-language interrupt handler entry point must take a worst-case view of which registers might be modified. If you use calling-convention attribute **_SAVE_ALL_REGS**, the compiler knows exactly which registers must be saved and restored and can generate the assembly automatically.

CAUTION Reenabling interrupts within an interrupt handler of the same level (1 or 2) might result in an infinite loop: if an interrupt of the same level is triggered (or the original triggering cause is not cleared), the context of the second interrupt replaces the context saved in the `ilink1` or `ilink2` register. Thus when the second interrupt returns to the first interrupt, the first interrupt has no context to return to in the calling program. The result is that the first interrupt effectively returns to itself infinitely. Reenable only interrupts of a higher level within an interrupt handler.

Use special caution when writing interrupt handlers. Because interrupt handlers run asynchronously, they should not use many run-time routines and features, such as the following:

- **malloc()**
- **new()**
- **delete()**
- **printf()**
- C++ exceptions

Instead, you can queue the event and post a condition variable for a thread to handle, or use one of the functions listed as reentrant in the *MetaWare C Library Reference*.

Reserving Registers for Fast Interrupt Handling

You can use option **-Hreg_reserve** to reserve registers for exclusive use by interrupt handlers. When the registers are reserved, the compiler never generates code that uses the registers, so you can use them freely without saving and restoring them.

You can also use pragma Core_register to remove, add, or reserve registers.

Generating Software Interrupts

If you are targeting ARCtangent-A5 or later or ARCtangent-A4 core version 8 or greater, you can use compiler-intrinsic function **_swi()** to generate a software interrupt using the software interrupt instruction (**SWI**).

Macros

For convenience, MetaWare C/C++ provides macros for declaring interrupt handlers. These macros allow you to declare calling-convention attributes as if they were type qualifiers.

Macro	Expands to
_Interrupt1	`_CC(_INTERRUPT1|_DEFAULT_CALLING_CONVENTION)`
_Interrupt2	`_CC(_INTERRUPT2|_DEFAULT_CALLING_CONVENTION)`
_Save_all_regs	`_CC(_SAVE_ALL_REGS|_DEFAULT_CALLING_CONVENTION)`

The following is a usage example:

```
_Interrupt2 void handler_6(void) {
...
}
```

See The Calling-Convention Type Qualifier on page 315 for information on the calling-convention type qualifier **_CC**.

Providing Space for Extensions to Save State

You can use a combination of compiler-intrinsic functions and pragma `Interrupt_ext_words` to save and restore additional state from your interrupt handlers written in C or C++.

Use pragma `Interrupt_ext_words` to provide the extra space in the interrupt frame for saving extension state. Use this pragma in conjunction with the following compiler intrinsic functions to save and restore extension state from your interrupt handlers.

- **_interrupt_ext_addr()**
- **_lr()**
- **_sr()**
- **_core_read()**
- **_core_write()**

Library Functions

MetaWare C/C++ provides two library functions to set interrupt vectors in the interrupt vector table. They are declared in `stdlib.h`.

```
extern void _setvect1
(int vector, _Interrupt1 void (*target)());
extern void _setvect2
(int vector, _Interrupt2 void (*target)());
```

The input `vector` is typically a number between 0 (zero) and 15 for a base-case processor (but is not restricted to that range). Input *target* is the interrupt handler function. Here is a usage example:

```
_setvect2(6, handler_6);
```

Adapting the Libraries to an RTOS

If the MetaWare C/C++ toolset does not explicitly support your RTOS, or if you are writing your own operating environment, you might need to write functions to interface between the MetaWare C Run-Time Libraries and your RTOS or operating environment.

For more information, see the following:

- Providing Low-Level System Functions on page 230
- Working with Startup Modules on page 227
- Porting an Interface for RTOS Thread Safety on page 238
- Providing Support for Statistical Profiling on page 242
- *MetaWare C Library Reference*

Determining and Eliminating Library Function Calls

To build a self-contained application that uses no libraries, or to find out which library functions an application references, use option **-Hnolib**.

The linker scans the libraries to resolve symbols not found in the ELF objects given in the link command. If you compile and link using option **-Hnolib** and your code does not explicitly resolve a label, the linker issues an error indicating the library functions that your application is attempting to reference; if the linker does not issue an error, your build is self-contained.

Working with Startup Modules

The source files for these modules are located in directory *install_dir*/`lib/src/c/startup_g`. If the code provided does not meet your needs, you must modify or write the run-time startup yourself.

If you are supplying your own replacement for the run-time startup module `crt1.o`, you must initialize the `%gp` register to the address of symbol `_SDA_BASE_` if you are using small data. To find the value of `_SDA_BASE_`, use the symbol-list utility (**nmarc** for ARCtangent-A4 or **nmac** forARCtangent-A5 and later). For information on how to use the symbol-list utility, see the *ELF Linker and Utilities User's Guide*.

If you modify or replace only file `crt1.o`, specify your version using driver option **-Hcrt**=*path_or_file_name*, where *path_or_file_name* tells the driver where to look for your version of the object file.

If you have a stand-alone assembly application and you do not need any of the supplied run-time initialization, specify driver options **-Hnocrt** and **-Hnolib** to suppress inclusion of all run-time startup and library code from the link.

For more information, see the following:

- *ARC ELF Linker and Utilities User's Guide*
- Chapter 4 — Using Driver Options
- Compiling and Linking in Two Steps on page 29

Startup Module crt1.o

The supplied `crt1.o` module performs the following actions:

- Checks for the presence of extensions the application uses (specify **-Hnoxcheck** to disable this check)
- Specifies the program entry point (`_start`)
- Initializes general-purpose registers to 0 (zero)
- Initializes the stack pointer to the end of `.stack`
- Initializes the small-data base register
- Calls **__arc_main()** to continue initialization

NOTE If you specify option **-Hpictable** for position-independent code and data, the linker includes startup module `crt1pt.o` instead of `crt1.o`. See Startup Module crt1pt.o for more information.

Program Entry Point

Startup module `crt1.o` provides global symbol `_start`. Usually you configure the reset vector to jump to `_start`, which the linker uses as the application entry point by default. You can specify a different entry point by using linker option **-e** on the linker command line, as follows:

ARCtangent-A4

`ldarc -e` *_entry_point*

ARCtangent-A5 and Later

`ldac -e` *_entry_point*

Replace *_entry_point* with the name of your entry point.

Startup Module crt1pt.o

If you specify option **-Hpictable** for position-independent code and data, the linker includes startup module `crt1pt.o` instead of `crt1.o`.

Startup module `crt1pt.o` performs all the functions of `crt1.o`, but after initializing the general-purpose registers to 0, it calls function **_init_pic** to relocate the code and data addresses in your application to their load locations instead of their link locations.

It is important to note that the `_start` entry point of `crt1pt.o` requires that the base address where the writable data program segment was loaded be passed in register `r0`. The base address of where the executable program segment was loaded is computed internally by **_init_pic**.

Startup Module initpic.o

If you specify option **-Hpictable** for position-independent code and data, the linker includes startup module `initpic.o`.

Startup module `initpic.o` provides function **_init_pic**, which performs the following actions:

- Initializes the stack pointer, adjusting for the fact that the `.stack` section might reside at an address different from the base address assigned by the linker.
- Calls **_initcopy_pic** to initialize any data (or code, or both) to be copied into place using the **INITDATA** feature described in Initializing Memory on page 221.
- Calls **__relocate_from_pictable** to perform the necessary relocations so that position-dependent instructions in your application can work with the addresses used by the loader.

Startup Module arc_main.o

Startup module `arc_main.o` performs the following actions:

- Calls **_initcopy()** to initialize any data (or code, or both) to be copied into place using the **INITDATA** feature described in Initializing Memory on page 221.

> **NOTE** If you specify option **-Hpictable** for position-independent code and data, startup module `initpic.o` instead of `arc_main.o` performs this step.

- Zeroes out the standard uninitialized data sections `.bss` and `.sbss`.

  Module `arc_main.o` performs this step unless you comment it out. If you prefer to use **INITDATA** to perform this task, be sure to comment out the section of **__arc_main()** that zeroes out `.bss` and `.sbss`.

- Calls **__start_kernel()** to initialize the RTOS, if any.

  The call to **__start_kernel()** might or might not return, depending on the nature of the RTOS and its initialization paradigm. If the call does not return, the RTOS should call function **_mwrtl_init()** (in `arc_main.c`) at some point during initialization to complete the required MetaWare C Run-Time Library initialization. If your application does not use an RTOS, this call does nothing.

- Registers function **_fini()** with **atexit()** to ensure that any call to **exit()** also calls global destructors.
- Calls **_init()** to execute global static constructors and other miscellaneous library initialization routines. This call also allocates and initializes any RTOS objects required for mutual exclusion.
- Builds parameters *argc* and *argv* and calls **main()**.
- If **main()** returns, calls **exit()**.

Startup Module crti.o

Startup module `crti.o` performs the essential function of setting up the `.init` and `.fini` ELF sections. Sections `.init` and `.fini` are executable code that is invoked by functions **_init()** and **_fini()** to initialize the library and C++ objects and call global destructors upon program exit. Functions **_init()** and **_fini()** are defined in `crti.o` and do not require target-specific modification.

> **NOTE** If you are not using the MetaWare C/C++ driver, be sure to include `crti.o` when linking.

Providing System Calls for Debugging from a Host

MetaWare C/C++ includes a system-call library (a *hostlink* library) that communicates with the MetaWare debugger to perform system calls on the host platform when debugging. The driver links this library by default.

If you want to supply your own system-call library, for example if you have produced a system-call library for use with a particular RTOS, use driver option **-Hhostlib**=*library* to specify the name of your library.

CAUTION If you use the hostlink library without a debugger, your application might hang. For no library, specify **-Hhostlib=** with nothing after the equals sign (=).

If you are not compiling and linking using the driver, you can remove the hostlink library from the link by eliminating library `libhlt.a`.

When you remove the hostlink library, the linker includes stub routines and a single output buffer instead of a hostlink library.

For related information, see Adapting the Libraries to an RTOS on page 226.

Providing Low-Level System Functions

For embedded systems, you (or your RTOS) might need to implement certain low-level system functions, especially I/O, to support the MetaWare Run-Time Libraries. This section describes the required functions. If the MetaWare Run-Time Libraries reference a function that you have not implemented, the linker includes a stub version instead. Which functions you implement depends on the level of support your application requires.

NOTE By default, the linker includes the hostlink library `libhlt.a`, which provides the low-level system functions for debugging using the MetaWare debugger. Applications linked with the hostlink library do not function without the MetaWare debugger; see also Providing System Calls for Debugging from a Host on page 229.

All functions are permitted to set global state variable `errno` in the event of error. If you are writing a multi-threaded application, see Global State Variable errno and Thread Safety on page 239 in Considerations for Developing Multi-Threaded Applications on page 238.

The functions listed in this section are divided into the following broad categories by functionality:

- File Processing on page 230
- Memory Management on page 234
- Process Management on page 234
- Error Handling on page 235
- Environment Identification on page 235
- Directory Identification on page 236
- Time Calculation on page 236
- Argument Processing on page 236
- Signal Handling on page 237

Many of these functions are documented in the *MetaWare C Library Reference*; the descriptions that follow are not intended to document the inner workings of the functions, but rather to describe the minimum implementation that the MetaWare C Run-Time Libraries expect.

File Processing

The run-time libraries require several functions to provide file I/O. This section describes the required functions.

NOTE If you want to support only **printf()** to `stdout` and no further file I/O functionality, you need to implement only the **_write()** function described in this section.

Opening Files

```
/* For existing files */
extern int _open(const char *name, int flags);

/* For new files */
extern int _open(const char *name, int flags, int mode);
```

Function **_open()** opens file *name* for the operation requested in *flags* (described in detail in the subsection entitled Flags on page 231) and returns a unique file handle. When *flags* contains `O_CREAT`, **_open()** takes three arguments. The third argument, *mode*, is the access mode for the file being created (rarely used in embedded applications).

Function **_open()** returns -1 in case of error.

The file handles must start from 0 (zero) and increment. The supplied run time assumes that your implementation has already opened handles 0, 1, and 2, with handle `0` representing standard input (`stdin` in `stdio.h`), handle `1` representing standard output (`stdout` in `stdio.h`), and handle `2` representing standard error (`stderr` in `stdio.h`). Therefore, you do not need to call **_open()** for `stdin`, `stdout`, and `stderr`.

When the supplied run time opens a file, **_open()** must return the lowest-numbered file handle available. For example, if an application closes file handle `1`, the result of the next call to **_open()** must be `1` (assuming that no application has closed handle `0` in the meantime). If your system does not support file handles or handles numbered consecutively from 0, create a table of handles and treat file handles passed from the supplied run time as indexes into the table.

When **_open()** opens a file, it assumes that the file position is the first byte of the file, except when opening for appending.

When calling this function with three arguments to create a new file, the supplied run time passes octal `0666` for *mode*. You can use or ignore this value, as desired.

Run-Time Library Requirements

The library calls this function for all operations that require opening a file, such as **fopen()**.

Flags

Argument *flags* is a bit mask of flags specifying the operation for opening. Your function should support the following values:

Flag	Value
`O_RDONLY`	0x0
`O_WRONLY`	0x01
`O_RDWR`	0x02
`O_APPEND`	0x08
`O_CREAT`	0x100
`O_TRUNC`	0x200

The MetaWare C Run-Time Libraries use the following combinations (separated by “or”) as opening flags:

O_RDONLY
Open the file for reading only. If the file does not exist, return -1 as an error code.

O_WRONLY|O_CREATE|O_TRUNC
Open the file for writing only. If the file does not exist, create it with a length of 0 (zero). If the file exists, truncate it to a length of 0 (zero).

O_WRONLY|O_CREATE|O_APPEND
Open the file for writing only. If the file does not exist, create it with a length of 0 (zero). If the file exists, preserve the existing contents and begin writing at the end of the file.

O_RDWR
Open the file for reading and writing. If the file does not exist, return -1 as an error code.

O_RDWR|O_CREATE|O_TRUNC
Open the file for reading and writing. If the file does not exist, create it with a length of 0 (zero). If the file exists, truncate it to a length of 0 (zero).

O_RDWR|O_CREATE|O_APPEND
Open the file for reading and writing. If the file does not exist, create it with a length of 0 (zero). If the file exists, preserve the existing contents and begin writing at the end of the file.

Closing Files

`extern int _close(int handle);`

Function **_close()** closes a file. *handle* is the unique file handle returned by **_open()** when it opened the file. After **_close()** has closed a file, **_open()** can re-assign the handle when it opens another file.

If closing is successful, **_close()** returns 0 (zero); in case of error, it returns -1.

Run-Time Library Requirements

The library calls this function when closing a file, such as from function **fclose()**.

Reading Files

`extern int _read(int handle, char *buff, unsigned int count);`

Function **_read()** reads *count* bytes in file *handle* and returns them to buffer *buff*. It advances the file position indicator to reference the byte following the byte it reads.

Function **_read()** returns the number of bytes read; it returns -1 if it encounters an end-of-file character or other error.

Run-Time Library Requirements

The Run-Time Library calls this function from functions that read files, such as **fread()**, **fgets()**, and **scanf()**.

Writing Files

`extern int _write(int handle, const char *buff, unsigned int count);`

Function **_write()** writes *count* bytes from buffer *buff* to file *handle* and advances the current position to reference the location following the byte it writes. Function **_write()** returns the number of bytes written, or -1 in case of error.

Run-Time Library Requirements

All library functions that produce output to a file or `stdout` call **_write()**.

Function **_write()** is the only function required to implement **printf()** to `stdout`.

Seeking to a Location in a File

`extern long _lseek(int handle, long offset, int method);`

Function **_lseek()** moves the file position indicator in file *handle* by adding *offset* to the starting point specified by *method*.

Function **_lseek()** returns the position where the next read or write will occur, relative to the beginning of the file. In case of error, **_lseek()** returns -1.

Calculation Methods

Argument *method* can be one of the following (from `stdio.h`):

Method	Value	Description
SEEK_SET	0	Adds *offset* to the absolute position of the first byte in the file, byte 0 (zero).
SEEK_CUR	1	Adds *offset* to the current file position. An *offset* of 0 (zero) keeps the file position indicator at its current position.
SEEK_END	2	Adds *offset* to the last position in the file. An *offset* of 0 (zero) seeks to the end of the file such that a subsequent write appends to the existing contents of the file.

NOTE Methods *SEEK_CUR* and *SEEK_END* must be able to handle negative values.

Run-Time Library Requirements

The following library functions call **_lseek()**:

fopen()
fseek()
fwrite()
ftell()

Testing File Access

```
extern int _access(const char *name, int mode);
```

Function **_access()** tests whether it is possible to access file *name* in mode *mode*.

Function **_access()** returns 0 (zero) if the file exists and a non-zero value if not or in case of error.

MetaWare C uses only *F_OK* (0) for *mode*. Value *F_OK* tests whether or not the file exists.

Run-Time Library Requirements

The library calls **_access()** to generate temporary file names for several **tmp*** functions and for function **_searchstr()**.

Testing for an Interactive Display Device

```
extern int _isatty(int handle);
```

Function **_isatty()** tests whether file *handle* is an interactive display device.

Function **_isatty()** returns 1 if the file is an interactive display device, and 0 (zero) if not, or in case of error.

Run-Time Library Requirements

The Run-Time Library calls function **_isatty()** to determine whether to use full buffering or line buffering by default when opening a file. For example, **_isatty(*1*)** must return 1 if you want the output of **printf()** to be flushed after every newline by default. (You can explicitly set the buffering mode using function **setvbuf()**.)

Removing Files

```
extern int remove(const char *name);
```

ANSI function **remove()** removes file *name*.

If removal is successful, **remove()** returns 0 (zero); in case of error, it returns -1.

Run-Time Library Requirements

The library calls **remove()** internally to delete files opened using function **tmpfile()**.

Renaming Files

```
extern int rename(const char *old, const char *new);
```

ANSI function **rename()** assigns name *new* to file *old*.

Function **rename()** returns 0 (zero) if renaming is successful and -1 in case of error.

Run-Time Library Requirements

The library does not call this function.

Memory Management

Allocating Memory

```
extern void *_sbrk(int size);
```

Function **_sbrk()** allocates the amount of memory requested in *size*.

If allocation is successful, **_sbrk()** returns a pointer to the block of memory allocated; in case of error, it returns `NULL`.

We recommend that your implementation attempt to return addresses such that memory from any two sequential calls is contiguous. Contiguous allocation of memory is not required, but reduces fragmentation of the memory heap.

Run-Time Library Requirements

The Run-Time Library calls `_sbrk()` to allocate memory.

Freeing Memory

```
extern void *_brk(void *block);
```

Function **_brk()** returns to the system the memory *block* most recently allocated by **_sbrk()**. It returns -1 in case of error.

Run-Time Library Requirements

The library calls **_brk()** to return the block of memory most recently allocated by **_sbrk()**.

NOTE If your implementation of **_sbrk()** always allocates memory in a contiguous sequence, you do not need to implement this function.

Process Management

Determining a Process ID

```
extern int _getpid(void);
```

Function **_getpid()** returns the unique identifier of the current process. It has no error return.

If your system does not contain multiple processes, your implementation of **_getpid()** can return a fixed value.

Run-Time Library Requirements

The following Run-Time Library functions call **_getpid()** (when generating temporary file names):

_mktemp()
_tempnam()
tmpfile()
tmpnam()

Terminating a Process

```
extern int _exit(int status);
```

Function **_exit()** immediately terminates the current process. The calling function passes an implementation-defined *status* that might be returned to the operating system. Function **_exit()** should not return to the caller. The default run-time environment makes no calls after **_exit()**, but simply shuts down the processor. Behavior on different RTOSs might vary.

Run-Time Library Requirements

Function **exit()** calls **_exit()**.

Error Handling

ANSI function **strerror()** requires the following variables:

```
extern int   _sys_nerr;
extern char *_sys_errlist[];
```

These variables are located in library source file *install_dir*/`lib/src/c/misc_g/_sys_err.c`. You can modify or replace this file for your hostlink implementation if required. If your target implementation provides system calls that mirror the host system calls, we recommend using the host numbering of error codes. Then your target can simply return the error codes computed by the host.

For example, **read()** executed on the target then calls **read()** on the host and returns the host error code in case of error.

Run-Time Library Requirements

The library uses these variables to translate `errno` values into error messages for function **strerror()**, which is itself used by **perror()**.

Environment Identification

Getting Environment Variables

```
extern char *getenv(const char *var);
```

ANSI function **getenv()** returns the address of the string associated with variable *var*, or `NULL` if *var* is not defined.

You can implement function **getenv()** as a call to **getenv()** on the host, or hard-code it to have a fixed environment or always return `NULL`.

Run-Time Library Requirements

The library does not call **getenv()** internally.

```
extern char *_mwgetenv2(const char *var);
```

Function **_mwgetenv2()** returns the address of the string associated with variable *var*. It is identical to **getenv()**, except that it uses a different internal buffer to return the address of the string.

Function **_mwgetenv2()** returns `NULL` in case of error.

The run-time library calls **_mwgetenv2()** to support the following:

- the `MALLOC_LEVEL` feature of the heap allocator
- function **_searchenv()**
- function **_tempnam()**
- integrated profiling

Directory Identification

Determining the Current Directory

```
extern char *_getcwd(char *buf, int len);
```

Function **_getcwd()** places the name of the current working directory in a buffer with address *buf* and maximum length *len*.

Function **_getcwd()** returns the address of the buffer, or `NULL` in case of error.

Run-Time Library Requirements

Function **_searchstr()** calls **_getcwd()**.

Time Calculation

Determining the Clock Value

```
extern clock_t clock(void);
```

ANSI function **clock()** returns the number of clock ticks elapsed since the beginning of an implementation-specific starting point. Use the return value only for comparisons. Function **clock()** has no required error return; we recommend returning `NULL` in case of error.

You can approximate time in seconds by dividing the difference of two calls to **clock()** by **CLOCKS_PER_SEC**.

Your implementation must either count clock ticks to match the setting of **CLOCKS_PER_SEC** (in `time.h`) or change the value of **CLOCKS_PER_SEC** to match the frequency of clock ticks.

Run-Time Library Requirements

The library does not call this function.

Determining the Time Value

```
extern time_t time(time_t *__timer);
```

ANSI function **time()** returns the current time on the system, or `NULL` in case of error.

Run-Time Library Requirements

The library calls this function to implement other functions from `time.h`.

Argument Processing

Counting Arguments

```
extern int __argc(void);
```

Function **__argc()** counts the number of arguments entered by the user when invoking the program and returns the result, or `NULL` in case of error.

The program name is the first argument (argument 0).

Run-Time Library Requirements

The supplied startup initialization code calls **__argc()** when invoking **main()**.

Finding the Value of an Argument

```
extern char *__argv(int num);
```

Function **__argv()** returns the a pointer to the value of program argument *num*, or `NULL` in case of error.

Number the arguments from 0 (zero) to `(__argc() - 1)`. Argument 0 is the program name.

Run-Time Library Requirements

The supplied startup initialization code calls **__argv()** when invoking **main()**.

Signal Handling

The MetaWare C Run-Time Libraries provide basic functions for registering and invoking a signal. You might want to modify or replace them for more sophisticated signal handling on your target.

Registering a Signal

```
extern void (*signal(int sig, void (*func)(int)))(int);
```

ANSI function **signal()** registers function *func* to handle signal *sig*.

If successful, **signal()** returns a pointer to the previous handler for signal *sig*. In case of error, **signal()** returns **SIG_ERR**.

See the *MetaWare C Library Reference* for a more detailed description and cautions on using **signal()**.

Run-Time Library Requirements

The library does not call this function.

Invoking a Signal

```
extern int raise(int sig);
```

ANSI function **raise()** sends signal *sig* to the current program.

Function **raise()** returns 0 (zero) if successful and a non-zero value if not successful.

Run-Time Library Requirements

The library calls **raise()** from function **abort()**.

Building Thread-Safe Applications

The MetaWare C/C++ toolset supports thread safety (driver option **-Hthread**) for applications that are designed to be thread-safe. This section discusses issues in using the toolset to build thread-safe applications.

> **CAUTION** It is not safe to call thread-safe MetaWare C/C++ run-time functions from interrupt handlers. The mutex semaphores that protect the run-time functions do not provide thread safety against interrupt handlers. Instead, queue the event and post a condition variable for a thread to handle, or limit your use of run-time functions to those listed as reentrant in the ***MetaWare C Library Reference***.

Thread Safety and Reentrancy Contrasted

A *thread-safe* function is one that can be safely called by multiple threads of a multi-threaded operating system.

A *reentrant* function is one that can be safely entered more than once by the same thread because it does not modify static storage. Reentrant functions are inherently thread-safe (and can be called from interrupt handlers). Refer to the *MetaWare C Library Reference* for the reentrancy status of individual functions.

Considerations for Developing Multi-Threaded Applications

In addition to driver option **-Hthread**, the MetaWare C/C++ toolset supports the storage modifier **__declspec(thread)**, described in Thread-Local Storage and the thread Attribute on page 254.

The default MetaWare C Run-Time Libraries are compiled for thread safety. For more information, see Linking in the Thread-Safe Library on page 32.

With threads come the possibility of race conditions and conflicts over resources and services that must be serialized for proper program behavior. This is true for the MetaWare C/C++ Run-Time Libraries as well as for application code. Multiple threads might call a function like **malloc()** at the same time and the run-time library must serialize updates to the global state variables that represent which memory is available and which is allocated.

For embedded targets, true thread safety and thread-local data require tailored library support for the specific RTOS used. This support can be provided by implementing a small set of functions. For supported RTOSs, these functions are implemented automatically if you specify driver option **-Heos=*rtos_name*,** where *rtos_name* is the name of a supported RTOS. If you are using an unsupported RTOS, you must implement the library support functions yourself.

For more information, see Porting an Interface for RTOS Thread Safety on page 238.

For a list of supported RTOSs, see option **-Heos**.

Porting an Interface for RTOS Thread Safety

The MetaWare C/C++ toolset includes a simple interface that you can port to your RTOS. After you port the interface to the RTOS and compile it, it supports thread safety for that RTOS.

The function declarations for the interface are located in header file `thread.h` in your MetaWare C/C++ distribution. The interface is defined as follows:

```
extern int *___errno(void);
extern void *_Preserve_flags _mwget_tls(void);

typedef void *_lock_t;
extern void _mwmutex_create(_lock_t*);
extern void _mwmutex_delete(_lock_t*);
extern void _mwmutex_lock(_lock_t);
extern void _mwmutex_unlock(_lock_t);
```

The stub (or dummy) implementations of these interfaces are in the following three files, located in *install_dir*`/lib/src/mw/misc_g` on your installation:

- `errnog.c`
- `tlsctorg.c`
- `mutexg.c`

Each file has comments documenting its implementation of the interfaces.

If you do not link an RTOS interface library or provide your own library support functions, MetaWare C/C++ links in the stub versions, which do not provide thread safety.

The implementations supplied automatically when you link with option **-Heos** and specify the name of a supported RTOS can serve as models for the functions you must write. You can view their source in the directories below *install_dir*`/lib/src`.

Thread-Local Storage

You can provide each thread in your program with a thread-specific instance of a global variable or objects by applying to the variable or object the specification **__declspec(thread)**, described Thread-Local Storage and the thread Attribute on page 254.

The MetaWare C++ Run-Time Library uses thread-local storage for exception handling. As you enter and exit C++ **try** blocks, the compiler instruments your code with calls to the run-time library to maintain a linked list. The linked list must be thread-specific for your program to behave properly.

The MetaWare C/C++ Run-Time Libraries also use thread-local storage when they need global state variables. Each global variable was independently declared using **__declspec(thread)**.

Global State Variable errno and Thread Safety

In many applications, the global state variable `errno` is the only variable that needs to be thread-local.

Undesired setting of `errno` is most likely to occur if you use **malloc()** to allocate memory, because **malloc()** attempts to set `errno` if it cannot allocate memory. To avoid **malloc()** setting `errno`, allocate memory using a low-level memory allocator such as **sbrk()** or an RTOS memory-allocation function.

The MetaWare C/C++ global state variable `errno` is actually a C preprocessor macro that calls **___errno()**. The fact that `errno` calls **___errno()** makes it possible to adapt the use of `errno` in different ways to provide multi-threaded library support. For example, if the RTOS provides its own thread-safe `errno`, your implementation of **___errno()** should return the address of the `errno` provided by the RTOS. If the RTOS does not provide `errno`, then your implementation of **___errno()** should provide `errno` thread-locally. You can provide `errno` thread-locally using **__declspec(thread)** as long as your implementation of **_mwget_tls()** does not need to set `errno` in any of the calls it makes to set up the thread-local area.

Because some implementations of **__declspec(thread)** use **malloc()**, we recommend implementing `errno` in a way that does not require **malloc()**. That is, implement **___errno()** so that it returns the address of a field in the thread control block, as shown in the following example.

Example 46 A thread-safe Implementation of errno

```
#undef errno
int errno;
int *___errno(){
   RTOS_THREAD *self = rtos_current_thread();
   if (self != NULL)
      return &self->errno_field_in_thread_control_block;
   else
      return &errno;
      }
```

Building a Thread-Safe Interface

After you configure `errnog.c`, `tlsctorg.c`, and `mutexg.c` for your RTOS, you must compile them and link with them. You can compile and link with them directly as you would your own application source, or you can build your own RTOS interface library and include your library in the link. Place the library between the C/C++ library and `libmw`. To do this when linking with the driver, add your library to the argument for driver option **-Heoslib**.

If you adhere to the following guidelines, you should be able to implement thread-safe versions of the functions in `errnog.c`, `tlsctorg.c`, and `mutexg.c` and compile them using driver option **-Hthread** (the default):

- In code that allocates a semaphore, do not call any function that would allocate a semaphore.
- When allocating thread-local storage, do not reference thread-local variables or call any functions that reference thread-local variables
- Avoid calling **malloc()** or any function that might reference `errno` unless `errno` is implemented in a thread-safe fashion as described in Global State Variable errno and Thread Safety on page 239.

MetaWare C/C++ includes makefiles in the library source tree. You can rebuild the run-time libraries there using **gmake**. For more information, see Rebuilding the C Run-Time Libraries on page 241.

Avoiding Library Usage of Semaphores and Thread-Local Storage

It is sometimes beneficial, even necessary, to avoid use of semaphores and thread-local storage. For example, you might want to avoid use of semaphores and thread-local storage during startup, until the RTOS is ready to service memory-allocation and semaphore requests.

CAUTION Never call a routine that uses semaphores or thread-local storage from an interrupt handler. The mutex semaphores that protect the MetaWare C/C++ run-time functions do not provide thread safety against interrupt handlers. Instead, queue the event and post a condition variable for a thread to handle, or limit your use of run-time functions to those listed as reentrant in the *MetaWare C Library Reference*.

You can avoid using semaphores and thread-local storage by avoiding use of run-time library services listed in the following sections:

- Thread-Local Storage in the Run-Time Libraries on page 240
- Semaphores in the Run-Time Libraries on page 240

Thread-Local Storage in the Run-Time Libraries

The services provided by the following include files use thread-local storage when you compile with option **-Hthread**. Avoid these services if you do not want to use thread-local storage in your startup code (see also Semaphores in the Run-Time Libraries on page 240).

`stdio.h`	`time.h`
`locale.h`	`math.h`
`float.h`	`signal.h`
`errno.h`	`malloc.h`

Each of these library services has or references a global state variable that is thread-specific; the most commonly referenced variable is `errno`.

When a thread-local global state variable is referenced, function **_mwget_tls()** is called. The implementation of **_mwget_tls()** varies by RTOS.

Semaphores in the Run-Time Libraries

The following Run-Time Library services use semaphores:

- File I/O
- Functions **exit()** and **atexit()**
- Functions in `malloc.h`, including **free()**

- C++ operators **new()** and **delete()**
- C++ constructors, destructors, and exceptions

Rebuilding the C Run-Time Libraries

MetaWare C/C++ includes the complete source of the MetaWare Run-Time Libraries, and makefiles for building each library. Prebuilt libraries are supplied for the most popular ARC configurations. Users typically rebuild the libraries to change the optimization level, to add debug information with **-g**, or to enable use of extensions that are not included in option **-Xlib**.

Other Possible Reasons

- Your application is not multi-threaded, and you want to eliminate the slight overhead incurred by the thread-safe libraries by recompiling using driver option **-Hnothread**.
- You have modified the startup code or other library modules.

Rebuilding the Libraries

The libraries include makefiles in the directories by processor series below *install_dir*`/lib`. Use **gmake** and the makefile to compile all the libraries for your processor series.

In addition, each library directory further down the hierarchy contains a makefile for rebuilding that specific library set (little-endian, extensions, etc.). Use **gmake** to rebuild the library set for your configuration.

The command shown in Example 47 Rebuilding the ARCtangent-A5 Little-Endian Extension Libraries rebuilds the C, C++, hostlink and `libmw` libraries and deposits the new libraries in the current directory (overwriting the existing libraries).

Example 47 Rebuilding the ARCtangent-A5 Little-Endian Extension Libraries

```
cd install_dir/lib/a5/le/xlib
gmake
```

At any of the levels, you can modify the makefiles or use command-line specifications to alter the selection of what compiler is used and what compiler options are used. For example, to build the entire set of libraries with a specific compiler and additional options:

```
cd arc/lib
gmake CC=/path/to/my/specific/toolkit/bin/mcc cflags="my add'l options"
```

Creating Your Own Library Directories

You can copy the libraries to a new directory structure and rebuild them using the existing makefiles as a guide. If you copied the files to a new directory, use driver option **-Hrtldir**=*dir* to link the libraries from the new directory. Unless you specify a full path, option **-Hrtldir** assumes *dir* is under *install_dir*`/lib`. For example, the following command line tells the driver to look for the new libraries in *le_extensions* directly below the existing *install_dir*`/lib` directory:

```
mcc -Hrtldir=le_extensions
```

Using option **-Hrtldir**, you can easily select from multiple configurations of the run-time libraries.

If compiling and linking in two steps (i.e., without the driver), or if you wish to specify the libraries by name, see the *ELF Linker and Utilities User's Guide*.

Providing Support for Statistical Profiling

The MetaWare C/C++ toolset offers several options and features to support execution profiling, described in detail in Using Statistical Profiling on page 293.

Your distribution includes full implementations of the MetaWare C Run-Time Library to support the statistical-profiling feature for popular targets. Because this instrumentation code is specific to a target and RTOS, you might need to perform some configuration work.

If your target is not supported, the default run-time library will collect only procedure-call counts. All timing-related information requires an implementation of an interval timer.

Timer Extension Support

The MetaWare C/C++ toolset provides an interrupting timer module for the enhanced system timer extensions. Code using this timer module performs correctly only with the timer extension. To specify execution profiling with the timer extension, link your program with both option **-p** or **-pg** and option **-Xtimer0** or **-Xtimer1** (if you specify both modules, the one specified last is included in the link). If you do not specify a timer, the linker issues a warning.

NOTE We recommend you not use option **-Xtimer0** or **-Xtimer1** for profiling on the debugger's simulator; the instruction counts used by **-Xtimer0** and **-Xtimer1** are a statistical approximation only.

For more accurate simulator profiling, we recommend you use the MetaWare debugger profiling feature, which uses the simulator's instruction counts. For more information on profiling using the MetaWare debugger, see the *MetaWare Debugger User's Guide*.

The source to the timer module is *install_dir*`/lib/src/c/misc_g/angeltim.s`. It contains several constants that you can select to change the default interrupt frequency. If you are using the MetaWare debugger, you can change the frequency from the debugger. For more information about profiling display in the debugger, see the *MetaWare Debugger User's Guide*.

If you specify a supported RTOS using option **-Heos**, the RTOS rather than the supplied `angeltim.s` initializes the timers. Do not link using option **-Xtimer0** or **-Xtimer1** unless the RTOS uses one timer and you wish to profile independently of the RTOS using the other.

Providing Instrumentation Code

The profiling instrumentation code for the MetaWare C Run-Time Library is provided in *install_dir*`/lib/src/c/misc_g`. This directory contains several C source files that implement the UNIX-like **monitor()** interface (the **monitor()** function is documented by AT&T).

To port the execution profiler to a particular target board or OS, you must provide the following functions (see *install_dir*`/inc/mon.h`):

```
extern void _mwstart_profile_clock(void);
extern void _mwstop_profile_clock(void);
extern unsigned _mwprofile_clock_freq(void);
extern unsigned _mwprof_countdown_ticks;
extern unsigned _mwprof_countdown_freq;
```

Function _mwstart_profile_clock()

When your program begins to execute, it makes an initial call to **monitor()**. This first call to function **monitor()** results in a call to function **_mwstart_profile_clock()** which, in turn, starts an interval timer that produces an asynchronous interrupt.

At every timer interrupt, the following function must be called from the interrupt handler to record the current execution point within the application:

```
_Save_all_regs extern void _mwprofile_tick(void *pc);
```

The input *pc* is the value of the program counter when the interrupt occurred.

Function **_mwprofile_tick()** saves all registers (both volatile and non-volatile) that it modifies in the processor. This means that the interrupt handler, if written in assembly, does not have to save and restore any registers other than those its own assembly portion uses.

Function _mwstop_profile_clock()

When your program execution is complete, it makes a second call to **monitor()**. This second call to function **monitor()** results in a call to function **_mwstop_profile_clock()**, which disables the interval timer.

> **NOTE** The file *install_dir*`/lib/src/c/misc_g/profclk.c` contains stub routines and documentation for the **_mwstart_profile_clock()** and **_mwstop_profile_clock()** functions.

Function _mwprofile_clock_freq()

Function **_mwprofile_clock_freq()** returns the frequency of the timer interrupts, in units of "interrupts per second." For example, if the timer interrupt is once every 100 microseconds, the function returns `10000`.

The value that this function returns should be equivalent to `_mwprof_countdown_freq` divided by `_mwprof_countdown_ticks`.

> **NOTE** Although faster interrupt frequencies generate more run-time overhead, they also more accurately determine where the program is spending its time.

Variable _mwprof_countdown_ticks

Variable `_mwprof_countdown_ticks` contains the number of clock ticks between interrupts. We recommend you write your profiling timer interrupt handler in such a way that you are able to dynamically alter the interrupt frequency by referencing the value of this variable when reloading the timer's counter register. This allows you to change the interrupt frequency directly from the MetaWare debugger (for details, see the *MetaWare Debugger User's Guide*).

If you do not want to change the interrupt frequency directly from the MetaWare debugger, you do not need to define this variable.

Variable _mwprof_countdown_freq

Variable `_mwprof_countdown_freq` reflects the frequency of the timer, in ticks per second. Thus, if the timer is running at 3.125 MHz, the value would be 3125000. The profiler uses this variable to generate accurate timing in the profiling report you can view when using the MetaWare debugger (for details, see the *MetaWare Debugger User's Guide*).

If you do not want to view profiling reports in the MetaWare debugger, you do not need to define this variable

Using Closely Coupled Memory

If you have configured your processor to have separate instruction and data spaces (known as closely coupled memory or a Harvard architecture), do the following:

1. Build with option **-Hccm** to enable support for closely coupled memory and link in the appropriate libraries.
2. Create a special linker command file to ensure that all data sections are linked in the data space and only executable instructions are linked in the instruction space. See the *ELF Linker and Utilities User's Guide* for an example command file.
3. Notify the debugger simulator of the addresses of instruction and data memory. For more information, see the *MetaWare Debugger User's Guide*.

Using Position-Independent Code and Data

To load an executable at an arbitrary address other than the one where it was linked requires that pre-initialized data storage be properly relocated. Specifically, each data word (whether part of an instruction or data) that is initialized to the address of a function or (static) variable must be adjusted. To meet this requirement, ARC® International provides the pictable mechanism, which you can use by specifying option **-Hpictable**.

> **TIP** An alternative approach is to use toggle `Switch_pic`, which causes the compiler to generate position-independent `switch` statements. Using toggle `Switch_pic` might provide sufficient position-independence of code (but not data), for your application.

The pictable mechanism is useful, for example, when code resides in ROM or flash but must be copied to read-write memory for execution.

The pictable mechanism links programs so that they can dynamically relocate their code group, their data group, or both at run time using a dynamic relocation table (a *pictable*) instead of a dynamic loader.

In the pictable mechanism, the linker builds the pictable into the program image (in read-only data) with all of the information needed to relocate the data and start running. You can burn the program into ROM or flash memory and move it into RAM.

A limitation of the pictable mechanism is that code must be writable so that it can be fixed up at run time. If you wish to execute directly out of ROM or flash, the pictable mechanism is normally not necessary.

Building Position-Independent Code and Data

To build position-independent code and data, specify driver option **-Hpictable**; if you are invoking the linker without the driver, use linker option **-Bpictable** (documented in the *ELF Linker and Utilities User's Guide*).

Driver option **-Hpictable** passes linker option **-Bpictable** to instruct the linker to do the following:

- create a table for relocating pre-initialized data
- include special startup modules for position-independent code and data

The linker creates the dynamic relocation table in a section named `.pictable` and defines symbol `__PICTABLE__` as the base of the section.

The pictable mechanism requires that the generated executable have no more than two segments: a code segment and a data segment. By default, the linker generates these two segments, and places executable code and read-only literals in the code segment and everything else in the data segment.

If you provide your own linker command file with more than two segments when using **-Bpictable**, the linker issues an error.

Furthermore, the code and data groups must each remain contiguous after linking. You cannot load part of the code to one area of memory and the rest elsewhere, and you cannot load part of the data to one address and the rest elsewhere. You can use a linker command file to move an entire group, but you must treat it as a single contiguous entity and copy it in its entirety to the new address range.

Pictable Processing and Startup Files

When you use option **-Hpictable**, the linker includes an alternative startup module named `crt1pt.o`. The `_start` entry point of `crt1pt.o` requires one argument to be passed in register `r0`: the base address where the data segment was actually loaded (as opposed to the address assigned by the linker).

Startup file `crt1pt.o` calls function **_init_pic** to perform the necessary relocations. Upon return, all references are appropriately adjusted.

For more information, see Working with Startup Modules on page 227.

Pictable Processing and Memory Initialization

You can use the INITDATA feature of the ARC® International ELF linker (see the *ELF Linker and Utilities User's Guide* for details) to initialize portions of the data and code segments, even if the data and code segments are loaded at an address other than the one where they were linked. The startup code invokes function **_initcopy_pic** before pictable processing to initialize the segments.

After INITDATA initializes the data, the pictable mechanism relocates the segments appropriately.

For more information, see Working with Startup Modules on page 227.

Chapter 11 — Data Storage

In This Chapter

- Overview
- Implementation-Defined Behavior
- Compiler Limits
- Storage Mapping for Class Objects

Overview

This chapter describes how the MetaWare C/C++ compiler maps C++ objects to storage and provides other implementation-defined information as required by the ANSI C Standard.

Implementation-Defined Behavior

The ANSI C Standard (ANSI document X3.159) requires that each C compiler implementation provide information about how various system specifics behave. The following sections document MetaWare C implementation-defined behavior.

Identifiers

Number of Characters in an Identifier

The compiler places no limit on the number of characters in an identifier, with or without external linkage.

Identifier Case-Sensitivity

External identifiers are case-sensitive. The implementation used to map sequences of characters to external source file names is case-sensitive.

Number of Identifiers

The compiler places no limit on the number of external identifiers in one translation unit, the number of identifiers with block scope declared within one block, or the number of macro identifiers simultaneously defined in one translation unit.

Characters

Source and Execution Character Sets, Members and Mapping

Source and execution character sets are both standard ASCII. Each character in the source character set maps into the identical character in the execution character set. Without exception, all character constants map into some value in the execution character set. A character is stored in a byte (eight bits).

char Type Compatibility

The type specifier **char**, when not accompanied by an adjective, denotes by default an unsigned character type.

You can change the default type of **char** by turning off the toggle `Char_default_unsigned`. However, even though you make the **char** type be signed or unsigned, **char** still is not type-compatible with either **signed char** or **unsigned char**.

To make **char** type-compatible with whichever type is the default representation, turn on the toggle `Char_is_rep`. This toggle causes the **char** type to be identically either **signed char** or **unsigned char**, depending on the default.

Character Constants

In C, a character constant is of type **int**. The ANSI C Standard requires the type to remain **int**, even if truncation occurs. MetaWare C/C++ permits a character constant that contains more than one character.

This is the value of a two-character constant:

(256 * (*value of the first character*)) + (*value of the second character*)

For example, the value of `'ab'` is (256 * `'a'`) + `'b'`. In general, when 0 < *N* < 5, the value of an *N*-character constant is recursively defined as:

(256 * (*value of the first N-1 characters*)) + (*value of the Nth character*)

The value of a wide-character constant that contains *N* multi-byte characters is the value of the last (*N*th) multi-byte character. For example, the value of `L'abcd'`, `L'bcd'`, and `L'cd'` is the value of `L'd'`.

Integers

Integer Representations

Integers are represented in twos-complement binary form; there are four bytes in an **int**. Various integer types are restricted to certain ranges of values, as shown in Size, Alignment, and Range of Integer Data Types on page 249.

Table 10 Size, Alignment, and Range of Integer Data Types

Integer Data Type	Size (Bytes)	Alignment (Bytes)	Minimum Value	Maximum Value
signed char	1	1	-128	127
unsigned char	1	1	0	255
short	2	2	-32,768	32,767
unsigned short	2	2	0	65,535
int	4	4	-2,147,483,648	2,147,483,647
unsigned int	4	4	0	4,294,967,295
long	4	4	-2,147,483,648	2,147,483,647
unsigned long	4	4	0	4,294,967,295
long long	8	4	-2^{63}	$2^{63}-1$
unsigned long long	8	4	0	$2^{64}-1$

Integer Conversions

These are the rules for conversions in which the result cannot be represented exactly:

- **int** to **float**: Convert to the nearest value, higher or lower
- **float** to **int**: Round toward 0 (zero)

An integer is converted to a shorter signed integer or **int** bit field by bit truncation. That is, when an *X*-bit value *Xval* is to be stored into a *Y*-bit receptacle *Yval*, where *X* > *Y*, the rightmost *Y* bits of *Xval* are stored in *Yval*.

An unsigned integer *U* is converted to a signed integer *I* where **sizeof(*U*) = sizeof(*I*)** by transferring the bits of *U* into *I*, whether or not the value of *U* is representable in *I*.

> **NOTE** If the value of an unsigned integer is larger than the largest positive signed integer value, it is interpreted bit by bit and yields a negative value.

Operations Involving Signed Integers

In an integer divide operation (*X*/*Y*), when either of the operands is negative, the quotient is truncated toward 0 (zero).

In an integer remainder operation (*X***%***Y*), when either or both of the operands is negative, *X* **%** *Y* is the same as *X* - (*X*/*Y*) * *Y*.

The result of a bitwise operation on a signed integer is the same as if the integer were treated as unsigned.

The sign of the remainder on integer division is the same as the sign of the dividend.

The right-shift of a signed integral type is arithmetic; that is, the sign bit is propagated to the right.

Floating-Point Numbers

Floating-Point Representations

The MetaWare C/C++ compiler uses the IEEE Standard 754 formats to represent floating-point data. Each floating-point data type has a sign bit, exponent bits, and mantissa bits. Size and Alignment of Floating-point Data Types on page 250 and Minimum and Maximum Absolute Values of Floating-Point Integers on page 250 summarize this information.

Table 11 Size and Alignment of Floating-point Data Types

Floating-point data type	Size (bytes)	Alignment (bytes)	Sign (bit)	Exponent (bits)	Mantissa (bits)
float	4	4	1	8	23
double	8	4	1	11	52
long double	8	4	1	11	52

Table 12 Minimum and Maximum Absolute Values of Floating-Point Integers

Floating-Point Data Type	Minimum Absolute Value	Maximum Absolute Value
float	1.1755×10^{-38}	$3.4028 \times 10^{+38}$
double	2.2251×10^{-308}	$1.7977 \times 10^{+308}$
long double	2.2251×10^{-308}	$1.7977 \times 10^{+308}$

For details about the IEEE data format, refer to the ANSI/IEEE Standard 754/1985 documents.

Floating-Point Conversions

These are the rules for conversions in which the result cannot be represented exactly:

- **float**-to-**int** conversions

  When a floating-point number is truncated to an integral type, the truncation is toward 0 (zero). For example:

  — –2.7 is truncated to –2

  — –1.2 is truncated to –1

  — 2.7 is truncated to 2

- **int**-to-**float** conversions

  When an integral type is converted to a floating-point number, the truncation is toward the nearest value, higher or lower.

- **double**-to-**float** conversions

  When a double-precision value is converted to a floating-point number, the compiler always rounds to the nearest value, higher or lower.

Arrays and Pointers

Arrays

Arrays are stored in *row-major* order. This means that the last subscript in a multi-dimensional array varies fastest.

For example, this array:

```
int x[4][6]
```

is stored as this:

```
x[0][0]
x[0][1]
x[0][2]
...
x[3][3]
x[3][4]
x[3][5]
```

Type of size_t

The type of `size_t`, the value returned by **sizeof()** (that is, the type of integer required to hold the maximum size of an array), is type **int** if representable as an **int**. Otherwise, it is type **unsigned**.

Casting a Pointer to an Integer and Vice Versa

When toggle `Pointers_compatible_with_ints` is on, the compiler allows pointers of any type to be compatible with **int** types, with an appropriate warning.

When toggle `Pointers_compatible_with_ints` is off, the compiler disallows this practice.

See toggle `Pointers_compatible_with_ints` in Alphabetical List of Compiler Toggles on page 86 for more information.

An **int** cast to a pointer type is always system-dependent because the size of a pointer on a given system may be different from the size of an **int**.

ARC pointers are large enough to hold all the bits of an **int**, and the conversion copies all the bits of the **int** straight over to the pointer without modification.

Difference of Two Pointers

The difference of two pointers is of type **ptrdiff_t**, which is defined as type **int** in `stddef.h`.

Placing Objects in Registers

An **auto**- or **register**-class variable can spend a portion or all of its lifetime in a machine register or even in multiple registers. Multiple variables can be mapped to a single register.

The compiler attempts to map each **auto**- or **register**-class variable to a register if its type is appropriate and its address is not taken.

auto variables that are not mapped into registers are placed at fixed offsets within the function's stack frame. This is necessary if the variable's address is required, or if the variable is of an aggregate type (for example, **struct** or array).

A **register** variable is treated exactly as an **auto** variable as far as code generation is concerned, except when a function calls **setjmp()**. In such a case, all **auto** variables are forced to memory so that their values are not lost across a subsequent **longjmp()** call.

How the Compiler Maps Variables to Storage

The compiler defines several default control sections for placing code and data. Table 13 Default Control Sections for Code and Data" summarizes these default sections.

NOTE We recommend treating these section names as reserved and not using them as names for your own sections.

Table 13 Default Control Sections for Code and Data

Name	Contents
`.arcextmap`	Debug information relating to processor extensions
`.bss`	Uninitialized variables that are not **const**-qualified (startup code normally sets `.bss` to all zeroes)
`.data`	Static variables (local and global) and writable strings
`.data1`	Initialized variables larger than 8 bytes
`.fini`	C++ destructor code
`.fixtable`	Function replacement prologs
`.got`	Global offset table
`.heap`	Uninitialized memory used for the heap
`.init`	C++ constructor and library-initialization code
`.initdata`	Holds initialized variables and code (usually compressed) to be copied into place during run-time startup.
`.pictable`	Table for relocating pre-initialized data when generating position-independent code and data
`.plt`	Procedure-linkage table
`.rodata`	Read-only string constants and read-only static variables (that is, those that are **const**-qualified)
`.sbss`	Uninitialized small data, set to all zeroes by startup code
`.sdata`	Initialized small data
`.stack`	Stack information
`.text`	Executable code
`.tls`	Thread-local data
`.ucdata`	Data accessed using cache bypass

Special features might create additional sections; for overlay-related sections, see the *Automated Overlay Manager User's Guide*.

Behavior on Pragma Use of Default Names

We recommend treating the default section names as reserved and not using them as names for your own sections.

If you use a pragma to map data to a section with one of the default names listed in Table 13 Default Control Sections for Code and Data on page 252, your data are mapped in alongside the data that would "naturally" be mapped there. The compiler issues a warning and appends '$*N*' to the section name in assembly and object files, where *N* is the order in which the compiler uses the section. In the final executable, the linker combines the files into a section of the original name. For example:

```
#pragma bss(".bss")
Int x; // Appears in .bss$N in .s and .o files.
#pragma bss()
```

Local Static Variables

Each local **static** variable is placed in one of the following sections:

- `.rodata`, if **const** qualified
- `.data`, if initialized and 8 bytes or less
- .data1, if initialized and larger than 8 bytes
- `.bss`, if uninitialized and 8 bytes or larger
- `.ucdata`, if declared **_Uncached** or if declared **volatile** and compiled with toggle `Volatile_cache_bypass` turned on (the default).

NOTE If pragmas controlling the use of small-data sections are in effect, local **static** variables might be placed in `.sdata` or `.sbss`, depending on the nature of the variables and the parameters passed to pragma `Push_small_data`. For information on controlling the use of small-data sections, see Placing Data in Small-Data Sections on page 214.

Global Variables

Each uninitialized global variable with no **extern** specifier is defined as a common block or placed in the `.bss` or `.sbss` section, depending on the setting of toggle `Multiple_var_defs`.

Each initialized global variable with no **extern** specifier is mapped into the `.data` or `.data1` section and given the `global` attribute.

Auto Variables

Each **auto** variable is assigned either to a machine register or to storage in the routine's stack frame. The compiler attempts to place each **auto**-classed variable in a register, provided the variable's type is appropriate and its address is not required. If **setjmp()** is called within a function, all **auto** variables are mapped to memory unless the **register** class is explicitly specified, so that the semantics of **setjmp()** and **longjmp()** work correctly.

For more about stack frames, see Interrupt Stack Frames on page 279.

const-Qualified Variables

Each **const**-qualified static initialized variable, whether local or global, is mapped to the read-only data section. For example:

```
main () {
   static const char hextable[] = "0123456789abcdef";
   int i;
   ...
   f(hextable[i]);
   }
```

Any attempt to directly assign to `hextable` elicits an error message from the compiler. Any indirect attempt, such as through passing a pointer to `A` or to `hextable`, will most likely cause a protection fault or other error at execution time.

Note that string literals are placed in a read-only section only if they are part of a **const** data object. For example:

```
const char * P = "a string";
```

Here `"a string"` is placed in the read-only section, but `P` is placed in data. To place `P` in the read-only section, do this:

```
const char * const P = "a string";
```

extern Variables

Each **extern** variable is given the `global` and `undefined` attributes.

Thread-Local Storage and the thread Attribute

To provide thread-local instances of a global variable or object, use the extended-storage class attribute **thread** with keyword **__declspec**. Keyword **__declspec** can appear anywhere a storage-class keyword (such as **static**) can appear. The **thread** storage-class attribute does not specify the type of a declared item; it simply affects storage allocation.

The following example declares and initializes an integer thread-local variable:

```
__declspec(thread) int tls_i = 0;
```

To make your code more understandable, use a macro definition to create an alias for the **__declspec(thread)** command:

```
#define Thread __declspec(thread)
. . .
Thread int tls_i = 0;
```

NOTE Most implementations of **__declspec(thread)** use **malloc()** and **free()**, which may pull more C run-time code into your program than you intend.

You must observe the following restrictions when declaring statically bound thread-local variables and objects:

- Apply the **thread** attribute only to data declarations and definitions. You cannot use it on function declarations or definitions.
- Apply the **thread** attribute only to data items with static storage duration:
  - — global data objects (both **static** and **extern**)
  - — local static objects
  - — static data members of C++ classes
- You must both declare and define a thread-local object with the **thread** attribute, whether the declaration and definition occur in the same file or in separate files.

NOTE You cannot apply the **thread** attribute to automatic data objects.

Thread-local variables are generally useful only when the application is running under an operating system or RTOS that supports multiple threads of execution. In such cases, run-time library support is required to implement TLS. When you use driver option **-Heos** to specify an RTOS supported by the compiler, the necessary run-time support is provided automatically. For more information, see Chapter 4 — Using Driver Options.

For information on providing support for multiple threads for RTOSs not supported by the compiler, see Building Thread-Safe Applications on page 237.

For information about a compiler feature that automatically applies the **thread** storage attribute to all data items within a given scope, see toggle `Thread` in Alphabetical List of Compiler Toggles on page 86.

Structures, Unions, Enumerations, and Bit Fields

Size, Padding, and Alignment of Structures, Classes, and Unions

The size of a **struct**, **class**, or **union** depends on whether the compiler generates padding to align the construct's members.

The compiler generates such padding by default under the control of the `Align_members` pragma. For more information, see Aligning Data Structures on page 150.

The keywords **_Packed** and **_Unpacked** have been added to MetaWare C/C++ to allow control over member alignment on an individual **struct**, **class**, or **union** basis. A **_Packed struct** or **class** is not padded; an **_Unpacked struct** or **class** is padded.

> **CAUTION** A **struct**, **class**, or **union** is aligned according to the most stringent requirements of its members. However, members of a **_Packed struct** or **class** might not be aligned on word boundaries.

The size of an unpadded **union** is the size of its largest member. The size of a padded **union** is the size of its largest member, padded so that its size is evenly divisible by its alignment.

The size of an unpadded **struct** or **class** is the sum of the sizes of its members. Non-bit-field members always start on byte boundaries and there is no padding between members, except in bit members. The size of a padded **struct** or **class** is the sum of the sizes of its members, including alignment padding between members. A padded **struct** or **class** is padded so that its size is evenly divisible by its alignment.

For C++ only

A **struct**, **class**, or **union** containing elements specific to C++ might contain compiler-defined members that increase the size of the **struct**, **class**, or **union**. See Storage Mapping for Class Objects on page 260 for more information.

Size of Enumerations

The size of an **enum** type depends on the status of toggle `Long_enums`.

- If toggle `Long_enums` is off, the **enum** type maps to the smallest of one, two, or four bytes, such that all values can be represented.
- If toggle `Long_enums` is on, an **enum** maps to four bytes (matching the AT&T Portable C Compiler convention).

Bit Fields

MetaWare C/C++ supports both signed and unsigned bit fields.

- A bit field declared as **int** is treated as **unsigned int**.
- A bit field declared as **signed int** is interpreted as **signed**.

By default, bit fields are unsigned — that is, **int** `x:11` specifies an unsigned bit field, and **long** `x:19` specifies an unsigned long bit field. You can use the keywords **signed** and **unsigned** to specify signedness and unsignedness, respectively.

A bit field cannot exceed 32 bits and is packed in consecutive bytes.

Table 14 Bit-Field Types: Width and Range of Values shows the possible widths for bit fields, where *w* is maximum width (in bits).

Table 14 Bit-Field Types: Width and Range of Values

Bit-field Type	Max. Width *w* (Bits)	Range of Values
signed char	1 to 8	$2^{(w-1)}$-1 to -$2^{(w-1)}$
char	1 to 8	2^{w}-1 to 0
unsigned char	1 to 8	0 to 2^{w}-1
signed short	1 to 16	-$2^{(w-1)}$ to $2^{(w-1)}$-1
short	1 to 16	0 to 2^{w}-1
unsigned short	1 to 16	0 to 2^{w}-1
signed int	1 to 32	-$2^{(w-1)}$ to $2^{(w-1)}$-1
int	1 to 32	0 to 2^{w}-1
enum	1 to 32	0 to 2^{w}-1
unsigned int	1 to 32	0 to 2^{w}-1
signed long	1 to 32	-$2^{(w-1)}$ to $2^{(w-1)}$-1
long	1 to 32	0 to 2^{w}-1
unsigned long	1 to 32	0 to 2^{w}-1

A storage-unit boundary for a bit field is determined by the number of bytes needed to store a value for the associated type.

- **char** bit fields have boundaries on every byte.
- **short** bit fields have boundaries on even addresses.
- **long** bit fields have boundaries on addresses divisible by 4.

The **long long** types do not support bit fields.

Implications of Big- and Little-Endian Formats

In big-endian mode, the compiler allocates bit fields from the most significant bit to the least significant bit.

In little-endian mode, the compiler allocates bit fields from the least significant bit to the most significant bit.

How the Compiler Maps Bit Fields

The compiler maps bit fields so that they never cross a storage-unit boundary. A bit field must fit within a four-byte word that is aligned to a four-byte boundary. If necessary, the bit field starts on the following unit boundary; padding is inserted where appropriate.

Bit fields affect the boundary alignment of the enclosing **struct** or **union** as follows:

- A field that does not cross a byte boundary imposes no alignment requirement on the **struct**.
- A field that crosses a single byte boundary requires the enclosing **struct** to be two-byte aligned.
- A field that crosses two or more byte boundaries requires the enclosing **struct** to be four-byte aligned.

Named bit fields impose the same alignment on the enclosing **struct** or **union** as non-bit-field members do.

Unnamed bit fields of non-zero length do not affect the external alignment. In all other respects, they behave the same as named bit fields. An unnamed bit field of zero length causes alignment to occur at the next unit boundary, based on its type. Thus, the following alignments occur:

- A zero-length bit field of type **char** is aligned on the next one-byte boundary.
- A zero-length bit field of type **short int** is aligned on the next two-byte boundary.
- A zero-length bit field of type **int** is aligned on the next four-byte boundary.

An unnamed bit field of zero length also imposes a four-byte alignment on an enclosing **struct**.

For example, the **struct** shown below is mapped to memory as illustrated either in Figure 1 Mapping a struct (Big-Endian Mode) or Figure 2 Mapping a struct (Little-Endian Mode), according to the endian format.

```
struct {
   unsigned x:11, y:9, :0, w:13, z:1;
   char  c;
   short i;
   }
```

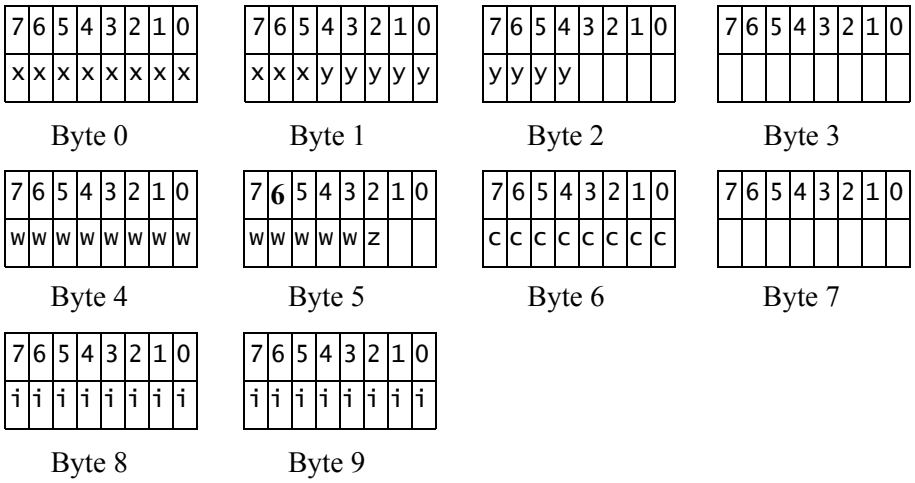

Figure 1 Mapping a struct (Big-Endian Mode)

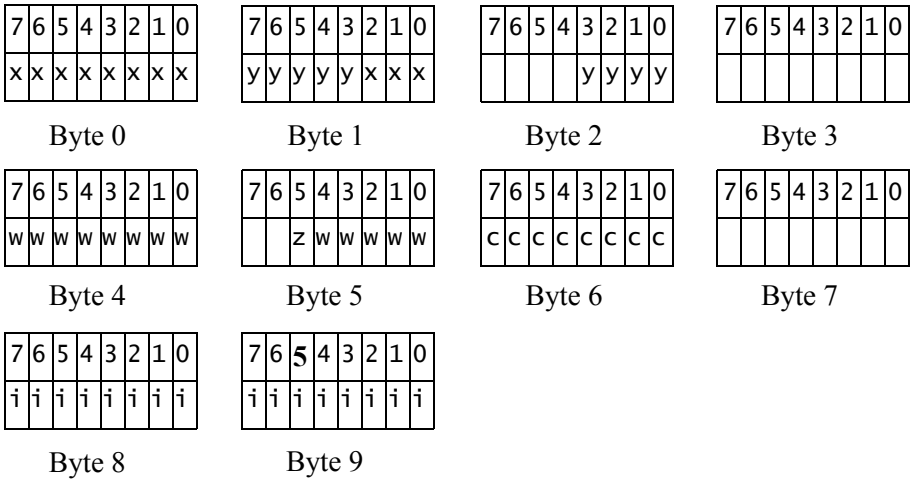

Figure 2 Mapping a struct (Little-Endian Mode)

This **struct** is aligned on address boundaries divisible by 4 because it contains **int** types. Note that the unnamed bit field (`:0`) forces padding, while alignment rules *sometimes* pad.

If `w` were changed to a **char** type, it would still be forced to begin in byte 4. If there were no unnamed bit field, `w` would begin in byte 2, 3, or 4, depending on whether it could fit in the space remaining without crossing its storage-unit boundary (which is 4).

Accessing a Volatile Object

An object that has a **volatile**-qualified type is accessed if any byte or word of the object is either read or written.

Reads and writes occur in the order implied by the source code, and using the **.di** instruction extension when toggle `Volatile_cache_bypass` is on (the default).

The **volatile** qualifier directs the compiler to not optimize any expression involving volatile variables. This is because, in a real-world application (such as an operating system), volatile variables represent data ports or physical memory addresses of memory-mapped I/O devices.

When targeting a system with a data cache, any data declared **volatile** must be aligned on a cache-line boundary. For more information, see Specifying the Alignment of Global or Static Variables on page 149.

MetaWare C/C++ for ARC also provides the **_Uncached** type qualifier for designating variables you want loaded and stored without use of a data cache. Variables declared **_Uncached** are loaded and stored using the **.di** instruction extension. By using related toggles, you can also place all **_Uncached** data in its own section and align it. For more information, see Using the _Uncached Type Qualifier on page 224.

For more information on **volatile**, **_Uncached**, and data cache, see Working with Volatile and Uncached Variables on page 222.

Preprocessing Directives

This section describes the behavior of the MetaWare C/C++ compiler in relation to preprocessing directives. For more information about the MetaWare C/C++ preprocessor, see Using the Preprocessor on page 35.

Single-Character Constants in Conditional Preprocessing Directives

In a preprocessing directive, a single-character constant in a constant expression controlling conditional inclusion is always non-negative, ranging in value from 0 (zero) to 255.

Include Files

By default, the compiler searches the standard directories for include files. For more information, see options **-Hipname**, **-I***dir*, and **-YI,***dir* in General Driver Options on page 52. Also see toggle `Relative_includes` in Alphabetical List of Compiler Toggles on page 86.

The MetaWare C/C++ compiler supports **#include** `"filename.ext"` and **#include** `<filename.ext>`.

Mapping of Source-file Character Sequences

The MetaWare C/C++ compiler does not map source-file character sequences. The compiler expects ASCII characters.

If your computer system does not support the full ASCII character set, or if your system assigns foreign-language symbols to certain character values in the ASCII character set, you might need to use trigraphs to represent particular ASCII characters. (See toggle `Trigraphs`.) For more information, see the *MetaWare C/C++ Language Reference*.

Behavior on Encountering Pragma Directives

When the MetaWare C/C++ compiler encounters a supported pragma directive, it behaves as described for that pragma in Chapter 6 — Using Compiler Pragmas. When the compiler encounters an unsupported pragma, it emits a warning message.

To suppress warning messages about an unsupported pragma, specify:

`#pragma Push_ignore(`*Unsupported_pragma_name*`)`

To restore the compiler's previous state of awareness of any pragma, specify:

`#pragma Pop_ignore(`*Pragma_name*`)`

Behavior When Date and Time Translation are Not Available

If your application is running on a system where date and time translation are not available, the **__DATE__** macro returns `jan 1 1970` and the **__TIME__** macro returns `00:00:00` (12:00 a.m.).

Compiler Limits

The limit for the number of nesting levels for files included with **#include** is 40. This is an arbitrary limit to detect recursive **#include** directives.

In compliance with the ANSI C Standard, the MetaWare C/C++ compiler places no limits on numbers of any of the following:

- Nesting levels
  - compound-statement nesting levels
  - conditional-inclusion nesting levels
  - nesting levels of parenthesized declarators within a full declarator
  - nesting levels of parenthesized expressions within a full expression
  - levels of nested structure or union definitions in a single **struct**-declaration list
- Declarators
  - pointer, array, and function declarators, in any combination, modifying an arithmetic or incomplete type in a declaration
  - pointer, array, and function declarators, in any combination, modifying a structure or union type in a declaration
- Characters
  - significant initial characters in an internal identifier or macro name
  - significant initial characters in an internal or external identifier
  - characters in a logical source line
  - characters in a character string literal, or a wide string literal after concatenation
- Number of identifiers
  - external identifiers in one translation unit
  - identifiers with block scope declared within one block
  - macro identifiers simultaneously defined in one translation unit

- Parameters
  - parameters in one function definition
  - parameters in one macro definition
- Arguments
  - arguments in one function call
  - arguments in one macro invocation
- Other components
  - bytes in an object (host environment only)
  - **case** labels for a **switch** statement, excluding those for any nested **switch** statement
  - members in a single structure or union
  - enumeration constants in a single enumeration

Storage Mapping for Class Objects

This section describes how storage is arranged for C++ class objects, treating such issues as virtual-function tables, virtual and non-virtual bases, and non-static members.

Minimum Size and Alignment

Standard C++ requires all objects to be at least one byte in size, even if they are completely empty. Alignment might further increase the minimum class size. Consider a snippet as in Example 48 An Empty Class Generates a Four-Byte Object.

Example 48 An Empty Class Generates a Four-Byte Object

```
class A {};
class B : A { int x ; };
```

This code generates an 8-byte object (four bytes for `B` and four inherited from `A`), preserving space in class `B` for inherited class `A`.

The minimum-size requirement also allows you to pass a `B*` into an object expecting an `A*` without the two sharing the storage space. This is useful when conditional compilation results in classes with no members.

Virtual-Function Tables

A class that contains virtual functions must have a virtual-function table, which is accessed whenever your program calls a virtual function. Each instance of a class with virtual functions contains a pointer to its virtual-function table. For example, if **fnc()** is a virtual function in class *x*, `x.fnc()` essentially becomes `(x.vtab[`*index_of_fnc*`])()`, where *index_of_fnc* is the index of the function's address in the virtual-function table.

To help guarantee that the virtual function tables for the class are emitted, the compiler uses the COMDAT solution described in the *MetaWare C/C++ Language Reference* and emits multiple copies of the tables for each file that might need them. The linker combines the copies so that the final image contains only one.

To avoid having multiple copies of the virtual-function tables for a class, turn off toggle `Class_common_ok`.

If a class's member functions are all pure **virtual** or **inline**, the compiler makes a copy of the class's virtual-function table for each compilation unit that makes use of the virtual-function table. The compiler issues a warning for each such class. You can turn the warning off with pragma `Offwarn`.

We recommend that you not make a class's destructor be **inline** if all other functions in the class are pure **virtual** or **inline**, because the virtual tables will then be generated in the module in which you define the destructor. For related information, see toggles `Import_vtabs` and `Export_vtabs`.

Direct and Indirect Bases

A *direct base* of a class is another class listed in the first class's declaration. For example, in this declaration, classes **Y** and **Z** are direct bases of **X**:

```
class X: Y,Z {...}
```

An *indirect base* of a class *X* is a class that is not a base of *X* but is a base of one of *X*'s direct bases. For example, if we add this declaration, class **X** is a direct base of **A**, and **Y** and **Z** are indirect bases of **A**:

```
class A: X {...}
```

NOTE For information about virtual and non-virtual bases, see the *MetaWare C/C++ Language Reference*.

Suppose a class *X* has the following base classes:

- direct non-virtual-base classes
- direct virtual-base classes
- indirect virtual-base classes

Storage for the components of an instance of an object of class *X* with two virtual base classes is organized as shown in Figure 3 Storage for an Instance of an Object of Class X, except that redundant pointers might be optimized away.

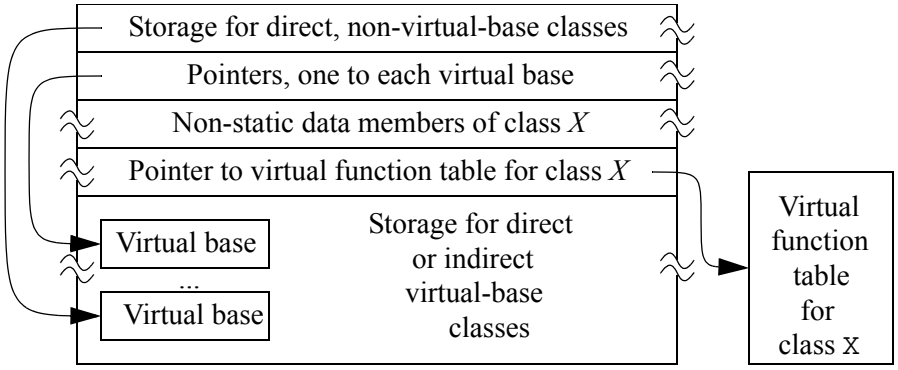

Figure 3 Storage for an Instance of an Object of Class X

The non-static data members are allocated the same way as **struct** members in plain C; that is, consecutively as they appear in the source, subject to boundary alignment. Access specifiers do not affect the mapping.

The direct, non-virtual bases laid out at the beginning of class *X*'s storage do not include those bases' own virtual bases (if any); the storage for all virtual bases is in class *X* and not in any of its bases. *X* has a pointer to each of its direct virtual bases. When *X* is constructed, these pointers are initialized; so, too, are any virtual-base pointers that reside in *X*'s direct or indirect bases (virtual or not). So the

storage for any base *B* allocated in an instance of *X* does not include any virtual bases of *B*. The storage for *B* in *X* may be thought of as a "truncated instance" of *B*.

If possible, rather than allocating a pointer to a virtual-function table within *X*, the compiler uses that same pointer in the first direct, non-virtual-base class *B* — provided *B* also has virtual functions. The compiler rearranges the storage of direct, non-virtual-base classes so that the first base stored has a virtual-function pointer, if any such base does. Consider this example:

```
class A: B,X,D
```

In this case, if **X** has virtual functions and **B** does not, class **X** is stored first, followed by **B** and then **D**.

Examples of Class Layout

This section illustrates two examples of class layout:

- Layout for an instance of class **D**
- Layout for virtual bases that have virtual bases

Example 49 Layout for an Instance of Class D

```
class VB { int vb; };
class B  { int b; };
class D: B, virtual VB { int d; };
```

Figure 4 Layout of an Instance of Class D shows the layout for an instance of class **D**.

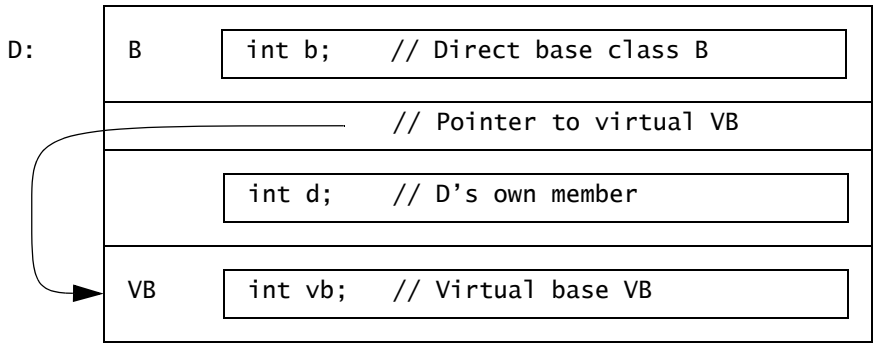

Figure 4 Layout of an Instance of Class D

Example 50 Class Layout for Virtual Bases That Themselves Have Virtual Bases

```
class VA { int va; };
class VB { int vb; };
class DVA: virtual VA {
   int dva;
   };                          // From virtual VA
class DVB: virtual VB {
   int dvb;
   };                          // From virtual VB
class DVB2: virtual VB {
   int dvb2;
   };                          // From virtual VB
class D: virtual DVA, virtual DVB,
   virtual VB,   DVB2 { int d; };
```

The following output shows the class lattice that the compiler produces for this example.

```
D {d}
---------------------------------------------
v:DVA {dva} v:DVB {dvb} v:VB {vb} DVB2 {dvb2}
```

```
----------- -----------            -----------
v:VA {va}   v:VB {vb}              v:VB {vb}
```

In Example 50 Class Layout for Virtual Bases That Themselves Have Virtual Bases on page 262, there are two pointers to virtual base **VB**: one within **DVB** and one within **DVB2**. When an object of class **D** is constructed, both are initialized to point to the single occurrence of **VB**. There would have been yet another pointer to **VB** from **D**, but it is optimized away. A total of five virtual-base pointers are initialized when a **D** object is constructed. See toggle `Print_lattices` for more information about printing class lattices.

Figure 5 Object Layout for an Instance of Class D shows the object layout for an instance of **D**.

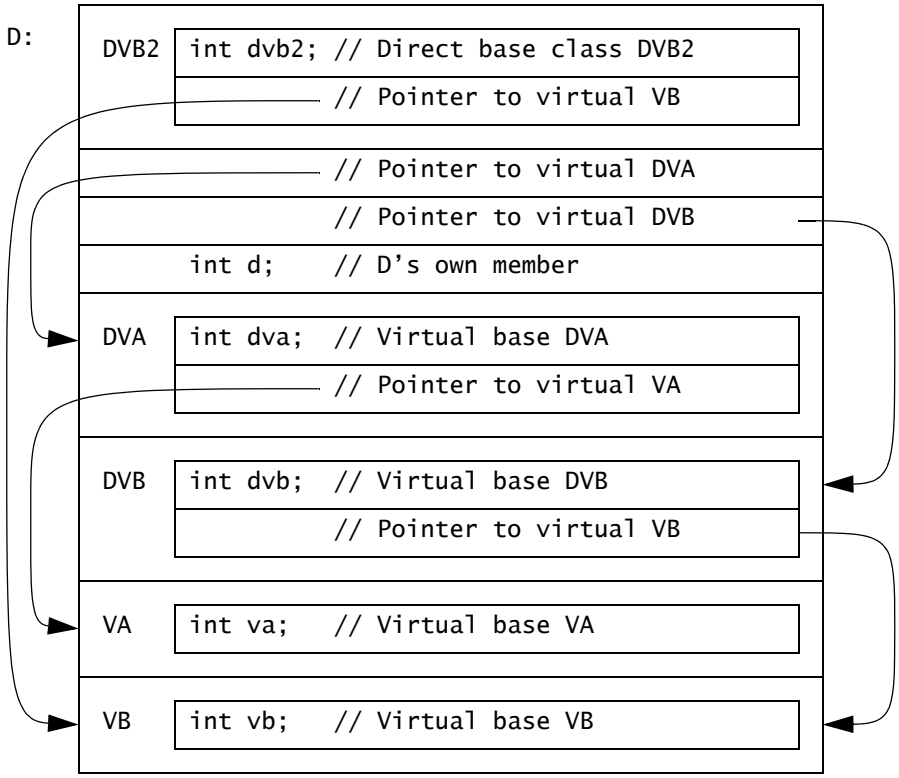

Figure 5 Object Layout for an Instance of Class D

Chapter 12 — Assembly Interface

In This Chapter

- Overview
- Making Assembly Routines Accessible from MetaWare C/C++
- Function-Naming Conventions
- Calling Assembly Routines from C or C++
- Calling C or C++ Functions from Assembly Routines
- Sharing Variables across Modules
- Direct Register Assignment for Variables

Overview

This chapter describes MetaWare C/C++ conventions for C and C++ functions and their interaction with assembly-language routines.

The MetaWare C/C++ conventions ensure that the linker can correctly link the C and C++ language modules with assembly-code modules you have written to optimize program performance.

By using the MetaWare C/C++ conventions, you ensure the sharing of global data among your program's functions: C and C++ functions can call assembly functions, and assembly functions can call C and C++ functions.

Making Assembly Routines Accessible from MetaWare C/C++

To call a general assembly-language routine from a MetaWare C/C++ program, you must write the assembly as shown in the following snippet of code.

```
        .text
        .global name
name:
```

In this code, *name* is the routine's name as referenced from C.

For a description of how arguments are passed and how function results are returned, see Chapter 13 — Run-Time Organization.

Function-Naming Conventions

This section covers function-naming conventions in C and C++.

Naming Conventions in C

A function that is global (that is, a function that can be called across module boundaries) must have information provided to the linker that associates the name of the function with its address.

The name in the symbol table does not need to be the same as the actual name of the function; it only needs to be derivable from it. The symbol-table entry is called the *external presentation* of the internal name.

In assembly language, the name that is placed in the symbol table is lexically identical to the name of the corresponding symbol.

In C the external presentation is, by default, lexically identical to the internal name.

Because the external presentation is identical to the internal name in both C and assembly, aliasing is not required. However, you can use the pragmas described below to rename functions when declaring them.

Changing External Presentations

The MetaWare C/C++ compiler provides two pragmas for changing external presentations of internal names:

- You can use pragma `Global_aliasing_convention` to cause all external names to be altered: prefixed by an underscore, for example, or converted to uppercase.
- You can use pragma `Alias` to define an alternate external presentation of a single identifier (that is, to explicitly define how a public symbol is to appear in the symbol table).

Using Pragma Global_aliasing_convention

This code fragment shows how to use pragma `Global_aliasing_convention` to call an assembly function named **_fnc** from a MetaWare C/C++ program.

```
/* Add '_' prefix to all symbol-table names. */
#pragma Global_aliasing_convention("_%r")
...
/* Function fnc() is affected by aliasing. */
extern fnc();
...
/* No more aliasing.     */
#pragma Global_aliasing_convention()
...
void main() {
   ...
   fnc(); /* Call '_fnc' */
   ...
   }
```

Using Pragma Alias

This code fragment shows how to use pragma `Alias` to call an assembly function named **_fnc** from a MetaWare C/C++ program.

```
extern fnc();
...
/* Call '_fnc' instead of default 'fnc'. */
...
#pragma Alias(fnc,"_fnc")
...
void main() {
   ...
   fnc();
   ...
   }
```

Naming Conventions in C++

In C++, function names in the symbol table are in mangled form. The compiler derives the mangled name from the function's source-code name, its argument types, and any enclosing class definitions. It is difficult to access a C++ function from assembly language because of the name mangling.

To access an assembly-language function from C++, declare the function as a C function using **extern** `"C"`. The compiler does not mangle C function names.

For information about a utility that unmangles function names, see the *ELF Linker and Utilities User's Guide*.

Calling Assembly Routines from C or C++

Example 51 Calling an Assembly Function from MetaWare C/C++ demonstrates how to call assembly language function **peek()** from MetaWare C/C++ code.

The call to **peek()** returns the byte at address 0x8000.

Example 51 Calling an Assembly Function from MetaWare C/C++

MetaWare C/C++

```
/*
 * Uncomment 'extern "C"' for C++. This command causes
 * the C++ compiler to avoid C++ name mangling.
 */
/* extern "C" */ int peek(long adr);
...
void main(){
   char  b;
   ...
   b = peek(0x8000);
   ...
   }
```

Assembly

```
        .text
        .global peek
peek:
        j.d     [%blink]
        ldb.di  %r0,[%r0]
```

Calling C or C++ Functions from Assembly Routines

You can call C or C++ functions from assembly routines directly, or indirectly using function pointers.

Calling Functions Directly

To call a C or C++ function directly from an assembly routine, do the following:

1. Load the first eight words of arguments into registers `%r0` — `%r7`.
2. Push additional arguments onto the stack in right-to-left order.
3. ARCtangent-A4 only: Subtract 16 from the stack pointer (`%sp` register).
4. Execute the **bl** instruction.
5. ARCtangent-A4 only: Restore the stack pointer to the value it had before the calling sequence.

Example 52 Calling a C or C++ Function from Assembly illustrates this process.

Example 52 Calling a C or C++ Function from Assembly

MetaWare C/C++

```
/*
 * Uncomment 'extern "C"' for C++. This command causes
   the compiler to avoid C++ name mangling.
 */
/* extern "C" */ void write_string (char *s){
   printf("%s\n", s);
   }
```

Assembly for ARCtangent-A4

```
.L0:
        .asciz  "hello"
        sub     %sp, %sp, 16
```

```
; Call write_string
        mov     %r0, .L0
        bl      write_string
        add     %sp, %sp, 16
```

Assembly for ARCtangent-A5 and Later

```
.L0:
        .asciz  "hello"
; Call write_string
        mov_s   %r0, .L0
        bl      write_string
```

Calling Functions Indirectly

If you wish to call a function indirectly using function pointers, first load the address of the function into a register and then call it through the register. All other steps are the same as the steps for calling directly listed in Calling Functions Directly on page 268, except that instead of using the **bl** instruction to call directly, you use the **jl** instruction to call indirectly. However, for ARCtangent-A4 and previous processors, you must use the `@h30` identifier to achieve the required two-bit shift.

ARCtangent-A4

In ARCtangent-A4 and previous processors, only the high-order 30 bits of a 32-bit address are represented in a register to be used for an indirect call or jump. Thus, the full 32-bit address must be shifted right by two in the address to be loaded. The linker performs the required shifting when you use the `@h30` identifier attribute (documented in the *ELF Assembler User's Guide*). See Example 53 Calling a MetaWare C/C++ Function from Assembly Indirectly.

Example 53 Calling a MetaWare C/C++ Function from Assembly Indirectly

(ARCtangent-A4)

```
  mov   %r0, write_string@h30  ; load function pointer
  jl    [%r0]                  ; call function indirectly
```

All function pointers generated by the compiler are initialized with the `@h30` identifier attribute to insure that only the high-order 30 bits are represented and are compatible with the behavior of the machine instructions that use them.

ARCtangent-A5 and Later

In ARCtangent-A5 and later processors, function pointers consist of the full 32-bit address of the function, and should be called without using the `@h30` identifier attribute, as shown in Example 54 Calling a MetaWare C/C++ Function from Assembly Indirectly.

Example 54 Calling a MetaWare C/C++ Function from Assembly Indirectly

(ARCtangent-A5 and Later)

```
     mov   %r0, write_string  ; load function pointer
     jl    [%r0]              ; call function indirectly
```

Sharing Variables across Modules

Global variables in MetaWare C/C++ programs are aliased in the same manner as public functions.

The external presentation of a global variable's name in a MetaWare C/C++ program is determined in the same way as the external presentation of an external function's name, and can be overridden in the same way with pragma `Alias` or pragma `Global_aliasing_convention`. (See Naming Conventions in C on page 266 for details.)

How the Compiler Maps Variables to Storage on page 252 explains how the various C data types are mapped into storage. Note that uninitialized global variables without the **extern** qualifier are actually defined as individual global data blocks, unless toggle `Multiple_var_defs` is on, in which case they are made common (that is, merged).

NOTE If accessing a variable stored in a small-data section, you must use identifier attribute `@sda` (documented in the *ELF Assembler User's Guide*).

Example 55 Sharing Variables across C and Assembly Modules illustrates code that shares variables across C and assembly.

Example 55 Sharing Variables across C and Assembly Modules

MetaWare C/C++

```
int     alpha, beta;
char    hextable[] = "0123456789ABCDEF";
/* Read-only table of names: */
extern  char *names[];
extern  short status;
```

Assembly

```
        .comm   alpha,4
        .comm   beta,4
        .extern hextable ; Imported from C
        .rodata
        .global names    ; Read-only segment
names:  .long   L01
        .long   L02
        .long   L03
        .long   0
        .align  4
L01:    .asciz  "alfred"
        .align  4
L02:    .asciz  "bonny"
        .align  4
L03:    .asciz  "charlie"
        .data
        .global status
status: .short  -1
```

Direct Register Assignment for Variables

You can use direct register assignment (an extension to the C and C++ languages) to assign a register directly to a variable. Simply use the syntax

`==` *integer_constant_expression*

in the variable's declaration. For example:

```
int i == 3;
```

The numbers map directly to register numbers. For example, `0` maps to `%r0`, `1` maps to `%r1`, and so on. For more information about the registers, see Register Usage on page 274.

The compiler issues an error message if it cannot map the variable into the specified register. Reasons for not mapping a variable include the following:

- The *integer_constant_expression* is an invalid number (it does not map to a register).
- The register is not available (for example, `sp` or *pc*).

NOTE The compiler also reserves the right to not map the variable simply because too many variables have already been mapped.

For direct register assignment with local variables, the optimizer attempts to avoid using the registers assigned to variables. However, this avoidance might not be possible when registers are assigned dedicated purposes.

For global variables, the compiler reserves the register for the global variable and does not use it for other purposes by default (barring any explicit use of pragma `Core_register` to instruct the compiler otherwise). This provides a means of having "fast" global variables. Note that when you are using an RTOS, such variables tend to be inherently thread-local unless specific arrangements are made otherwise.

For example, consider this assignment, at the global scope in an application:

Example 56 Assigning a Global Variable Directly to a Register

```
my_struct *data_block == 23;
```

By default, register `r23` here is reserved by the compiler (not used for code generation), and references to **data_block** are translated as references to `r23`.

NOTE Direct register assignment does not guarantee precise control of read and write access to registers. If you require explicit control of register access, use the **_core_read()** and **_core_write()** intrinsic functions instead.

Chapter 13 — Run-Time Organization

In This Chapter

- Overview
- Register Usage
- Stack Conventions
- Passing Arguments
- Return Values
- Prolog and Epilog Code
- Dynamic Stack Allocation

Overview

This chapter describes the function-calling sequence and explains how that communication uses the stack frame.

The *function-calling sequence* is defined as the communication that occurs at run time between a function that is called (the *callee*) and the function that calls it (the *caller*).

NOTE Hand-written assembly must conform to the run-time model described in this chapter. For information about communication between C or C++ and assembly modules, see Chapter 12 — Assembly Interface.

Register Usage

A base-case processor provides 32 word-sized (32-bit) registers and a number of special-purpose auxiliary registers.

When you use ARC 600 option **-rf16** for a reduced register set, registers `r4` to `r9` and `r16` to `r25` are not available. The remaining registers have the functions listed in this section.

NOTE Auxiliary registers can be meaningfully used only by the **LR** and **SR** instructions. When you use an auxiliary register in any other context, the assembler treats it as a symbolic constant. For example, the following assembly lines are equivalent:

```
ror %identity, %identity
ror 0x4, 0x4
```

This code results in an immediate value for the destination. To the processor, this means that the result is being thrown away. The result is the same to the processor, no matter what the immediate value.

General-Purpose Registers

Table 15 General and Program-Counter Register Functions summarizes the general and program-counter registers and their functions.

Table 15 General and Program-Counter Register Functions

Name	Primary Function	Secondary Function
`r0`	Integer result; argument 1	Volatile scratch register
`r1`	Argument 2	Volatile scratch register
`r2`	Argument 3	Volatile scratch register
`r3`	Argument 4	Volatile scratch register
`r4`	Argument 5	Volatile scratch register
`r5`	Argument 6	Volatile scratch register
`r6`	Argument 7	Volatile scratch register
`r7`	Argument 8	Volatile scratch register
`r8`	Volatile scratch register	—
`r9`	Volatile scratch register	—
`r10`	Volatile scratch register	—
`r11`	Volatile scratch register	Used to pass the static link for supporting nested functions

Table 15 General and Program-Counter Register Functions (con't)

Name	Primary Function	Secondary Function
r12	Volatile scratch register used for temporary calculations	—
r13 — r25	Non-volatile register variable (callee saved)	—
r26	Small-data base register (gp)	—
r27	Frame pointer (fp)	—
r28	Stack top pointer (sp)	—
r29	Maskable interrupt link register 1 (ilink1)	—
r30	Maskable interrupt link register 2 (ilink2)	—
r31	Branch link register (blink)	—
r60	Loop counter (lp_count)	—
r63	ARCtangent-A5 and later program-counter value (pcl)	

The lp_count register (r60) is the loop-counter register. It is 24 bits in ARCtangent-A4 and 32 bits in ARCtangent-A5 and later.

See Passing Arguments on page 283 and Data Storage on page 247 for related information.

Auxiliary Registers

Table 16 summarizes the most commonly used auxiliary registers. Due to the large number of auxiliary registers possible on an ARC processor, this listing is necessarily incomplete, and may vary from one implementation to another. Refer to the *Programmer's Reference Manual* for a complete listing.

Table 16 Auxiliary-Register Functions

Name	Function	
0x0	status (ARCtangent-A4)	
0x1	Inter-process/host semaphore	
0x2	Loop start address (lp_start)	
0x3	Loop end address (lp_end)	
0x4	Processor identification	
0x5	debug	
0x6	Program counter (nextpc)	ARCtangent-A5 and later
0xa	Condition flags (status32)	ARCtangent-A5 and later
0xb	Status save for level-1 exceptions	ARCtangent-A5 and later
0xc	Status save for level-2 exceptions	ARCtangent-A5 and later
0x21	Processor-timer-0 count value	
0x22	Processor-timer-0 control value	
0x23	Processor-timer-0 limit value	
0x25	Interrupt-vector base address	
0x43	Interrupt-level status	
0x68	Default vector-base build configuration	
0x100	Processor-timer-1 count value	
0x101	Processor-timer-1 control value	
0x102	Processor-timer-1 limit value	
0x200	Interrupt-level programming	
0x201	Software interrupt	
0x400	Exception return address	ARC 700

Table 16 Auxiliary-Register Functions

0x401	Exception-return branch-target address	ARC 700
0x402	Exception-return status	ARC 700
0x403	Exception cause	ARC 700
0x404	Exception-fault address	ARC 700
0x40a	Level-1-interrupt cause	ARC 700
0x40b	Level-2-interrupt cause	ARC 700
0x40c	Interrupt-mask programming	ARC 700
0x40d	Interrupt-sensitivity programming	ARC 700
0x410	User-mode extension enables	ARC 700
0x412	Branch-target address	ARC 700
0x413	Level-1-return branch target	ARC 700
0x414	Level-2-return branch target	ARC 700
0x415	Interrupt-pulse cancel	ARC 700
0x416	Interrupt pending	ARC 700

ARCtangent-A4

The `status` auxiliary register contains the program counter and the condition flags.

ARCtangent-A5 and Later

The `nextpc` auxiliary register contains the program counter; the `pcl` register contains the 4-byte aligned value of the program counter. The `status32` auxiliary register contains the condition flags.

For information on which registers can be used by which 16-bit instructions, see the *Programmer's Reference Manual* for your processor.

Stack Conventions

The run-time model implements a stack frame in memory. This stack frame is used for local variables that cannot be contained in registers, and as temporary storage for the contents of certain registers.

Stack Mechanics

ARCtangent-A4

The stack is a singly-linked list of activation records (stack frames) that are linked through a stack back-trace data structure that is stored at the high-address end of each frame. The back-trace data structure for a called function is allocated in the caller's stack frame. The remainder of the stack frame required by a called function is allocated by the function's prolog code.

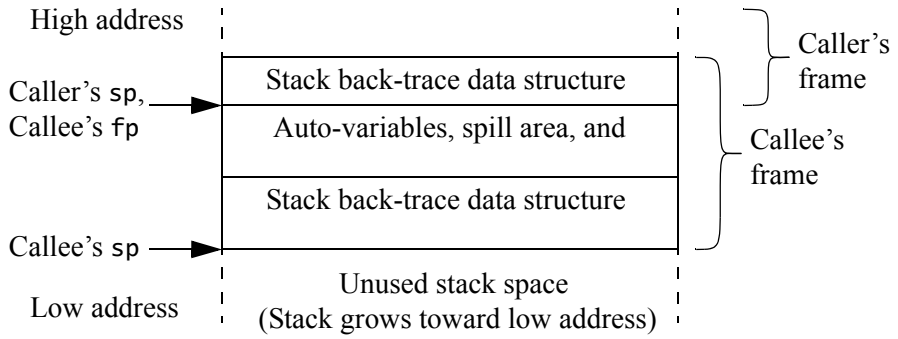

Figure 6 Stack Frame (ARCtangent-A4)

ARCtangent-A5 and Later

The stack is a sequence of activation records. An activation record is private to a function and is generated for all functions that must save and restore registers or that require the use of temporary storage for local variables and computations.

Leaf functions (functions that do not make any calls to other functions) might not require an activation record.

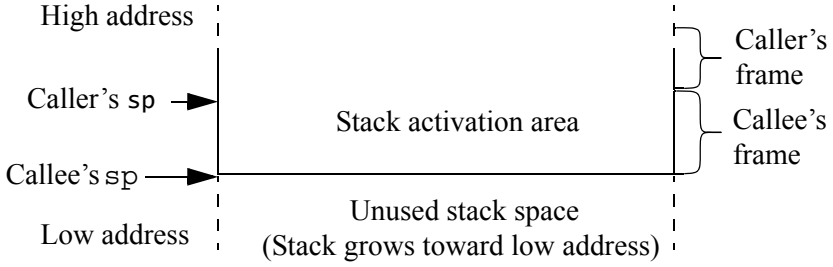

Figure 7 Stack Frame (ARCtangent-A5 and Later)

The Stack-Pointer Register

The stack-pointer (`sp`) register always points to the lowest used address of the most recently allocated stack frame. The value of `sp` is a multiple of 4.

The Frame-Pointer Register

ARCtangent-A4

The frame pointer (`fp`) register always points to a back-trace data structure that can be used to back-trace through the currently outstanding function calls. The value of `fp` is a multiple of 4.

ARCtangent-A5 and Later

The frame pointer (`fp`) register is used in some cases, such as for nested functions or when a function calls **alloca()** to allocate space on the stack.

The callee's Prolog Code

ARCtangent-A4

The callee's prolog code saves everything that needs to be saved. Saved values include the value of the caller's frame-pointer (`fp`) register and the value of `blink`, the return address register. If a nested function is executing, the callee's prolog code also saves the static link frame pointer, so that variables in outer scopes can be accessed.

The caller's stack-pointer (`sp`) register does not need to be saved because, after saving the `fp` register, the callee's prolog code initializes the `fp` register for the new frame with the value of the caller's `sp` register.

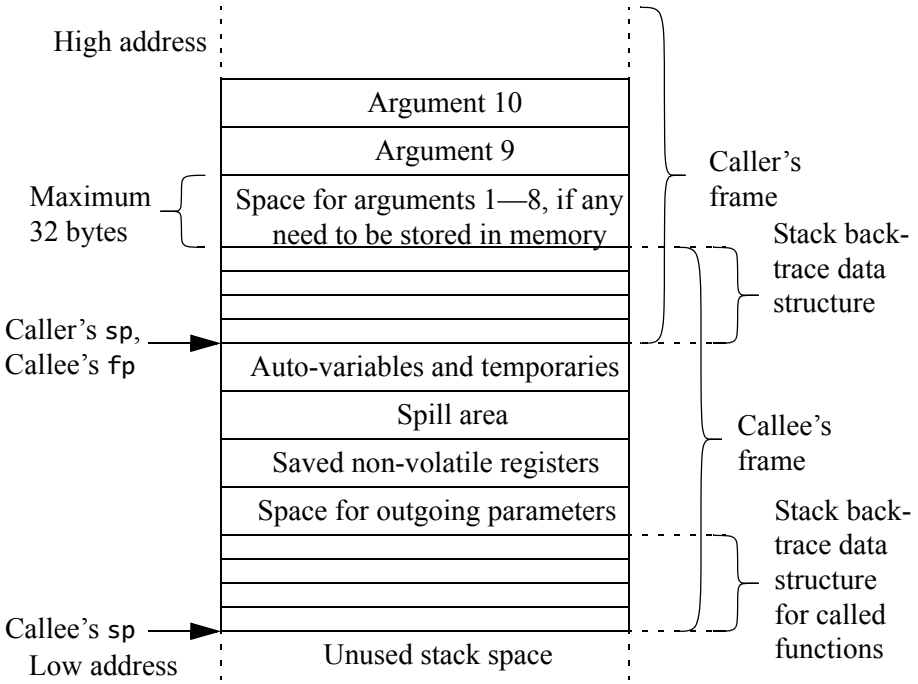

Figure 8 Stack Frame for one Function Invocation (ARCtangent-A4)

Stack Back-Trace Data Structure (ARCtangent-A4 and Previous)

Figure 9 Stack Back-Trace Data Structure (ARCtangent-A4 and Previous) illustrates the stack back-trace data structure that occupies the high-address end of each stack frame. The back-trace data structure is initialized in the entry code, or prolog, of a called function.

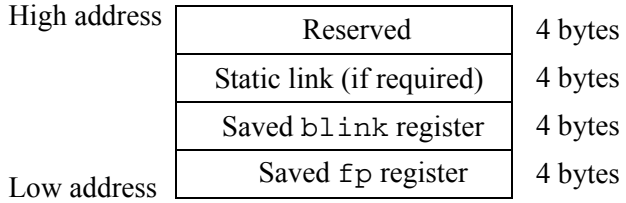

Figure 9 Stack Back-Trace Data Structure (ARCtangent-A4 and Previous)

The back-trace data structure is allocated in the caller's frame (at the low end, within reach of the caller's `sp`), but it is primarily a part of the callee's frame (at the high end). That is, the caller's and callee's frames overlap at the back-trace data structure.

From the caller's perspective, the room it allocates for the callee's back-trace data structure is just uninitialized memory — the caller does not reference it. Only the callee initializes and uses the back-trace data structure allocated by the caller. Having the caller allocate the back-trace data structure makes the callee's standard epilog code more efficient, because the caller's `fp` and `sp` can be restored in one instruction.

ARCtangent-A5 and Later

The callee's prolog code saves all registers that need to be saved. Saved values include the value of the caller's `blink` (return address) register, non-volatile registers used by the function, and the frame-pointer register if required. If a nested function is executing, the callee's prolog code might also save the static-link frame pointer to provide access to variables in outer scopes.

The caller's stack-pointer (`sp`) register does not need to be saved because the compiler is able to restore the stack pointer for each function to its original value (for example, by using an **add** instruction).

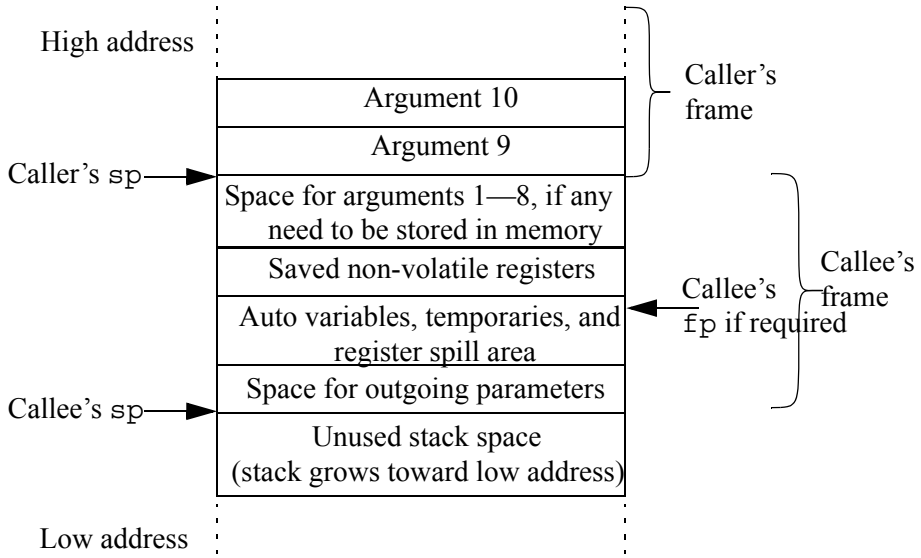

Figure 10 Stack Frame for One Function Invocation (ARCtangent-A5 and Later)

Interrupt Stack Frames

Interrupt handlers are most efficient if they make no function calls because the compiler knows precisely what state needs to be preserved. For limited calling to local functions, calling-convention attribute **_SAVE_ALL_REGS** improves efficiency.

New stack space must be allocated for states saved by the interrupt handler. No general purpose registers or flags may be altered by the interrupt function without being saved on interrupt and restored prior to returning from the interrupt. On ARCtangent-A4 and previous, the interrupt handler must allocate a new back-trace data structure before calling a C or C++ function.

You can meet these requirements by writing your interrupt handlers in C or C++ and using calling-convention attributes **_INTERRUPT1** and **_INTERRUPT2**. For more information, see Writing Interrupt Handlers on page 224.

Example 57 Compiler Output for an Interrupt Handler (ARCtangent-A4) shows the ARCtangent-A4 compiler output for an interrupt handler.

Example 58 Interrupt Handler with a Function Call (ARCtangent-A5 and Later) on page 282 shows an interrupt handler for ARCtangent-A5 or later with the resulting assembly, and Example 59 Interrupt Handler without a Function Call (ARCtangent-A5 and Later) on page 283 shows the same interrupt handler for ARCtangent-A5 or later without a function call, and the resulting assembly.

Example 57 Compiler Output for an Interrupt Handler (ARCtangent-A4)

```
;; Enough space for all saved info
.set _ISR_FPSIZE, 128
```

```
    ;%sp ----------------------------------
    ; + 0:   Back-trace data structure
    ; + 4:   for C or C++ functions
    ; + 8:   called from this
    ; +12:   handler
    ; -------------------------------------
    ; +16: %fp+ 0: Saved %fp
    ; +20: %fp+ 4: Saved %ilink1 or %ilink2
    ; -------------------------------------
    ; +24:  Saved %r0
    ; +28:  Saved %r1
    ; +32:  Saved %r2
    ; +36:  Saved %r3
    ; +40:  Saved %r4
    ; +44:  Saved %r5
    ; +48:  Saved %r6
    ; +52:  Saved %r7
    ; +56:  Saved %r8
    ; +60:  Saved %r9
    ; +64:  Saved %r10
    ; +68:  Saved %r11
    ; +72:  Saved %r12
    ; +76:  Saved %blink
    ; +80:  Saved %lp_start  C/C++ may use
    ; +84:  Saved %lp_end    zero delay
    ; +88:  Saved %lp_count  loops
    ; ...:
    ; ...:  Saved AUX registers and other
    ; ...:  state for processor extensions (such
    ; ...:  as a multiplier with %mlo and %mhi).
    ; ...:
    ; -------------------------------------

    sub     %sp, %sp, _ISR_FPSIZE
    ; Allocated space to save state.

    ; Assume top 16 bytes is back-trace data
    ; structure allocated for C/C++ functions to be
    ; called from this handler.

.ifdef ISR_DEBUG
    st      %fp,     [%sp, 16]
    st      %ilink1, [%sp, 20]
    add     %fp, sp, 16
.endif

    ; Because we save %ilink1 and %fp and re-allocate
    ; %fp like a real back-trace data structure, the
    ; debugger displays the call stack from this
    ; interrupt in a useful way. You may omit the above
    ; three instructions for performance at the cost of
    ; clean debugging.

    st    %r0, [%sp, 24]
    st    %r1, [%sp, 28]
    st    %r2, [%sp, 32]
    st    %r3, [%sp, 36]
    st    %r4, [%sp, 40]
    st    %r5, [%sp, 44]
```

```
    st    %r6, [%sp, 48]
    st    %r7, [%sp, 52]
    st    %r8, [%sp, 56]
    st    %r9, [%sp, 60]
    st    %r10, [%sp, 64]
    st    %r11, [%sp, 68]
    st    %r12, [%sp, 72]
    st    %blink, [%sp, 76]
    lr    %r10, [%lp_start]         ; Pickup lp_start
    lr    %r9,  [%lp_end]           ; Pickup lp_end
    st    %lp_count, [%sp, 88]      ; Save lp_count
    st    %r10, [%sp, 80]           ; Save lp_start
    st    %r9,  [%sp, 84]           ; Save lp_end

.ifdef    __Xmult32
    st      %mlo, [%sp, 92]             ; Save mlo
    st      %mhi, [%sp, 96]             ; Save mhi
.endif

   ;; Can call C/C++ functions here.

    ;; epilog

.ifdef    __Xmult32
    ld    %r1,  [%sp, 92]       ; Recover mlo value
    ld    %r0,  [%sp, 96]       ; Recover mhi value
    mulu64  0, %r1, 1
    ; Restore mlo (can take up to 10 cycles).
.endif

    ld    %r10,      [%sp, 80]    ; Recover lp_start
    sr    %r10,      [%lp_start]  ; Restore lp_start
    ld    %r11,      [%sp, 84]    ; Recover lp_end
    sr    %r11,      [%lp_end]    ; Restore lp_end
    ld    %lp_count, [%sp, 88]    ; Recover lp_count
    ld    %r2,       [%sp, 32]
    ld    %r3,       [%sp, 36]
    ld    %r4,       [%sp, 40]
    ld    %r5,       [%sp, 44]
    ld    %r6,       [%sp, 48]
    ld    %r7,       [%sp, 52]
    ld    %r8,       [%sp, 56]
    ld    %r9,       [%sp, 60]
.ifdef    __Xmult32
    sr    %r0,       [%mulhi]
    ; Stores to mulhi restores mhi and sets up mmid.
.endif

    ld    %r10,   [%sp, 64]
    ld    %r11,   [%sp, 68]
    ld    %r12,   [%sp, 72]
    ld    %blink, [%sp, 76]
    ld    %r0,    [%sp, 24]
    ld    %r1,    [%sp, 28]

    ;; Restores status of flags and program counter.
    j.d.f   [%ilink1]
    add     %sp, %sp, _ISR_FPSIZE
```

Example 58 Interrupt Handler with a Function Call (ARCtangent-A5 and Later)

MetaWare C/C++ for ARCtangent-A5 and Later

```
extern void doit(unsigned);

_Interrupt1 void handler(void) {
   pragma interrupt_ext_words(2);
   long *p = _interrupt_ext_addr();
   p[0] = _core_read(50);
   p[1] = _core_read(51);
   doit(0x8000);
   _core_write(p[0], 50);
   _core_write(p[1], 51);
   }
```

Assembly for ARCtangent-A5 and Later

```
   push_s   %blink
   push_s   %r12
   push     %lp_count    ; save zero-delay-loop registers
   lr       %r12, [%lp_start]
   push_s   %r12
   lr       %r12, [%lp_end]
   push_s   %r12
   push_s   %r0
   push_s   %r1
   push_s   %r2
   push_s   %r3
   push     %r4
   push     %r5
   push     %r6
   push     %r7
   push     %r8
   push     %r9
   push     %r10
   push     %r11
   push_s   %r13
   sub_s    %sp,%sp,8    ; allocate requested space
   mov_s    %r13,%sp     ; and save extension registers
   st       %r50, [%r13]
   st       %r51, [%r13, 4]
   mov_s    %r0,0
   bl.d     doit
   bset_s   %r0,%r0,15
   ld       %r50, [%r13]  ; restore saved extension state
   ld       %r51, [%r13, 4]
   add_s    %sp, %sp, 8   ; recover requested space
   pop_s    %r13          ; restore all registers
   pop      %r11
   pop      %r10
   pop      %r9
   pop      %r8
   pop      %r7
   pop      %r6
   pop      %r5
   pop      %r4
   pop_s    %r3
   pop_s    %r2
```

```
   pop_s    %r1
   pop_s    %r0
   pop_s    %r12
   sr       %r12, [%lp_end]
   pop_s    %r12
   sr       %r12, [%lp_start]
   pop      %r12
   mov      %lp_count, %r12
   pop_s    %r12
   pop_s    %blink
   j.f      [%ilink1]      ; return from interrupt,
                           ; restoring flags
```

Example 59 Interrupt Handler without a Function Call (ARCtangent-A5 and Later)

MetaWare C/C++ for ARCtangent-A5 and Later

```
_Interrupt1 void handler(void) {
   pragma interrupt_ext_words(2);
   long *p = _interrupt_ext_addr();
   p[0] = _core_read(50);
   p[1] = _core_read(51);
   _core_write(p[0], 50);
   _core_write(p[1], 50);
   }
```

Assembly for ARCtangent-A5 and Later

```
handler:
   push_s   %r12
   push_s   %r0
   push_s   %r1
   sub_s    %sp,%sp,8
   mov_s    %r1,%sp
   st       %r50, [%r1]
   mov_s    %r0,%r51
   st_s     %r0,[%r1, 4]
   ld       %r50, [%r1]
   ld       %r51, [%r1, 4]
   add_s    %sp, %sp, 8
   pop_s    %r1
   pop_s    %r0
   pop_s    %r12
   j.f      [%ilink1]
```

Passing Arguments

Arguments are passed as an ordered list of machine-level values from the caller to the callee.

- The first eight words (32 bytes) of arguments are loaded into registers `r0` to `r7`.
- The remaining arguments are passed by storing them into the stack immediately above the stack back-trace data structure (ARCtangent-A4) or stack-pointer register (ARCtangent-A5 and later).

All arguments are passed this way. For information about the single value returned from a callee to the caller, see Return Values on page 284.

ARCtangent-A4

Figure 8 Stack Frame for one Function Invocation (ARCtangent-A4) on page 278, shows space for arguments 1 — 8 (if they must be stored in memory) above the back-trace data structure and below

argument 9. Before initializing the back-trace data structure, the callee (not the caller) allocates space for any of arguments 1 — 8 whose addresses must be taken. The caller is ignorant of this space.

ARCtangent-A5 and later

Figure 10 Stack Frame for One Function Invocation (ARCtangent-A5 and Later) on page 279, shows space for arguments 1 to 8 (if they must be stored in memory) below argument 9. Before saving non-volatile registers, the callee (not the caller) allocates space for any of arguments 1 to 8 whose addresses must be taken. The caller is unaware of this space.

Return Values

Functions return results in register `r0`, except for **struct**s, which are stored at an address passed to the function as a hidden first argument.

- A double-word result (**double** or **long double**) is returned in registers `r0` to `r1`.
- Sixty-four-bit integers (**long long**s) are returned in registers `r0` to `r1`.

Prolog and Epilog Code

A function's prolog and epilog code establishes the environment needed by the body of the function.

- The prolog establishes a stack frame, if necessary, and can save any non-volatile registers the function uses.
- The epilog generally restores registers that were saved in the prolog code, restores the previous stack frame, and returns to the caller.

The following sections provide examples of prolog and epilog code.

Prolog Code Examples

In each of the prolog code examples in this section, the variable components (*framesize* and *btsize*) have the following meanings:

- *framesize* is the size, in bytes, of the area needed for auto variables, spill temporaries, saved registers, and possible back-trace data structure.
- *btsize* is the size of the stack back-trace data structure, which is 16 bytes.

Standard Prolog Code (ARCtangent-A4)

Standard prolog code performs the following tasks, in this order:

1. Saves the return address (`blink`) register in the back-trace data structure.
2. Saves the caller's frame pointer in the back-trace data structure provided by the caller.
3. Initializes the frame pointer to reference the caller's stack pointer.
4. Allocates the frame by decrementing the stack pointer *framesize* bytes to create space for auto variables, register spill area, and back-trace data structure (if required for further calls).
5. Saves any non-volatile registers that are modified by the function.

This is the standard prolog code:

```
; Save return address register:
        st      %blink, [%sp, 4]
```

```
; Save caller's frame pointer:
        st      %fp, [%sp]
; Set new frame pointer:
        mov     %fp, %sp
; Allocate frame:
        sub     %sp, %sp, framesize
; Save registers r13, r14, r15, and so
;  on (all non-volatile registers that must be saved):
        st      %r13, [%sp, btsize]
        st      %r14, [%sp, btsize+4]
        st      %r15, [%sp, btsize+8]
        . . .
```

Standard prolog code (ARCtangent-A5 and later)

Standard prolog code performs the following tasks, in this order:

1. Saves the return address (`blink`) register on the stack.
2. Saves any non-volatile registers that are modified by the function.
3. Allocates any additional space required in the frame by decrementing the stack pointer accordingly.

This is the standard prolog code:

```
; Save return address register:
    push_s   %blink
; Save registers r13, r14, r15, and so on
; (all non-volatile registers that must be saved):
    push_s   %r13
    push_s   %r14
    push_s   %r15
: Allocate remainder of frame
    sub      %sp, %sp, additional_size
```

Prolog Code for Argument on Stack (ARCtangent-A4)

A function's incoming argument(s) must be stored to memory if both of the following conditions are true for a given argument:

- The function takes the address of the argument.
- That argument is passed in a register.

If both conditions are true, the register arguments that must be stored are placed adjacent to any incoming arguments already on the stack, above the back-trace data structure.

To accomplish this placement, the prolog code first adjusts the current stack pointer to make space for the registers to be stored. The adjusted `sp` is then used, as in a standard prolog, to point to the back-trace data structure. (See the caller's `sp` in Figure 8 Stack Frame for one Function Invocation (ARCtangent-A4) on page 278.)

Here is the prolog code for a function in which the address of the first argument is taken:

```
; Adjust stack pointer to make space:
        sub     %sp, %sp, 4
; Save return address register and
;  caller's frame pointer, then
;  initialize new frame pointer:
        st      %blink, [%sp, 4]
        st      %fp, [%sp]
        mov     %fp, %sp
```

```
; Allocate frame:
        sub     %sp, %sp, 24
; Save registers r13, r14, r15, and so on
;  (all non-volatile registers that must be saved):
        st      %r13, [%sp, btsize]
        st      %r14, [%sp, btsize+4]
        st      %r15, [%sp, btsize+8]
;       . . .
; Store first argument above the stack
;  back-trace data structure
        st      %r0, [%fp, btsize]
```

Vararg functions (that is, functions with variable argument lists) are treated as if all incoming arguments have their address taken. Therefore, for vararg functions, all argument registers are stored to memory.

Prolog code for Argument on Stack (ARCtangent-A5 and Later)

A function's incoming argument(s) must be stored to memory if both of the following conditions are true for a given argument:

- The function takes the address of the argument.
- That argument is passed in a register.

If both conditions are true, the register arguments that must be stored are placed adjacent to any incoming arguments already on the stack.

To accomplish this placement, the prolog code first adjusts the current stack pointer to make space for the registers to be stored. The adjusted `sp` is then used as in a standard prolog (see the caller's `sp` in Figure 10 Stack Frame for One Function Invocation (ARCtangent-A5 and Later) on page 279.)

Here is the code for a function in which the address of the first argument is taken, assuming 24 bytes of additional stack space are needed:

```
; Adjust stack pointer to make space:
        sub_s    %sp,%sp,4
; Save return address register:
        push_s   %blink
; Save registers r13, r14, r15 and so on:
; (all non-volatile registers that must be saved):
        push_s   %r13
        push_s   %r14
        push_s   %r15
: Allocate remainder of frame:
        sub_s    %sp, %sp, 24

; Store the first argument "under" saved blink reg:
        st_s     %r0,[%sp, 40]
```

Vararg functions (that is, functions with variable argument lists) are treated as if all incoming arguments have their address taken. Therefore, for vararg functions, all argument registers are stored to memory.

Epilog Code Examples

In each of the epilog code examples in this section, the variable components (*framesize* and *btsize*) have the same meanings as defined in Prolog Code Examples on page 284.

Standard Epilog Code (ARCtangent-A4)

Standard epilog code performs the following tasks:

1. Restores all saved non-volatile registers.
2. Restores the return address register (`blink`).
3. Restores the caller's frame pointer register (`fp`).
4. Restores the caller's stack pointer register (`sp`).
5. Returns to the caller through the `blink` register.

This is the standard epilog code:

```
; Restore all saved non-volatile registers:
      ld      %r13, [%sp, btsize]
      ld      %r14, [%sp, btsize+4]
;     . . .
; Restore the return-address register:
      ld      %blink, [%fp, 4]
; Return to the caller through blink reg:
      j.d     [%blink]
; Restore the caller's frame-pointer register
;  and stack-pointer register:
      ld.a    %fp, [%sp, framesize]
```

NOTE If *framesize* is large enough to require a long immediate instruction encoding, the final **ld** instruction cannot be placed in the delay slot of the return jump.

Standard Epilog Code (ARCtangent-A5 and Later)

Standard epilog code performs the following tasks:

1. Frees additional space used in the stack frame by incrementing the stack pointer.
2. Restores all saved non-volatile registers.
3. Restores the return-address register (`blink`).
4. Returns to the caller through the `blink` register.

This is the standard epilog code:

```
; Free additional stack space
      add      %sp, %sp, additional_size

; Restore all saved non-volatile registers
      pop_s    %r15
      pop_s    %r14
      pop_s    %r13

; Restore the return address (blink register)
      pop_s    %blink

; Return to caller through blink register
      j_s      [%blink]
```

Epilog code for Argument on Stack (ARCtangent-A4)

Here is the epilog code for a function in which one incoming argument is passed in a register and has its address taken:

```
; Restore saved non-volatile registers:
      ld      %r13, [%sp, 16]
      ld      %r14, [%sp, 20]
;     . . .
; Restore the return-address register:
      ld      %blink, [%fp, 4]
; Restore the caller's frame-pointer register
;  and stack-pointer register:
      ld.a    %fp, [%sp, 24]
; Return to the caller through blink register:
      j.d     [%blink]
; Before returning, adjust stack pointer to
;  compensate for adjustment made in prolog:
      add     %sp, %sp, 4
;     . . .
```

Epilog code for Argument on Stack (ARCtangent-A5 and Later)

Here is the epilog code for a function in which one incoming argument is passed in a register and has its address taken, assuming 24 bytes of additional stack space is needed:

```
; Restore stack pointer to beginning of
; saved register area
      add_s    %sp, %sp, 24

; Restore saved non-volatile registers
      pop_s    %r15
      pop_s    %r14
      pop_s    %r13

; Restore the return address register blink
      pop_s    %blink

; Reclaim stack space for saved parameter
      add_s    %sp, %sp, 4

; Return to the caller through blink register
      j_s      [%blink]
```

Prolog and Epilog Variations

If a function is a leaf function (one that does not call other functions), the compiler often can simplify the prolog and epilog.

ARCtangent-A4

In the simplest case, the prolog requires no code whatsoever, and the epilog consists of just the instruction to return to the caller:

```
j       [%blink]
```

If the function requires a stack frame for auto variables, the prolog consists of a single instruction to allocate the frame:

`sub %sp, %sp,` *framesize*

while the accompanying epilog returns to the caller and restores the caller's stack-pointer register:

```
j.d     [%blink]
add     %sp, %sp, framesize
```

In this case the auto variables are accessed by way of the `sp` register.

ARCtangent-A5 and Later

In the simplest case, the prolog requires no code whatsoever, and the epilog consists of just the instruction to return to the caller:

```
j_s     [%blink]
```

If the function requires a stack frame for auto variables, the prolog consists of a single instruction to allocate the frame:

```
sub     %sp, %sp, framesize
```

while the accompanying epilog returns to the caller and restores the caller's stack-pointer register:

```
add     %sp, %sp, framesize
j_s     [%blink]
```

Other Possible Variations

The compiler can undertake other simplifications as well. For example, if a function is a leaf function but is complex enough to require a frame pointer, the standard prolog and epilog sequences are performed but `blink` is not saved and restored.

Space Optimization

When optimizing for space, the compiler might call run-time subroutines to save and restore registers. This results in smaller code at the expense of the overhead of calling a subroutine. The customized subroutines for this purpose are referred to as "millicode" routines.

ARCtangent-A4

Here is a prolog that uses millicode:

```
st    %blink, [%sp, 4]   ; Save return address
bl.d  __prolog_save17sp  ; Save registers 13-17 and
                         ; Allocate frame
mov %r12, 60             ; amount to decrement stack

; The value 60 is based on 20 bytes for register
; save, 16 bytes for the back-trace data structure
; (as this is not a leaf function), and 24 bytes
; of additional stack space required.
```

Here is an epilog that uses millicode:

```
b.d   __epilog_load17
mov   %r12, 60
```

ARCtangent-A5 and Later

Here is a prolog that uses millicode:

```
push_s    %blink              ; Save return address
bl        __ac_push_13_to_17  ; Save registers 13-17
sub_s     %sp, %sp, 24        ; Allocate any
                              ; additional stack space
```

Here is an epilog that uses millicode:

```
add_s     %sp, %sp, 24        ; Free allocated
                              ; stack space
```

```
b         __ac_pop_13_to_17   ; Pop registers 13-
                              ; 17 and blink, return
```

Dynamic Stack Allocation

C and C++ allow you to dynamically grow the current stack frame using the intrinsic **alloca()** function:

```
#include <alloca.h>
extern void *alloca(size_t);
```

Implementations of **alloca()** must maintain a frame pointer and the stack mechanics outlined in Stack Conventions on page 276 through Prolog and Epilog Code on page 284. The stack frame is maintained using the frame pointer (`%fp`) instead of the stack pointer (`%sp`), as in the following example:

ARCtangent-A4

```
;  alloca(64)
mov      %r0, 64
sub      %sp, %sp, %r0
add      %r0, %sp, 16; back-trace data structure
```

ARCtangent-A5 and Later

```
push_s   %blink
...
push     %fp
mov      %fp, %sp
       ;alloca(64)
mov_s    %r0, 64
sub      %sp, %sp, %r0
...
mov      %sp, %fp
pop      %fp
...
pop_s    %blink
j_s      [%blink]
```

Chapter 14 — Diagnostics

In This Chapter

- Overview
- Generating Debugging Information
- Reducing Debugging Information
- Tracing Function Calls
- Execution Profiling

Overview

This chapter describes how to compile your application for analysis using the debugging and diagnostic tools.

Generating Debugging Information

Driver option **-g** directs the compiler to generate debugging information.

Reducing Debugging Information

You can substantially reduce the amount of debugging information in an executable by using option **-g0** or **-g1**. Option **-g0** turns on toggle `Emit_line_records`, which generates only line-number debugging information. This allows you to debug a program line by line but does not provide symbolic debugging information. Option **-g1** provides more information, but permits reliable source-level examination of data only at function entry and exit points.

Tracing Function Calls

The MetaWare compiler can generate code to trace entry to, exit from, and calls to functions in your application. Each time one of these events occurs, the appropriate trace function is called. Simple versions of these trace functions are included in the library source directory, in file `trace.c`.

You can customize the functions to meet your particular needs. Within a trace function you can, for example, print the name of the function entered, exited, or called, or observe the function-call behavior on a particular set of inputs. Before you can use the functions, you must compile and include them.

One use of the tracing facility is to help you understand code you have not written yourself.

Enabling Function-Call Tracing

To enable function-call tracing, compile your code with the following toggles turned on:

- Toggle `Call_trace` calls function **_call_trace()** just before a function is called.
- Toggle `Prolog_trace` calls function **_prolog_trace()** when execution enters a function.
- Toggle `Epilog_trace` calls function **_epilog_trace()** when execution exits a function.

The tracing functions have the following signatures:

```
_call_trace  (const char *from_proc,
              const char *from_module,
              unsigned    line,
              const char *to_proc);
_prolog_trace(const char *func);
_epilog_trace(const char *func);
```

The parameters of these functions represent the following values:

func	The name of the function entered or exited
from_proc	The name of the function making the call

from_module	The source module from which the call is made
line	The line of the source module where the call is made
to_proc	The name of the function called

Execution Profiling

The toolset provides several ways to analyze the run-time performance of your executables:

- Branch prediction, described in Appendix E — Branch Prediction
- Statistical profiling, described in Using Statistical Profiling

If you discover a bottleneck in your code, consider using one of the functions described in Conditional-Jump Minimization on page 175.

Using Statistical Profiling

When building an application you want to profile, you must consider the following:

- Compiling and Linking Options on page 293
- Providing Run-Time Instrumentation Code on page 294
- Configuring Profiling Parameters on page 294

After you have compiled and linked the executable, you can view the statistical profile using the MetaWare debugger or the the GUI-based integrated profiling tool from ARC International. See the *User's Guide* for each of those tools for details.

For information about using the command-line profiling utilities **profarc** and **profac**, see the ELF Linker and Utilities User's Guide. However, we recommend using the MetaWare debugger or the GUI-based integrated profiling tool from ARC International.

Compiling and Linking Options

You can compile and link your executable to include information for statistical profiling with or without call-graph information.

- To produce an executable with statistical-data information but without call-graph information, compile and link using option **-p**.
- To produce an executable with both statistical-data information and call-graph information, compile and link using option **-pg**.
- To profile using a timer extension, as on the ARCangel development board or equivalent, you must also link using option **-Xtimer0** or **-Xtimer1**. For more information, see Timer Extension Support on page 242.
- For profiling on the MetaWare debugger simulator, we recommend you not use option **-Xtimer0** or **-Xtimer1** for timing. Instead, profile using the MetaWare debugger profiling feature, and use the simulator's instruction counts for profiling. For details, see the *MetaWare Debugger User's Guide*.

CAUTION Executables compiled for profiling might require a sizeable heap — between 50% and 100% of the size of the executable, depending on the granularity and range of the

sampling. The default hostlink run-time support provides this heap allocation. For more information, see Using a Heap on page 213.

Providing Run-Time Instrumentation Code

We provide full implementations of the MetaWare C Run-Time Library to support profiling for popular targets. Because this instrumentation code is specific to a target and RTOS, you might need to perform some configuration work. For details of the functions you must provide, see Providing Support for Statistical Profiling on page 242.

Configuring Profiling Parameters

You can configure the profiling parameters to strike a balance between accuracy, memory use, and risk of counter overflow while obtaining the granularity and focus you need for your specific application.

Profiling maps every executable address in your program to a counter and increments the counter whenever a profiling interrupt occurs while execution is at the executable address associated with that counter. Each counter spans eight bytes of machine instructions.

NOTE If you are using the MetaWare debugger, you can set the following parameters from the debugger without modifying the profiling source. For more information, see the *MetaWare Debugger User's Guide*.

To change the profiling source manually, make your changes to *install_dir*`/inc/mon.h`. and recompile *install_dir*`/lib/src/c/misc_g/profrt.c`.

User-Configurable Profiling Parameters

You can configure the following profiling parameters:

- Interrupt frequency

  A higher frequency produces a more accurate profile but slower execution.

- Total range of `pc` addresses being profiled

  By default, profiling analyzes the entire `.text` section of your executable. You can reduce this range to reduce the memory usage by the counters or to focus on an area of interest in your program. To modify the range, substitute your desired range for `_ftext` and `_etext` in *install_dir*`/src/c/misc_g/profrt.c`.

- Granularity of profiling (range of `pc` addresses per counter)

  By default, each counter spans an 8-byte range of `pc` addresses and counts each occurrence of an interrupt while execution is at an address within that range. Reducing the range of addresses (finer granularity) yields a more accurate profile, but at the cost of increased memory usage. Finer granularity also reduces the risk that a counter might overflow. Coarser granularity (more addresses per counter) uses less memory but reduces accuracy and increases the risk of counter overflow.

  Granularity is controlled by the constant **BYTES_PER_COUNT** (in *install_dir*`/inc/mon.h`). More **BYTES_PER_COUNT** requires less memory.

- Size of counters

  Profiling uses 16-bit counters by default. Increasing their size to 32 bits uses more memory but reduces the chance that a counter might overflow.

  The amount of memory required is calculated as follows:

```
(counter size) * (code size) / BYTES_PER_COUNT
```

The counter size is controlled by the type of **WORD** in *install_dir*/`inc/mon.h`.

Chapter 15 — Generating Listings

In This Appendix

- Overview
- Customizing the Listing with Pragmas
- Listing Format

Overview

This appendix describes the source listings the compiler can generate and explains how you can control the format of the listings.

To get a listing of your source code, specify option **-Hlist** on the command line, or turn on toggle `List`. The source listing goes to standard output unless you specify a different destination as an argument to **-Hlist** on the command line.

You can use pragmas `Page`, `Skip`, and `Title`, respectively, to control the number of lines per page, insert blank lines in the listing, and specify a title for your listing.

Customizing the Listing with Pragmas

To cause *n* page ejects at some point in the listing, insert this line:

`#pragma Page(`*n*`)`

n is the number of form feeds, or the number of pages to eject. Ordinarily you would set *n* to 1 to get a page break.

To generate *n* blank lines at some point in the listing, insert this line:

`#pragma Skip(`*n*`)`

n is the number of blank lines.

To get a title at the top of each successive page, place the following pragma in the source:

`#pragma Title("`*Listing_title*`")`

Listing_title is the title to be printed.

Each successive `Title` pragma changes the title for subsequent pages. The only way to title the first page is to place the `Title` pragma in the `.pro` file and use **#pragma On(List)** at the end of the `.pro` file, or wherever the listing should start in the source file.

When you request a listing by specifying option **-Hlist** on the command line, the first page is not titled, because the listing starts before the compiler sees the `Title` pragma.

Listing Format

Ruler

The first line on each page, after any header and title lines, is a ruler that appears as follows:

```
Levels LINE#|----+----1----+----2----+----3----+----4----+----5--+
```

This ruler defines three fields:

- the level numbers
- the line number
- the line contents

Level Numbers

Use level numbers to find a missing "}" or comment terminator when the compiler produces a message such as "Unexpected end-of-file." All three level numbers are initially 0 (zero), but they print as blanks rather than as a 0 (zero).

- The first level number indicates the scope nesting level for **struct** or **union** declarations.
- The second level number indicates the statement nesting level. It is incremented at the beginning of each "{" and decremented at the corresponding "}".
- The third level number indicates the structure initialization nesting level. As with the second level number, it is incremented at the beginning of each "{" and decremented at the corresponding "}".

Include Files

A first-level include file named *File_name* is shown as starting after a line with "+(File_name" in the line-number field, and ending just before a matching "+)File_name" line.

The included lines have "+" in the leftmost column of the line-number field, and those lines are numbered independently of the main source file.

An included file inside an include file has an extra "+" on each line for each level of inclusion, except that line numbers take precedence over "+" characters in the line-number field if the "+" characters would otherwise intrude into that field.

The .pro File

The `.pro` file, if any, is listed as an include file. See Using .pro Files on page 39 for more information.

Appendix A — Intrinsic Functions

In This Appendix

- Overview
- Compiler-Intrinsic-Function Reference
- Calling an Equivalent Library Function
- Defining Intrinsics for Extensions Using C/C++
- Using Intrinsics to Access Auxiliary Registers

Overview

Compiler-intrinsic functions are functions recognized by the compiler without the presence of a prototype. Because intrinsics are essentially language extensions recognized by the compiler, the compiler can fully optimize around them, generating more efficient code than if function calls or inline assembly were used.

f you provide a prototype for an intrinsic, you must also provide code to execute it (because intrinsics are not in the library).

NOTE Do not confuse intrinsics with ANSI Standard C functions that the compiler recognizes and might inline. For information about the recognized ANSI Standard C functions, see Inlining Certain ANSI Standard C Functions on page 178.

Compiler-Intrinsic-Function Reference

This section provides a prototype and basic information on compiler-intrinsic functions. It refers to the *MetaWare C Library Reference* in the case of functions that are also present in the library.

Alphabetical Listing of Intrinsics

This section lists all MetaWare C/C++ compiler-intrinsic functions in alphabetical order by function name (preceded by return type where applicable).

_abs()

See the *MetaWare C Library Reference* for details.

NOTE The return type for this function is based on the types of the arguments, so it can be any scalar or floating point type.

void * _Alloca()

Intrinsic version of **alloca()** with no header file required. See **alloca()** in the *MetaWare C Library Reference* for details.

void _ASM()

See Inline Assembly Code on page 191 for details.

void _brk()

Insert a breakpoint instruction (**BRK**) into the generated code for ARCtangent-A5 or later or forARCtangent-A4 core version 7 or later.

NOTE To use this intrinsic on processors previous to ARCtangent-A5, you must compile using option **-core7** or **-core8**.

int __builtin_constant_p(*value*)

Use **__builtin_constant_p()** to determine if *value* is known to be constant at compile time so that the compiler can perform the constant-folding optimization on expressions involving that value. This can

be useful when memory is a critical resource; you can let a complex calculation be folded if it involves constants or call a function if it does not.

This function returns `1` if the argument is known to be a compile-time constant and `0` if the argument is not known to be a compile-time constant. A return of `0` does not necessarily indicate that the argument is not a constant; it indicates only that the compiler cannot prove the argument is a constant.

You can use this function in macros, but not in **inline** functions, because inlining occurs after evaluation of **__builtin_constant_p()**.

unsigned _condition(unsigned *cond*)

Test an extension condition code for true or false. Argument *cond* is the numeric value of the extension condition code to test and must be a numeric literal constant previously defined using pragma `Condition`. **_condition()** returns 1 (one) if *cond* is true, and 0 (zero) if *cond* is false.

By default, when you use **_condition()**, the compiler emits the required directives to define the condition codes to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle `Emit_core_directives`.

For more information, see Using Extension Condition Codes in C/C++ on page 166.

unsigned _core_read(unsigned *regno*)

Returns the content of the core register numbered *regno* (note that *regno* must be a compile-time constant). Using this intrinsic is the most precise way to guarantee that exactly one reference is made to the core register.

> **CAUTION** If read or write operations on your core register have side effects and you access the core register from C, be sure to use pragma `Core_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

By default, when you use **_core_read()** to access a register, the compiler emits the required directives to define the extension core registers to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle `Emit_core_directives`.

void _core_write(unsigned *val*, unsigned *regno*)

Writes the value *val* to the core register numbered *regno* (note that *regno* must be a compile-time constant). Using this intrinsic is the most precise way to guarantee that exactly one update is made to the core register.

> **CAUTION** If read or write operations on your core register have side effects and you access the core register from C, be sure to use pragma `Core_register` to inform the compiler that side effects occur. By default, the compiler assumes that no side effects occur. Results might not be what you expect if you do not inform compiler that side effects occur.

By default, when you use **_core_write()**to access a register, the compiler emits the required directives to define the extension core registers to the assembler. If you prefer to supply the directives in an assembly file using option **-Hextfile**, you must also turn off toggle `Emit_core_directives`.

void _disable()

Mask interrupt levels 1 and 2 without altering other flags.

void _disable1()
Mask interrupt level 1 without altering other flags.

void _disable2()
Mask interrupt level 2 without altering other flags.

void _enable()
Unmask interrupt levels 1 and 2 without altering other flags.

void _enable1()
Unmask interrupt level 1 without altering other flags.

void _enable2()
Unmask interrupt level 2 without altering other flags.

void _flag(unsigned *val*)
Insert a flag-setting instruction (**FLAG**) into the generated code, where *val* contains the flag values to set and is also the operand to the **FLAG** instruction.

> **NOTE** The format of the flag instruction is different for ARCtangent-A4 and for later processor series.

long * _interrupt_ext_addr()
When used with pragma `Interrupt_ext_words`, returns the address of the portion of the interrupt frame reserved for user-defined extension state.

You can use pragma `Interrupt_ext_words` to provide extra space in the interrupt frame to save extension state. For more information, see Providing Space for Extensions to Save State on page 226.

unsigned _lr(unsigned *addr*)
Loads the content of auxiliary register address *addr* using the **LR** instruction.

_max()
See the *MetaWare C Library Reference* for details.

> **NOTE** The return type for this function is based on the types of the arguments, so it can be any scalar or floating point type.

_min()
See the *MetaWare C Library Reference* for details.

> **NOTE** The return type for this function is based on the types of the arguments, so it can be any scalar or floating point type.

void _nop()

Each call to **_nop()** produces a single **NOP** instruction in the compiled code. You can use **_nop()** with extension instructions and registers that require a mandatory delay of some number of cycles.

void overlay_copy(*char *target*, int *bytes, var*)

Copy overlay data item *var* of size *bytes* to address *target*. For more information on using this function, see the *Automated Overlay Manager User's Guide*.

void _overlay_free(void *)

Free the locked data item returned by a call to **_overlay_load()** or dynamic overlay data allocated using **_overlay_malloc()**. For more information, see the *Automated Overlay Manager User's Guide*.

*** _overlay_load(**overlayVar*)**

Load a data item qualified with **_Overlay()**. For more information, see the *Automated Overlay Manager User's Guide*.

void * _overlay_malloc(int *size*)

Allocate dynamic overlay data. For more information, see the *Automated Overlay Manager User's Guide*.

void _prefetch(void *)

ARCtangent-A5 and Later

This intrinsic allows you to explicitly tell the compiler where and when you want a pre-fetch instruction generated to start filling a cache line. The compiler does not move the pre-fetch instruction during delay-slot filling or instruction scheduling, nor does it allow other instructions to move past or below the pre-fetch instruction.

***Example 60 Using the* _prefetch() intrinsic to generate a pre-fetch instruction**

```
// prime the cache for the first iteration
_prefetch(x);
_prefetch(y);
for (int i = 0; i < n; i += 8) {
        // prime the cache for the next iteration
        _prefetch(x+8);
        _prefetch(y+8);
        // use primed cache(s) for first iteration
        x[i] = y[i] * z;
        x[i+1] = y[i+1] * z;
        x[i+2] = y[i+2] * z;
        x[i+3] = y[i+3] * z;
        x[i+4] = y[i+4] * z;
        x[i+5] = y[i+5] * z;
        x[i+6] = y[i+6] * z;
        x[i+7] = y[i+7] * z;
        }
}
```

void _sleep()

Insert a sleep instruction (**SLEEP**) into the generated code for processors that support the **SLEEP** instruction.

NOTE To use this intrinsic on processors previous to ARCtangent-A5, you must compile using option **-core7** or **-core8**.

void _sr(unsigned *val*, unsigned *addr*)
Stores *val* in auxiliary register address *addr* using the **SR** instruction.

short _swap16(short)
Reverses the byte order of the 16-bit operand, reversing the endianness of the value.

int _swap32(int)
Reverses the byte order of the 32-bit operand, reversing the endianness of the value.

void _swi()
Each call to **_swi()** generates one software interrupt instruction (**SWI**) for processors that support the **SWI** instruction.

NOTE To use this intrinsic on processors previous to ARCtangent-A5, you must compile using option **-core8**.

Calling an Equivalent Library Function

If you want to call an equivalent library function instead of an intrinsic, you must provide an external declaration. For example, the following declaration disables the **_abs** compiler-intrinsic function.

```
extern int _abs(int);
```

Defining Intrinsics for Extensions Using C/C++

Pragma `Intrinsic` lets you define an intrinsic to generate an extension instruction outside of the set of extensions known internally to the compiler.

Functions that use instruction intrinsics are better-optimized than functions that use inline assembly. The benefits of using instruction intrinsics include the following:

- The compiler knows all registers referenced and modified by the instruction. With inline assembly the compiler must make pessimistic assumptions.
- The generated instruction can be moved into a delay slot. Inline assembly restricts the effectiveness of delay-slot filling.
- The generated instruction can be made to be conditionally executed.
- The generated instruction can be made to set the flags to eliminate comparison with zero.

For a description of this feature, see pragma `Intrinsic` in Alphabetical List of Pragmas on page 129 and Using and Optimizing Extensions in C/C++ on page 164.

Using Intrinsics to Access Auxiliary Registers

You can use intrinsics **_lr()** and **_sr()** to read and write, respectively, the contents of an auxiliary register.

The compiler does not support mapping variables and **struct**s into auxiliary register space. However, you can access individual locations directly using the **_lr()** and **_sr()** intrinsics. If you want to map a **struct** to auxiliary register space, you can write macros to automate the member calculations, as shown in Example 61 A Hidden Use of Intrinsics to Access Auxiliary Registers.

Example 61 A Hidden Use of Intrinsics to Access Auxiliary Registers

MetaWare C/C++

```
#include <stddef.h>
#define AUX_BASE 0x100000
typedef struct {
   unsigned first;
   unsigned second;
   unsigned third;
   unsigned fourth;
}  aux_mapped_data;

#define get_data(field) \
     (_lr(AUX_BASE +  offsetof(aux_mapped_data,field)/4))
#define set_data(field, val) \
     (_sr((val), AUX_BASE + offsetof(aux_mapped_data,field)/4))

main() {
   set_data(fourth, get_data(second));
   set_data(first, 0);  }
```

Assembly

```
main:
      lr %r0, [0x100001]
      sr %r0, [0x100003]
      sr 0x0, [0x100000]
      j [%blink]
```

The C code in Example 61 A Hidden Use of Intrinsics to Access Auxiliary Registers does not declare any data, but the **typedef** defines the layout of the data at a particular register address so that you can access it based on that layout.

Appendix B — Configuring the Driver

In This Appendix

- Overview
- Modifying Configuration Variables

Overview

This appendix describes how to modify the MetaWare C/C++ driver configuration file.

You can change the default values of many compiler controls by modifying the user-modifiable configuration file. You can edit this file to assign values to the variables shown in Modifying Configuration Variables on page 310.

Directions for modifying the configuration file appear as comments in the file itself. Consult those comments for additional information about user-modifiable values.

NOTE Before you modify the contents of the configuration file, we recommend you make a copy in a separate directory.

For additional information on using the driver configuration file, including default file names, see Using the Driver Configuration File on page 38.

Modifying Configuration Variables

This section documents some important variables in the user-modifiable configuration file. This list is not exhaustive. See the configuration file provided with your distribution for additional variables and default values.

In the driver configuration file, the character '#' is a comment delimiter, that is, any line that begins with a '#' is a comment. You must remove comment delimiters to activate the assignments described in this section. For example, to set a value for *HCDIR*, you must delete the '#' from the line that starts with `#HCDIR=`.

CAUTION If you modify the user-modifiable configuration file beyond making the changes described here, you do so at your own risk. Generally, we do not support extensive customer modifications to the configuration file. We are supporting your use of the MetaWare C/C++ compiler, not of the driver language.

Alphabetical List of User-Modifiable Variables

This section provides an alphabetical list of all user-modifiable configuration variables.

ARGS — Specify defaults for command-line arguments

Some operating systems place a limit on the length of the command line. You can preset a longer string of options by specifying them as parameters to `ARGS`, and the driver will process them as though you had specified them on the command line.

If there are options you always use, you can specify their values with `ARGS` so they take effect in all compilations. For example, any toggle can be given a default value with the `ARGS` variable, as shown here:

```
ARGS= -Hon=Pointers_compatible
```

NOTE Because these ARGS-specified options are invisible on the actual command line, it is easy to forget about them. To display the options you have set, specify option **-V** on the command line.

AS — Assembler variable

Variable `AS` is for assembling. The driver assumes this variable is defined. Within the definition, you specify the input source (`%s`) and the assembly output (`%o`). The driver provides the particular file names in the positions denoted by the `%` symbols.

CEXT — Specify C source file-name extension

Variable `CEXT` specifies the default file-name extension for C source files (usually `.c`). For more information, see Recognized Input File Types on page 27.

CPLUS — Specify C++ compilation

For C++ only

Variable `CPLUS` tells the compiler you intend to compile and link C++ files. When `CPLUS` is set to `1` (one), the MetaWare C++ header files and run-time library are included. `CPLUS` can also be set as an environment variable. See Compiling and Linking C++ Modules on page 30.

CPP — C preprocessor variable

For UNIX and Linux hosts only

Variable `CPP` specifies the UNIX and Linux C preprocessor, **cpp**. If this variable is defined, the driver invokes **cpp** as the preprocessor.

CPPEXT — Specify C++ source file-name extensions

For C++ only

Variable `CPPEXT` specifies the default file-name extensions for C++ source files. The default extensions are usually `.C` (UNIX and Linux hosts), `.cpp`, and `.cc`. You can also specify the default extension with option **-Hcppext** (see General Driver Options on page 52).

HCDIR — Specify MetaWare C/C++ installation directory name

Variable `HCDIR` specifies the parent directory where you initiated the installation.

For UNIX and Linux hosts only

You can also set `HCDIR` as an environment variable.

LINKER — Linker variable

Variable `LINKER` is for linking. The driver assumes this variable is defined. Within the definition, you specify the input objects (`%o`), the optional output load file (`%O`), the libraries, and the startup file. The driver provides the particular file names in the positions denoted by the '%' symbols.

TMPPREFIX — Specify alternate directory for temporary files

On UNIX and Linux hosts, temporary files required by the compiler are ordinarily created in a default directory, such as `/tmp`. On Windows hosts, the temporary files are usually created in your current working directory. To designate an alternate directory for temporary files, set the `TMPPREFIX` variable in the configuration file. For example:

```
TMPPREFIX= /cppfiles/my_temp
```

You can also set `TMPPREFIX` as an environment variable; for more information, see Setting an Alternate Directory for Temporary Files on page 35.

TOOLSDIR — Specify path to assembler and linker

Variable TOOLSDIR specifies the directory where your assembler and linker reside. (`TOOLSDIR` can also be set as an environment variable). See the *Getting Started* manual for details.

Appendix C — Calling Conventions

In This Appendix

- Overview
- The Calling-Convention Pragma
- The Calling-Convention Type Qualifier
- Calling-Convention Attributes
- Predefined Calling-Convention Names
- Replacing One Function with Another

Overview

This chapter describes the interface between a calling function and a called function.

A *calling convention* is the interface between a calling function and a called function: a communication protocol. MetaWare C/C++ compilers provide a way to specify a variety of calling conventions, so you can link MetaWare C/C++ code with programs written in languages with different protocols, such as Pascal and FORTRAN.

You can change the following aspects of communication between functions:

arguments	the order in which arguments are pushed on the stack, whether the caller or callee pops them off the stack, and whether registers are used to pass arguments
function address	whether the address is near or far
interrupts	whether the function being called is an interrupt handler

The code generated for a call to a function is the *calling sequence*. Choosing a particular calling convention that both the caller and the callee understand makes the calling sequence communicate correctly with the called function's prolog and epilog.

- The *prolog* is the code executed immediately after the call to set up the environment for the called function to run in.
- The *epilog* is the code executed immediately before the called function returns.

Calling-sequence code is in the caller, while prolog and epilog code are in the callee.

Different compilers and languages map data in different ways. Communicating data properly between functions written in different languages is the programmer's responsibility.

Using the __attribute__ Syntax

MetaWare C/C++ supports an **__attribute__** syntax similar to that of GNU that can be used with some of the calling-convention attributes listed in this chapter; see the *MetaWare C/C++ Language Reference* for details.

The Calling-Convention Pragma

The `Calling_convention` pragma provides another way to specify a calling convention.

This pragma has three forms:

- #pragma Calling_convention(*Attributes*)
- #pragma Calling_convention(*Attributes*, _DEFAULT)
- #pragma Calling_convention()

Attributes is a list of one or more calling-convention attributes concatenated using bitwise operators (**&**, |, and ~) with their usual meanings. See Calling-Convention Attributes on page 315 for information about constructing an *Attributes* string.

The first form of pragma `Calling_convention` sets the current calling convention.

The second form of pragma `Calling_convention` sets the current calling convention and also sets the default calling convention.

NOTE **_DEFAULT** must be spelled exactly as shown, in all uppercase.

The third form of pragma `Calling_convention` restores the calling convention to whatever it was prior to the last time it was set.

Before setting the calling convention, the first and second forms push the convention then in force on a stack; the third form pops the stack and sets the current calling convention to the popped value. If the stack is empty when execution encounters the third form of this pragma (as would be the case if the first use of **#pragma** `Calling_convention` had no argument), the calling convention is undefined.

The Calling-Convention Type Qualifier

The calling-convention type qualifier, **_CC**, is a MetaWare C/C++ extension for specifying the calling convention of a declared function or functionality type. It has this form:

`_CC(`*Attributes*`)`

where *Attributes* is a list of calling-convention attributes concatenated using bitwise operators (**&**, |, and ~) with their usual meanings. The use of this qualifier is typically hidden within a macro; see Example 62 Importing a Rarely Called Function That Never Returns.

Example 62 Importing a Rarely Called Function That Never Returns

```
#define RARE_NR _CC(_CALLED_INFREQUENTLY | _NEVER_RETURNS |
_DEFAULT_CALLING_CONVENTION)
extern RARE_NR void function1(int x);
```

Calling-Convention Attributes

To construct an *Attributes* string, use bitwise operators (**&**, |, and ~) with their usual meanings to combine the predefined attributes listed in this section.

Alphabetical Listing of Attributes

This section lists the calling-convention attributes in alphabetical order.

_CALLED_INFREQUENTLY — Designate function as one that is not often called

If a function is assigned the **_CALLED_INFREQUENTLY** attribute, and it is called in a conditional expression within a loop construct, the compiler does the following:

- reorders the nodes so the node containing the function call is moved physically out of the loop
- sets a jump from the node containing the function call back into the loop

For more information, see pragma `Called_infrequently` in Alphabetical List of Pragmas on page 129.

_FORCE_INLINE — Inline function regardless of inliner options

Inline the designated function regardless of optimization level or the settings of the outboard-inliner options **-Hic**=*N* and **-Hit**=*N*. Note that use of some language features and optimization levels prevents inlining in any case, for example:

- C++ exception handling
- **setjmp()**
- **alloca()**
- Recursion
- Compiling with low optimization levels or **-g**

Note that for C++, the **_CC** must be on the method declaration in the class, regardless of where the method is actually defined.

Example 63 Using _FORCE_INLINE with C++

```
class test {
    int _CC(_DEFAULT_CALLING_CONVENTION|_FORCE_INLINE) doit(void);
}
int test::doit(void) {
}
```

_INTERRUPT1 — Designate function as a level-one interrupt handler

Specifies that the function is a level-one interrupt handler that returns using register `ilink1`. This specification is required because a hardware interrupt uses a calling sequence different from that of a normal call: the machine's status is saved and restored across interrupts.

Macro **_Interrupt1** is provided for convenience. It expands to **_CC(_INTERRUPT1|_DEFAULT_CALLING_CONVENTION)**. The following is a usage example:

```
int _Interrupt1 func(void) {
    ...
    }
```

For more information, see Writing Interrupt Handlers on page 224.

_INTERRUPT2 — Designate function as a level-two interrupt handler

Specifies that the function is a level-two interrupt handler that returns using register `ilink2`. This specification is required because a hardware interrupt uses a calling sequence different from that of a normal call: the machine's status is saved and restored across interrupts.

Macro **_Interrupt2** is provided for convenience. It expands to **_CC(_INTERRUPT2|_DEFAULT_CALLING_CONVENTION)**. The following is a usage example:

```
int _Interrupt2 func(void) {
    ...
    }
```

For more information, see Writing Interrupt Handlers on page 224.

_NEVER_RETURNS — Designate function as one that does not return

If a function is assigned the **_NEVER_RETURNS** attribute, the compiler can improve register allocation, because no registers are modified across the call to this function.

> **NOTE** As far as the optimizer is concerned, for the **_NEVER_RETURNS** function currently being compiled, any call to a function that never returns does not "trash" any registers, due to the construction of the flow graph.

_NEVER_THROWS — Designate function as one that does not throw

For C++ Only

When the compiler encounters a function call within a try-block, it assumes that the function itself might throw an exception. This assumption causes the compiler to generate overly conservative code that assumes all machine registers can be modified.

Marking a function's calling conventions with **_NEVER_THROWS** tells the compiler that this function is guaranteed to return to the call site within the try-block. With this calling convention, the optimizer generates the normal ABI-compliant code for a call.

_PRESERVE_FLAGS — Designate function as one that preserves condition flags

Indicates that the function preserves condition flags. Knowing that a function does not modify condition flags when called helps the compiler optimize calls to that function, as when a call is part of a sequence of conditionally executed instructions.

Macro **_Preserve_flags** is provided for convenience. It expands to **_CC(_PRESERVE_FLAGS|_DEFAULT_CALLING_CONVENTION)**. The following is a usage example:

```
int _Preserve_flags func(void) {
    ...
    }
```

_SAVE_ALL_REGS — Save volatile and non-volatile registers used by a function

Causes the compiler to save all registers used by a function, both volatile and nonvolatile.

Macro **_Save_all_regs** is provided for convenience. It expands to **_CC(_SAVE_ALL_REGS|_DEFAULT_CALLING_CONVENTION)**. The following is a usage example:

```
int _Save_all_regs func(void) {
    ...
    }
```

Predefined Calling-Convention Names

MetaWare C/C++ provides two predefined calling-convention names, as follows:

_CALLING_CONVENTION	Holds the current calling-convention attributes
_DEFAULT_CALLING_CONVENTION	Holds the default calling-convention attributes to which the current calling convention is initially set

The calling convention affects all subsequent function declarations, whether or not the declaration is a definition.

Replacing One Function with Another

The MetaWare C/C++ toolset provides a method for “rewiring” (redirecting) a function so that a second function is called in its place during execution. This might be useful for handling bugs or enhancements. To rewire a function, proceed as follows:

1. Turn on toggle `Func_replacement_prolog` for the function (for information about turning on compiler toggles, see Chapter 5 — Using Compiler Toggles).
2. Overwrite the memory location at address `fixtable.f` in ELF section `.fixtable` to contain the new function, as shown in the following example.

```
        .section .fixtable, data
fixtable.f:
        .long   cont.f ; <= change to new func
                       ; (ARCtangent-A4 is "cont.f@h30")
        .previous
;-------------| f |-------------------
        .global f
f:
        ;--- begin func_replacement_prolog
        ld      %r12, [fixtable.f]
        j       [%r12]
cont.f:
        ;--- end func_replacement_prolog
```

Appendix D — The Heap Manager

In This Appendix

- Overview
- Allocated Memory
- Free Memory
- Integrity Checking

Overview

This appendix describes how the compiler manages the memory heap. The specifics of heap management are subject to change, but the general principles are presented here for help in debugging heap errors.

NOTE This appendix applies only if you are using the supplied (ANSI) library.

The *heap* is a region of memory used for dynamically allocated data structures, those whose size is not known at compile time and might change during execution.

An application can request (allocate) memory from the heap or return (free) previously allocated memory to the heap at any time during execution. Applications manipulate the heap using the C library functions **malloc()**, **realloc()**, and **free()**.

The data structures that reside in the heap come into existence, grow, shrink, and disappear in an unpredictable fashion, resulting in the following requirements:

- The operating system must be able to reclaim memory that is no longer being used by the application.
- The mechanism for organizing heap storage must be very flexible.

The heap is organized into a chain with each entry containing several bytes of information about the heap space it represents. Each block of memory in the heap is known to be either allocated or free, and the free blocks are doubly linked for maximum flexibility.

Allocated Memory

The heap manager organizes memory into a chain. Each allocated block of memory in the chain consists of three units of information (see Figure 11 Heap item):

- its size (four bytes)
- the size of the previous block (four bytes)
- the user data it contains (one-to-many bytes)

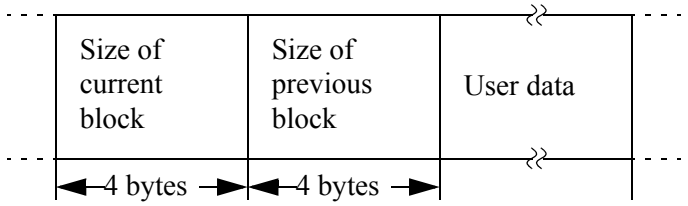

Figure 11 Heap item

Suppose the following heap activity occurs:

```
p1 = malloc (10);
p2 = malloc (10000);
p3 = malloc (512);
p4 = malloc (2048);
```

Then the heap has four entries on it:

- 10 bytes (allocated)
- 10,000 bytes (allocated)
- 512 bytes (allocated)
- 2,048 bytes (allocated)

Figure 12 Heap Chain After Four Allocations shows what the heap looks like after the four calls to **malloc()**. The first call to **malloc()** finds a chunk of memory in the free chain large enough to hold 20 bytes. This accounts for the four allocated blocks not being contiguous.

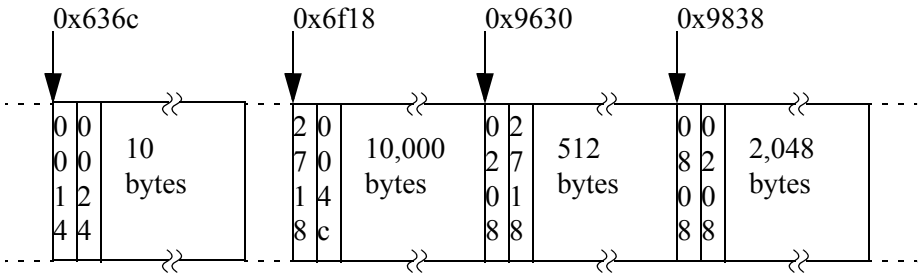

Figure 12 Heap Chain After Four Allocations

In tabular form, the heap contains the following allocated blocks:

```
Var  Addr     Size          Prev_Size   User Data
===  =======  ============  =========  ================
p1   0x6e6c:    14 (0xe)     0x24       starts @ 0x6374
p2   0x6f18:  2718 (0xa9e)   0x4c       starts @ 0x6f20
p3   0x9630:   208 (0xd0)    0x2718     starts @ 0x9638
p4   0x9838:   808 (0x328)   0x208      starts @ 0x9840
```

The hex values listed under `Addr` are assumed for this example to be the addresses where the entries are found. Each entry takes eight bytes, plus user data. The allocation pointer points to an address eight bytes from the beginning of each list entry.

The first allocated block has size 20 (rather than 10 bytes plus 8 bytes overhead), because blocks allocated by **malloc()** are 4-byte aligned. Therefore `Size` modulo 4 is always 0 (zero).

The blocks assigned to `p1` and `p2` have a `Prev_Size` of 0x24 and 0x4c respectively because the heap manager found previously allocated pieces of memory of that size to accommodate the two variables.

Free Memory

When an item is freed, the space that contained the user data becomes available to the heap manager. The free blocks are collectively referred to as the *free chain*. The next time it allocates memory, the heap manager can search the free chain to find a free block large enough to satisfy the request. For maximum efficiency in allocation, the heap manager maintains separate lists by size of the memory blocks in the free chain.

- The *free list* is a doubly linked list of free blocks. The heap manager has a static array of pointers to the first item in each free list. When an item is freed, it is placed at the front of the appropriate list and the free-list pointer is modified to point to it.
- A *free-chain link* consists of two pointers. These two pointers are located in the memory that is being added to the free chain, immediately following the heap-chain data. One of these pointers points to the next free block of memory; the other points to the previous free block of memory.

When space is requested from the heap, the heap manager allocates the first free block found that is large enough to hold the request. If the block is larger than the request, it is split into two items, the first allocated, the second free.

When an item is freed, if the item on either side is free, the items are coalesced into one free-chain item. Figure 13 Heap Chain Before a Block Is Freed shows the heap chain before a call to **free()**, and Figure 14 Heap Chain After a Block Is Freed shows the heap chain after a call to **free()**.

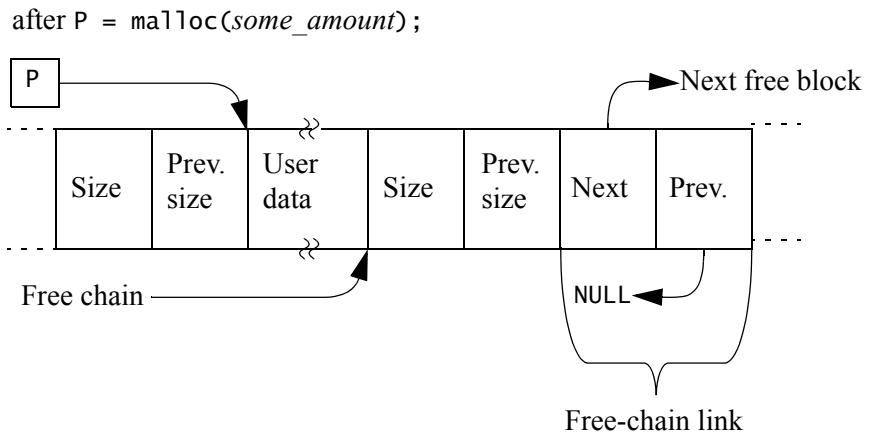

Figure 13 Heap Chain Before a Block Is Freed

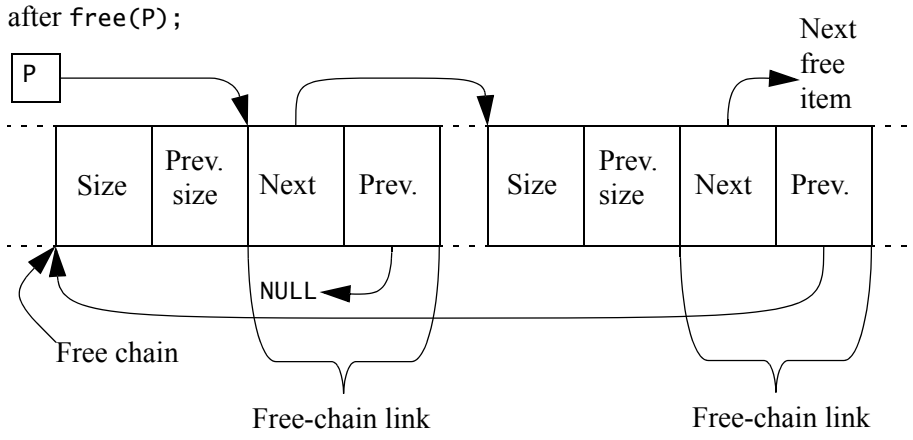

Figure 14 Heap Chain After a Block Is Freed

Integrity Checking

The heap manager provides integrity checking. To control the level of heap integrity checking, set environment variable `MALLOC_LEVEL`. You can set `MALLOC_LEVEL` in several ways.

In Source Code

To set `MALLOC_LEVEL` in your source code, include the following lines:

```
#include <stdlib.h>
_putenv("MALLOC_LEVEL=2");
```

In a Windows Command Prompt Window

On Windows hosts, enter this **set** command:

```
set MALLOC_LEVEL=2
```

You can also set `MALLOC_LEVEL` in an `autoexec.bat` file.

On the UNIX or Linux Command Line

On hosts running C shell, enter this **setenv** command:

```
setenv MALLOC_LEVEL 2
```

On hosts running Bourne or Korn shell, enter the following lines:

```
MALLOC_LEVEL=2
export MALLOC_LEVEL
```

Levels of Heap Integrity Checking

These are the levels of heap integrity checking:

0 No integrity checking.

1 Double links are used, which permits better coalescing of free space. No integrity checking. This is the default.

2 Items are checked as they are allocated and freed within the heap.

3 Complete heap integrity checking is performed at every call to **malloc()** and **free()**.

4 Complete heap integrity checking is performed at every call to **malloc()** and **free()**. In addition, if heap corruption is detected, a post-mortem heap dump is printed to standard output.

First-fit versus Best-Fit Strategy

By default, the heap manager uses a first-fit allocation strategy. However, it is also capable of using a best-fit allocation strategy, which results in more efficient use of available memory at the cost of run-time performance. To configure the heap manager to use the best-fit strategy, append the letter `b` or `B` to the numeric value you assign to `MALLOC_LEVEL`; for example:

```
set MALLOC_LEVEL=2b
setenv MALLOC_LEVEL 2B
```

Appendix E — Branch Prediction

In This Appendix

- Overview
- How to Perform Branch Prediction
- The bpmerge Utility
- Branch Prediction and Run-Time Performance
- The bplist Utility

Overview

This appendix describes an optimization known as branch prediction and two related utilities.

Branch prediction is an optimization that uses feedback from previous executions of the application to predict that certain branches will be taken (traversed) while other branches will not be taken.

Branch prediction relies on the compiler's ability to tag branch instructions according to whether one branch target is more likely to be used than the other. When you enable the branch-prediction optimization, the compiler analyzes branch-profiling statistics produced by previous "exercise runs" of the application.

If the profiling statistics provide the following information about a given branch:

- that the branch was visited at least once during the exercise run
- that one target of the branch was executed more frequently than the other

the compiler re-orders the code for that branch to reduce unneeded jumps; it also attempts to optimize fall-through cases.

These compiler options support branch profiling and prediction:

-Hgenbp_data	Compile the application and instrument it with code to generate a file containing branch-profiling information at execution time.
-Husebp=*bpfile*	Recompile the application using the branch-profiling data in *bpfile*.

For more information, see How to Perform Branch Prediction on page 326.

How to Perform Branch Prediction

Branch prediction consists of three main steps. The following sections cover these steps in greater detail, but here is a summary of the process:

1. Instrument your application with code to generate branch-profiling data.

   To enable the branch-prediction optimization, compile your application with option **-Hgenbp_data** specified. This option directs the compiler to instrument the application with code that generates branch-profiling data at run time.

2. Run the application to generate the branch-profiling data.

   After instrumentation, execute the application under a variety of circumstances (that is, perform multiple "exercise runs"), using as large a set of representative inputs as possible. Each exercise run produces a branch-profiling data file.

3. Recompile and incorporate the profiling data.

   When you have a sufficient amount of branch-profiling data, recompile your application with option **-Husebp** specified. The compiler incorporates the branch-profiling data into the compilation process.

Instrumenting Your Application

To instrument your application with code that produces branch-profiling information at run time, compile your application with option **-Hgenbp_data** specified; for example:

```
hcarc -Hgenbp_data -o test prog.c
```

At run time, the generated branch-profiling information is placed in the `.data` section. When the application terminates normally, it dumps the branch-profiling information into an ASCII file named `bprofile.out`.

Generating Branch-Profiling Data

To generate the branch-profiling information, run the application as many times as required, using different combinations of inputs to exercise as much of the application code as possible. After each execution, the application generates a `bprofile.out` file. This file contains the branch-profiling information for one exercise run of your application.

Save the `bprofile.out` file under a different name each time you finish an exercise run (for example, `bp1.out`, `bp2.out`, and so on).

When you complete all the exercise runs, invoke **bpmerge** to combine all of the individual profile files into a master profile file; for example:

```
bpmerge bp1.out bp2.out bp3.out master.out
```

See The bpmerge Utility on page 327 for detailed information about using **bpmerge**.

NOTE Running **bpmerge** is necessary only if you have more than one file of branch-profiling data.

Recompiling with Branch-Profiling Data

To optimize your application, making use of the generated branch-profiling information, recompile the application with option **-Husebp** specified; for example:

```
hcarc -Husebp=master.out -o test prog.c
```

Option **-Husebp** directs the compiler to incorporate the branch-profiling statistics from `master.out` into the compilation process, tagging branch instructions as “predicted taken” or “predicted not taken”, as appropriate, and making node-ordering decisions accordingly.

The bpmerge Utility

Use the **bpmerge** utility (located in the *install_dir*`/bin` directory) to merge branch-profiling statistics from multiple runs of the same application into a master statistics file.

Invoking bpmerge

This is the syntax for **bpmerge**:

`bpmerge` [*bp_opts*] *input_file*... [`-` | *output_file*]

The command-line components have the following meanings:

bp_opts	One or more **bpmerge** command-line options, separated by spaces
input_file	Input file (at least one required)
-|*output_file*	Optional output component

The output component, if specified, must be either the name of a file or the '`-`' character.

- If you specify '`-`' as the output, **bpmerge** writes ASCII output to `stdout`.
- If you specify an *output_file*, **bpmerge** writes its output to that file.
- If you do not specify an output component, **bpmerge** reads *input_file*, generates the output, and writes the output to *input_file*, overwriting the original *input_file*.

NOTE To preserve the contents of the *input_file*, you must specify an output component on the **bpmerge** command line.

bpmerge provides command-line options for specifying the output file's endianness (big or little) and mode (ASCII or binary), along with one option related to reading input files.

Table 17 Command-line Options for the bpmerge (Branch-Profiling) Utility

Option	Meaning
-a	Writes output as ASCII
-b	Writes output as binary
-be	Writes output as big-endian
-i	Ignores checksums in input files
-le	Writes output as little-endian

Determining Endianness and Mode of the Output

For the following example uses of **bpmerge**, suppose you have the following input files:

- `inp1` — Little-endian, ASCII mode
- `inp2` — Big-endian, ASCII mode
- `inp3` — Little-endian, binary mode
- `inp4` — Big-endian, binary mode

Default Attributes for a New Output File

When you merge more than one input file into an output file that does not yet exist, and you do not specify output endianness or mode with a **bpmerge** option, the first input file determines the default endianness and mode (ASCII or binary) of the output file.

For example, given this invocation, the new file `myoutput` is little-endian and ASCII, the same as `inp1`:

```
bpmerge  inp1 inp2 inp3 inp4  myoutput
```

Specifying Attributes of a New Output File

When you merge more than one input file into an output file that does not yet exist, and you do use **bpmerge** option(s) to specify output endianness or mode, the output file has the option-specified attributes, regardless of the first input file's endianness or mode.

For example, given this invocation of **bpmerge**, the newly-created output file `output2` is big-endian and binary, as specified by the **-be** and **-b** options:

```
bpmerge -be -b inp1 inp2 inp3 inp4  output2
```

Overwriting Attributes of an Existing Output File

When you merge one or more input files into an existing output file, and you use command-line options to specify endianness and mode, the output file takes on the specified attributes.

For example, assume that file `output4` already exists, and is little-endian and binary. Given this invocation of **bpmerge**, the input files `inp1` — `inp4` are merged with `output4`, which is rewritten to have the specified attributes, big-endian (**-be**) and ASCII (**-a**):

```
bpmerge -be -a inp1 inp2 inp3 inp4  output4
```

Retaining Attributes of an Existing Output File

When you merge one or more input files into an existing output file, and you do not use command-line options to specify endianness and mode, the output file retains its previous endianness and mode.

For example, assume that file `output3` already exists, and is little-endian and binary. Given this invocation of **bpmerge**, the input files `inp1` — `inp4` are merged into `output3`, which retains its original attributes (little-endian and binary):

```
bpmerge  inp1 inp2 inp3 inp4  output3
```

The attributes of `inp1` do not determine the attributes of `output3`, because `output3` already exists with different attributes.

NOTE With very few exceptions, the endianness of the generated code is not subject to change.

The only time that you might need to use the endianness controls (options **-be** and **-le**) is if you have another MetaWare C/C++ compiler on a different host system, and that compiler is generating output files that will be merged with output files created by this MetaWare C/C++ compiler.

Branch Prediction and Run-Time Performance

Even if all predictions are correct, you might not perceive an improvement in program execution time in all cases. The performance improvement is largely based on the state of the processor pipelines (the status of the various instructions in the pipeline at the time of branch processing).

Sometimes the time required to complete the processing of the instructions in the pipeline might be less than the time required to prefetch the instructions. Thus you might not see an improvement, even if the prefetch came from the correct target. You can be reasonably assured that your program's execution time will not be worse with branch prediction than it would be without. Some test cases, using branch prediction alone, have achieved a performance improvement of 8% to 10%.

The bplist Utility

The **bplist** utility (located in the *install_dir*/`bin` directory) processes a program's branch-profile output file, `bprofile.out`, and produces a source listing of the program annotated with information about the following:

- the number of times each line was executed
- the number of instructions generated for each line

See Generating Branch-Profiling Data on page 327 for information about producing a `bprofile.out` file.

With the **bplist** listing you can spot which source lines are hit the maximum number of times and are thus potential execution bottlenecks.

Invoking bplist

This is the syntax for **bplist**:

`bplist` [*bpl_opts*] *profile_data*

The command-line components have the following meanings:

bpl_opts	One or more **bplist** command-line options
profile_data	Input file of branch-profiling data generated from an exercise run of the program being analyzed

See Table 18 Command-Line Options for the bplist Utility for possible *bpl_opts* values.

Input File

The default **bplist** input file is `bprofile.out`, generated by the **bpmerge** utility. If you want to use a different input file, specify **bplist** option **-i** `filename`.

For information about producing a `bprofile.out` file, see Generating Branch-Profiling Data on page 327. For information about merging branch-profile information into a master file, see The bpmerge Utility on page 327.

Output

By default, **bplist** sends its output to `stdout`. If you want to save the output to a file, specify **bplist** option **-o** *filename* or the standard I/O redirection offered by your system.

Also by default, **bplist** paginates its output, placing header and footer information on each page of the output. To eliminate this header and footer information from all but the beginning and end of the listing, specify **bplist** option **-n**.

In the **bplist** output, long source lines are truncated, not wrapped over to the next line. The columns for line number, line count, and instruction count are dynamically resized to accommodate the largest values.

Specifying Format of the Output

bplist provides command-line options for specifying alternate input and output file names, along with various aspects of the output file format.

Table 18 Command-Line Options for the bplist Utility

Option	Meaning
-d	Displays debugging information
-h	Displays this table of options as help text
-i *filename*	Specifies alternate input file name; the default is `bprofile.out`
-l *length*	Specifies page length
-n	Specifies no pagination
-o *filename*	Specifies output file name; the default is `stdout`
-t *width*	Specifies tab width
-v	Sets verbose mode
-w *width*	Specifies page width

Example 64 Sample Output from the bplist utility provides an example of the branch-profile listing produced for the Eight Queens problem.

NOTE The values you get for the `Instr` (instruction count) column will vary, depending on your target processor. On some systems, this column is not supported.

Example 64 Sample Output from the bplist utility

```
 Branch Profile Lister
 Branch Profile And Source Coverage Report For File: queens.c

------+-------+-------+-------------------------------------
 Line | Count | Instr | Source Line
------+-------+-------+-------------------------------------
      |       |       | ...
      |       |       | ...
   25 |   113 |    18 | void Try(Integer I, Boolean *Q) {
   26 |   113 |    12 |    Integer J = 0;
   27 |       |       |    do {
   28 |   876 |     8 |       J++; *Q = False;
   29 |   876 |    29 |       if (Asub(J) && Bsub(I+J) && Cs
   30 |   113 |     6 |          Xsub(I) = J;
   31 |   113 |     1 |          Asub(J) = False;
   32 |   113 |     1 |          Bsub(I+J) = False;
   33 |   113 |     1 |          Csub(I-J) = False;
   34 |   113 |     3 |          if (I < size) {
   35 |   112 |     6 |             Try(I+1,Q);
   36 |   112 |     4 |             if (!*Q) {
   37 |   105 |     5 |                Asub(J) = True;
   38 |   105 |     1 |                Bsub(I+J) = True;
   39 |   105 |     7 |                Csub(I-J) = True;
   40 |       |       |                }
   41 |       |       |             }
   42 |     1 |    26 |          else *Q = True;
   43 |       |       |          }
   44 |       |       |       }
   45 |   876 |    28 |    while (!(*Q || J==size));
   46 |   113 |     0 |    }
```

```
47 |     1 |    14 | void main (int argc, char **argv) {
48 |     1 |     5 |    Integer I; Boolean Q;
49 |     1 |     4 |    size = 8;  /* default */
50 |     1 |     3 |    if (argc > 1){
51 |     0 |     6 |       long i = atoi(argv[1]);
52 |     0 |    13 |       if (i <= 0 ) printf("Bad size;
53 |       |       |       else
54 |     0 |    19 |       if ( i > MAX_SIZE) printf("Size
55 |       |       |          size = MAX_SIZE;
56 |       |       |       else
57 |     0 |    10 |          size = i;
58 |       |       |       }
59 |     1 |    13 |    printf("%s\n","go");
60 |     8 |    41 |    for (I =  1; I <=  size; Asub(I++)
61 |    15 |    49 |    for (I =  2; I <= size*2; Bsub(I++
62 |    15 |    51 |    for (I = -size+1; I <=  size-1; Cs
63 |     1 |     8 |    Try(1,&Q);
64 |     1 |     4 |    if (Q)
65 |     1 |    19 |       for (I = 1; I <= size;) {
66 |     8 |    35 |          printf("%4d",Xsub(I++));
67 |       |       |          }
68 |     1 |    10 |    printf("\n");
69 |     1 |     6 |    exit(!Q);
70 |     1 |     0 |    }
```

Appendix F — Diagnostic Messages

In This Appendix

- Overview
- File I/O Errors
- System Errors
- User Error and Warning Messages
- Changing the Message Format
- Controlling Warning Messages
- Checking the Compiler Return Code

Overview

This appendix describes the conditions under which the compiler is likely to produce diagnostic messages and presents ways to control the messages.

Diagnostic messages from the MetaWare C/C++ compiler report the following:

- file I/O errors
- system errors
- user errors and warnings

File I/O Errors

File I/O errors are fatal to your compilation. They can occur when you try to open a non-existent file, or when the compiler writes an output file and not enough disk space is available.

NOTE Fatal errors can result in compiler temporary files (files with a `.tmp` extension) being left on disk. You should remove these files.

These are the I/O error messages that you are most likely to encounter:

`filename: file not found.`

This message is produced when a source file specified on the command line cannot be found. In addition, when you run the compiler in ANSI mode, the files `hcansi.pt` and `hcansip.pt` must be accessible.

`***Cannot write to intermediate file.`

This message is usually caused by too little space on disk. Remove unnecessary disk files and try again.

System Errors

System errors are fatal to your compilation; you should rarely encounter them. The error messages take this form:

`>>>> S Y S T E M  E R R O R` *n* `<<<<, in` *Module*`:`*Function*
Error message text.

where the components of the message have the following meanings:

n	numbers the occurrences of system errors
Module	the module name
Function	the function name

These are the only system error messages that you should encounter:

`Dynamic array allocation/reallocation failed. Out of memory.`

This error indicates that there was too little memory to compile the program. This usually occurs when at least one function definition is pathologically large. Such functions should be broken up into smaller ones.

`Recover: Exceeded the following limit:` *Limit.*

A table overflowed while a syntax error was being repaired. The table limit is fixed, so no increase in memory can improve the situation. Repair the error.

Other system error messages are associated with internal compiler errors or inconsistencies that normally will not occur. If they do occur, send a test case to your technical-support provider.

User Error and Warning Messages

User diagnostic messages are grouped into three categories:

- lexical error messages
- syntactic error messages
- constraint error and warning messages

Error messages begin with 'E' and warnings begin with 'w'. Warnings do not ordinarily suppress object-file generation; errors always do. However, some warnings become errors when you run the compiler in ANSI mode. Also, you can change warnings to errors with pragma `Onwarn_error` (see Controlling Individual Warnings by Number on page 339).

Messages that report errors terminate compilation after the phase issuing the diagnostic, so errors that would otherwise have been detected by later phases are not reported until you have repaired all earlier errors and reinvoke the compiler.

All diagnostic messages are accompanied by the following:

- the file name
- a line number *nn*
- a column number *mm*
- a message number *xxx* (for compiler front-end messages)

in this form:

"*filename*", L*nn*/C*mm*/#*xxx*: Message text

Line and column numbers show where in your source the compiler detected the error. For lexical and syntactic errors, the erroneous line is displayed with a caret '^' beneath it at the point of error detection.

See the following sections for more information:

- Lexical Error Messages on page 336
- Syntactic Error Messages on page 336
- Constraint Error and Warning Messages on page 336

Lexical Error Messages

Lexical error messages are produced when an improperly formed word is detected, such as a string with a missing closing quote.

```
Levels   LINE #|----+----1----+----2----+----3----+----4----+--
             1 |void main() {
  1          2 |   char *S;
  1          3 |   S = "Hello;
       C16     ---------------^
E "ex1.c",L3/C16(#485):(lexical) String constant not terminated
```

Figure 15 Lexical Error Message: String Constant Not Terminated

Syntactic Error Messages

Syntactic error messages are produced for programs that are ill-formed at the phrase level, such as a missing "`;`" or an inserted spurious symbol. The message is accompanied by a statement of the `REPAIR` that the compiler performed so it could keep processing input.

```
Levels   LINE #|----+----1----+----2----+----3----+----4----+--
             1 |void main() {
  1          2 |   printf "Hello");
       C12     -----------^
  1          3 |   }
E "ex2.c",L2/C12:(syntactic)  unexpected
symbol:'<STRING>':"Hello"
REPAIR:   '(' was inserted before '<STRING>':"Hello"@"ex2.c",L2
1 user error   No warnings
End of processing, Fri Dec 31 23:59:99 1999          ex2.c
```

Figure 16 Syntactic Error Message: Unexpected Symbol

Constraint Error and Warning Messages

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are several hundred such diagnostics, each of which is meant to be

self-explanatory. Most of them prevent the generation of object code, but some are merely warnings intended to assist the programmer.

```
 Levels   LINE #|----+----1----+----2----+----3----+----4----+--
             1 |void main() {
  1          2 |   int i;
  1          3 |   i = Undeclared_identifier;
  1          4 |   }
 E "ex3.c",L3/C9(#237):Undeclared_identifier: Identifier is
undeclared.
 1 user error   No warnings
 End of processing, Fri Dec 31 23:59:99 1999          ex3.c
```

Figure 17 Constraint Error Message: Identifier Is Undeclared

```
 Levels   LINE #|----+----1----+----2----+----3----+----4----+--
             1 |void main() {
   1         2 |   int Unused;
   1         3 |   }
 No user errors   No warnings
 End of processing, Fri Dec 31 23:59:99 1999          ex5.c
 w "ex5.c",L2: Variable "Unused" is never referenced.
 No errors   1 warning
```

Figure 18 Constraint Warning: Variable Is Never Referenced

Changing the Message Format

You can modify the format of error messages and warning messages with command-line options **-Hefmat** and **-Hwfmat**, respectively. These options direct the compiler to format error and warning messages according to a quoted string containing a sequence of characters with special meanings. (You can look at the configuration file to view the actual text of the format directive.)

The formatting characters include the following:

%c	Column number
%f	File name
%l	Line number
%m	Comma: direct the compiler to include a comma
%n	Diagnostic message number
%q	Double quote: use to avoid quoting-convention problems of some shells
%t	Tab

Predefined Message Formats

You can specify several predefined formats by using special command-line options. This is the default MetaWare C/C++ error message format:

```
E "test.c",L1/C7/#237:  J: Identifier is undeclared.
```

The same message would appear as shown for each of the following predefined formats:

Format	Option	Message
Microsoft C	-Hmscerr	`test.c(1): error C237: J: Identifier is undeclared.`
Borland C++	-Hturboerr	`Error test.c 1: J: Identifier is undeclared.`
UNIX C	-Hunixerr	`"test.c", line 1: J: Identifier is undeclared.`

Changing the Error Message Format

To change the default MetaWare C/C++ error message format, such as this:

```
E "test.c",L1/C7/#237:  J: Identifier is undeclared.
```

to match Microsoft C error messages, you can specify option **-Hmscerr**.

You can also modify the `ARGS` line in the configuration file to include the following **-Hefmat** specification:

```
-Hefmat="%f(%l): error C%n: "
```

which generates the Microsoft C error message format:

```
test.c(1): error C237: J: Identifier is undeclared.
```

Changing the Warning Message Format

Similarly, to change the default MetaWare C/C++ warning message format:

```
w "test.c",L2/C7/#291:  printf: Function called
but not defined.
```

to match Microsoft C warning messages, you can specify option **-Hmscerr**.

You can also modify the `ARGS` line in the configuration file to include the following **-Hwfmat** specification:

```
-Hwfmat="%f(%l): warning C%n: "
```

which generates the Microsoft C warning message format:

```
test.c(2) : warning C291: printf: Function called
but not defined.
```

Controlling Warning Messages

You can control many of the compiler warning messages by level or by message number. By default, the compiler displays all warning messages. You can disable the display of specific numbered warning messages with pragma `Offwarn`, described in Alphabetical List of Pragmas on page 129.

For more information, see Controlling Groups of Warnings by Level on page 338 and Controlling Individual Warnings by Number on page 339.

Controlling Groups of Warnings by Level

The compiler supports five levels of warning diagnostics.

- Level 1 warnings are considered to be of the most interest.
- Levels 2 and 3 warnings are of less interest.
- Level 4 warnings are informational.

- Warning level 0 (zero) suppresses all warnings.

You can specify the level of warning messages to display with option **-w***n* or with pragma `Warning_level`(*n*). The `Warning_level` pragma has the following form:

```
#pragma Warning_level(n)
```

where *n* is a decimal integer constant denoting the level. Setting the warning level to *n* means that warning diagnostics with severity level *n* or lower are displayed. A setting of 0 (zero) suppresses all warnings. The default warning level is 3.

Controlling Individual Warnings by Number

Many warnings can be controlled individually by message number. The following pragmas override the current level of warning messages set by pragma `Warning_level`(*n*) or command-line option **-w***n*:

- `Onwarn`
- `Onwarn_error`
- `Offwarn`
- `Popwarn`

For more information about these pragmas, see Alphabetical List of Pragmas on page 129.

To suppress individual warnings, first find the warning numbers by compiling with toggle `Msgno` turned on (the default). Then use pragma `Offwarn` to specify the numbers of the warnings you want to suppress. For example, warning #257 has been disabled in this code:

```
#pragma Offwarn(257)
... if (i = j) ...    /* Warning disabled   */
#pragma Popwarn()
... if (j = k) ...    /* Warning restored   */
```

You can use pragma `Onwarn` to restore (re-enable) the warnings, although pragma `Popwarn` is preferable. When you want MetaWare C/C++ to be more stringent than usual, but you do not want to use full ANSI mode, use pragma `Onwarn_error`.

To change warnings to errors, find the warning numbers by compiling with toggle `Msgno` turned on (the default). Then use pragma `Onwarn_error` to specify the numbers of the warnings you want to promote to compiler errors.

For example, the compiler emits warning #60 when you use **return;** within a function that returns **int**. You can change this warning to an error, and cause the compilation to fail, by including this pragma in your code:

```
#pragma Onwarn_error(60)
```

However, not all warnings are emitted at the point in the code where the offending text applies. For instance, functions called but not declared are diagnosed at the end of compilation, so that only the last warning pragma has effect for all of them. For example:

```
#pragma Offwarn(291)
... undef_func1() .../* No warning issued here */
#pragma Popwarn()
... undef_func2() ...              /* Nor here */
/* Now all undefined function warnings are
   emitted, so that both lines are diagnosed   */
```

If your code contains undeclared functions, it is best to disable warning #291 for the entire compilation. You can use option **-Hpragma** to specify this on the command line, like this:

```
hcarc  prog.c  -Hpragma=Offwarn(291)
```

or on your ARGS line in the configuration file.

Common Superfluous Warnings

This section outlines some of the most common sources of warnings and how to disable them.

C++ Standard Template Library

Some implementations of the C++ standard template library (STL) produce the following compiler warnings with great frequency:

```
(# 88)  Expression has no side-effects.

(#160)  Preceding type qualifier is
        superfluous.

(#217)  Initialization of reference type '...'
        with type '...' (...) causes
        generation of temporary.

(#287)  Inline function of ... tree nodes
        exceeds size threshold of 100 tree
        nodes of compiler intermediate
        code for expansion; use -Hinlsize=nnn
        to change.
```

You can safely ignore the superfluous warnings listed in this section. To disable warnings, use pragma Offwarn.

C Programs that Predate void Return Value

Turning on toggle Int_function_warnings might generate numerous superfluous warnings in old C programs that do not use the reserved word **void** to indicate a function returning no result.

Inline Assembly Macros

On ARCtangent-A5 and later, you can turn off toggle Compress_asm_warn to disable warnings about inline assembly macros inhibiting optimization.

Functions That Never Return

If you are aware that your application contains functions that do not return, you can disable the related warnings by turning off toggle Check_function_returns.

Checking the Compiler Return Code

You can check the compiler's return code for errors or warnings with a batch command or script.

Windows Hosts

On Windows hosts, you can check the compiler's return code with the batch **if** command; for example, in a batch file containing these lines:

```
hcarc myprog
if errorlevel 1 goto err
...
:err
echo Errors detected.
```

With option **-Hretwc** you can direct the compiler to add the warning count to the return code as well:

```
hcarc myprog -Hretwc
if errorlevel 1 goto err
...
:err
echo Warnings and/or Errors detected.
```

UNIX and Linux Hosts

On UNIX and Linux hosts, you can check the compiler's error return code using a shell script such as the following Bourne or Korn script:

```
err(){
   echo Errors detected;;
   exit $ERRNO;;
   }
hcarc myprog
ERRNO=$?
if [ $ERRNO -ne 0 ] then
   err;;
fi
...
```

With option **-Hretwc** you can direct the compiler to add the warning count to the return code as well:

```
err(){
   echo Errors and/or Warnings detected;;
   exit $ERRNO;;
   }
hcarc myprog -Hretwc
ERRNO=$?
if [ $ERRNO -ne 0 ] then
   err;;
fi
...
```

This is a C-shell equivalent to the with-warning-count script:

```
hcarc myprog -Hretwc
if ($status != 0) then
echo Errors and/or Warnings detected
exit $status
endif
...
```

Appendix G — Quick Options List

This section presents a quick-access list of driver options. For complete listings of driver options and information on how to use them, see Chapter 4 — Using Driver Options

-A-	Recognize only predefined assertions and macros that begin with two underscores
-A*name***(***token***)**	Create a preprocessor #assert directive
-a4	Specify the ARCtangent-A4 processor series
-a5	Specify the ARCtangent-A5 processor series
-arc600	Specify the ARC 600 processor series
-arc700	Specify the ARC 700 processor series
-c	Generate object file but suppress linking
-compat_a6	Optimize for ARC 700 with ARC 600 compatibility
-compat_a7	Optimize for ARC 600 with ARC 700 compatibility
-core1	Generate code specific to ARCtangent-A5 and later version 1
-core2	Generate code specific to core version 2
-core3	Generate code specific to ARCtangent-A5 core version 3
-core5	Generate code specific to core version 5 (ARCtangent-A4 and previous)
-core6	Generate code specific to core version 6 (ARCtangent-A4 and previous)
-core7	Generate code specific to core version 7 (ARCtangent-A4 and previous)
-core8	Generate code specific to core version 8 (ARCtangent-A4)
-D*name*[*=def*]	Define a name to the preprocessor
-dwarf1	Generate debugging information in DWARF 1 format
-E	Preprocess without compiling; write preprocessed file to standard output
-fdouble	Use not less than double-precision floating-point arithmetic
-fsingle	Use single-precision floating-point arithmetic
-fsingle2	Do not widen float types to double types
-g	Generate debugging information

-g0	Generate debugging information with line numbers only
-g1	Generate only “optimization-friendly” debugging information
-h[*string*]	Search mini-help files for information about driver options or about string
-H	Display names of files included during compilation
-Hall_sdata	Place all data in small-data sections
-Hanno	Annotate assembly listing with lines from the source file
-Hansi	Accept only ANSI C Standard-conforming programs
-Hasext=*ext-list*	Recognize new assembly-language filename extension
-Haom	Compile application to use the Automated Overlay Manager
-Haom_nolib	Compile overlay-safe code without Automated Overlay Manager run time
-Haom_overlay_only	Prepare module for inclusion in a overlay package
-Hasmcpp	Process C-style preprocessing directives in assembly source file
-Hasopt=*xxx*	Pass assembler options to the assembler
-HB	Compile in big-endian mode
-Hbatch[=*filename*]	Create a batch file to call the compiler and linker
-Hbatchnotmp[=*filename*]	Create a batch file to call the compiler and linker without temporaries
-Hcext=*ext-list*	Recognize new C source-file extension
-Hccm	Build for closely coupled memory (a Harvard architecture)
-Hcomdat	Manually enable COMDAT solution for template instantiation
-Hcplus	Include C++ header files and link C++ run-time library
-Hcpp	Use the outboard macro preprocessor
-Hcppext=*ext1*, *ext2*, ...*extn*	Specify the C++ source-file extensions
-Hcppfile_lvl=*n*	Specify the level of C++ compilation by file type
-Hcpplvl=*n*	Specify the level of C++ compilation
-Hcrt=*path_or_file_name*	Specify a run-time startup object file to be linked in
-Hc_wrap={*xxx*|^*yyy*}	Interpret header files included in C++ files as C, not C++
-Hdepend=depend-dir/	Specify directory for makefile dependency file

-Hecpp	Disable certain C++ features for Embedded C++ support
-Hefile={@[E]|filename}	If output sent to stderr, redirect it to stdout or to a file
-Hefmat=format	Set the format of error messages
-Heos=rtos_name	Select the specified RTOS library
-Heoslib=*pathname*	Specify path to the RTOS library file
-Hexcept	Compile and link exception-aware code
-Hexesize=*size*	Specify maximum size of executable
-Hextfile=*file*[,*file*...]	Specify extension directives in assembly files
-Hfirst=*obj_file*	Link specified object file first
-Hgac=*format*	Set a value for pragma `Global_aliasing_convention`
-Hgcc	Improve compatibility with GCC
-Hgenbp_data	Instrument application with code to generate branch-profiling data
-Hheap=*size*	Specify the heap size
-Hhlsize	Specify hostlink buffer size
-Hhostlib=*library*	Specify a library to implement run-time system calls
-Hi	Inline routines type-qualified with **_Inline**
-Hi[modifier]	Invoke the outboard inliner during compilation
-Hinlfactor=*nnn*	Suppress inlining if function would expand by more than specified percentage
-Hinlsize=*size*	Specify the maximum size of inlined functions
-Hipname=*NAME*	Set the name of the environment variable containing the include-file search path
-Hkanji	Recognize Kanji character set
-Hkeep	Do not delete object file at link time
-Hkeepasm	Do not delete intermediate assembly source files
-HL	Compile in little-endian mode
-Hldopt=*xxx*	Pass linker options to the linker
-Hlibobj=*filename*[, *filename*…]	Link in alternative library object files
-Hlines=*n*	Specify number of lines printed per listing page

-Hlist[=*filename*]	Generate a source listing
-Hlm=*xxx*	Enable left margin settings
-Hmake[=*name*]	Write a list of makefile dependencies to standard output
-Hmakeof=*filename*	Write a list of makefile dependencies to the named file
-Hmerge	Concatenate intermediate source
-Hmscerr	Use Microsoft error message format
-Hnalib	Allow extra-ANSI function calls without leading underscore
-Hnoarcexlib	Do not include `bin/arcexlib.s`
-Hnoarcsimd	Do not include `bin/arcsimd.s`
-Hnocomdat	Disable the COMDAT solution for template instantiation
-Hnocopyr	Do not display copyright message
-Hnocplus	Do not include C++ `.h` files or run-time library
-Hnocpp	Use the inboard macro preprocessor
-Hnocrt	Do not link in supplied run-time startup files
-Hnoinline	Do not invoke the outboard inliner, regardless of optimization level
-Hnolib	Do not link in MetaWare library
-Hnonexcept_wrap=*filename*	Wrap specified include file to compile without exception handling
-Hnoobject	Do not generate object code or invoke the linker
-Hnopro	Do not read a `.pro` file
-Hnosdata	Generate code without use of a small-data area
-Hnothread	Compile and link single-threaded programs
-Hnoxcheck	Disable run-time checking for ARC extensions
-Hnoxcheck_obj	Exclude **-Hnoxcheck** stub from link
-Hobjdir=*path*	Specify an object-file directory
-Hoff=*Toggle_name*	Turn off a toggle
-Hoffwarn	Disable specified warning messages
-Hon=*Toggle_name*	Turn on a toggle
-Hpcc	Place the compiler in PCC mode to relax some ANSI C Standard restrictions

-Hpfnoflt	Link in `pfnoflt` object file
-Hpictable	Generate position-independent code and data
-Hpragma=*pragma_name*(*pragma_argument ...*)	Invoke a pragma on the command line
-Hpro=*file*	Specify a `.pro` file
-Hprofile	Instrument executable to display run-time profile information at exit
-Hreg_remove=*reg*[*s*]	Remove a register from the processor
-Hreg_reserve=*reg*[*s*]	Reserve a register for application-specific use
-Hrel	Omit absolute include files from dependency entries
-Hretwc	Add the warning count to the compiler's return code
-Hrm=*xxx*	Enable right margin setting
-Hrtldir=*dir*	Specify an alternate run-time library directory
-Hsdata0	Generate code without use of a small-data area
-Hsfnoflt	Link in `sfnoflt` object file
-Hsilent	Suppress screen output
-Hstack=*size*	Specify the stack size
-Hsuffixfix=`.`*suffix*	Specify the object file extension
-Hthread	Compile and link thread-safe programs
-Htmpprefix=*temp_dir*	Set a directory for temporary files
-Hturboerr	Use Borland error message format
-Humake[=*macro*]	Write makefile dependencies to a `.u` file
-Hunixerr	Use AT&T Portable C Compiler error message format
-Hunroll=*Loop_count*,*Size*	Replace loops with loop-body code
-Husebp=*bpfile*	Recompile the application using branch-profiling data
-Hv4compat	Revert to version 4.x settings
-Hwcsize=*value*	Specify the size of type **wchar_t**
-Hwfmat=*format*	Set the format of warning messages
-I*dir*	Specify an alternate directory to search for include files
-L*paths*	Specify paths to search for libraries specified with option **-l**

-l*name*	Specify a library to include in the link
-M*model*	Specify far data, near or far code
-o *output_file*	Specify name of executable file
-O[*n***]**	Set the optimization level
-p	Generate code to count calls to each function (enable statistical profiling)
-P	Preprocess source file but do not compile; write output to a file
-pg	Generate code to count calls to each function (enable statistical profiling with call graphing)
-rf16	Support "register file 16" reduced register set
-S	Produce an assembly source file instead of an object file
-U*name*	Undefine a symbol (macro name)
-V	Display controls passed to the compiler; also display subprocesses
-w*n*	Set the compiler diagnostic warning level
-X{*a*|*c*|*m*|*t*}	Specify the degree of conformance to the ANSI C Standard
-Xadds	Enable support for ADDS and SUBS instructions
-Xbarrel_shifter	Enable support for barrel-shifter instructions
-Xcrc	Enable support for CRC instruction
-Xdpfp[*_fast*|*_compact*]	Enable double-precision floating-point extensions
-Xdvbf	Enable support for ARCtangent-A4 dual Viterbi instruction
-Xdsp_packa	Use DSP Pack A extension instructions
-Xea	Enable support for extended arithmetic instructions
-Xlib	Enable commonly used extension options
-Xlibd	Enable commonly used extension options with DSP compatibility
-Xmin_max	Enable support for MIN and MAX instructions
-Xmpy	Enable the ARC 700 general-purpose multiply extension
-Xmul32x16	Enable 32x16 multiply instructions
-Xmul_mac	Enable support for MULMAC extension instructions
-Xmult32	Enable support for scoreboarded multiplier instructions

-Xmult32d	Remap registers to allow multiply and XMAC together
-Xnorm	Enable support for the NORM instruction
-Xscratch_ram	Enable support for the scratch RAM extension
-Xspfp[_*fast*|_*compact*]	Enable single-precision floating-point extension instructions
-Xswap	Enable support for the SWAP instruction
-Xtimer0	Use timer 0 extension
-Xtimer1	Use timer 1 extension
-Xvbfdw	Enable support for ARCtangent-A5 dual Viterbi instruction
-Xvraptor	Build for VRaptor Instruction Set
-Xxmac_16	Enable support for 16-bit XMAC extensions
-Xxmac_24	Enable support for 24-bit XMAC extensions
-Xxmac_d16	Enable support for dual 16-bit XMAC extensions
-Xsimd	Enable SIMD Extensions
-Xvraptor	Build for VRaptor Instruction Set
-Xxy	Enable support for the XY memory extension
-YI,*dir*	Specify a directory to search last for include files
-YP,*dir1*[:*dir2* [:*dir3*…]]	Specify default directories for locating libraries

Appendix H — Quick Toggles List

This section presents a quick-access list of compiler toggles. For complete listings of compiler toggles and information on how to use them, see Chapter 5 — Using Compiler Toggles

Align_branch_targets	Ensure word alignment where possible
Align_uncached_section	Pad uncached data section to multiple of 256
Asm_percent_reg	Require '%' on register names
Auto_class_member_instantiation	Enable automatic instantiation of class templates
AutoDebug	Place named SDI for a named **struct** in the **struct**'s debug repository
Auto_func_instantiation	Enable automatic instantiation of function templates
Auto_reuse_warnings	Warn if **auto** variable declared twice in a function
Behaved	Specify a "well-behaved" program
Bool	Enable support for the C++ **bool** type
Borland	Relax some C++ standards to conform to the Borland class library
Branch_reductions	Generate compare-and-branch instructions
Call_trace	Generate information about a function being called
Calls_preserve_flags	Preserve condition-code flags when calling functions
CFA	Generate DWARF 2 CFA information for call-stack tracing
Char_default_unsigned	Make default representation of **char** unsigned
Char_is_rep	Recognize type char as being identical to the default representation chosen for **char**
Check_function_returns	Issue warning if function does not return
Class_common_always	Always generate virtual function tables for a class
Class_common_ok	Generate virtual function tables when functions in the class are defined
Cmp_zero_elim	Remove compares with zero
Command_line_ident	Put information in a comment section in generated code
Common_in_small_data	Assign common variable declarations to small-data section
Compress_asm_warn	Warn on use of inline assembly macros

Conditional_calls	Emit conditionally executed function calls
Debugger_handles_reg_vars	Allocate variables to registers
Default_far	Generate **_Far** calls for all function calls not declared as **_Near**
Defer_template_defn	Defer the definition of a template class until its definition is needed
Define_static_members	Supply implicit definition of static class data members
Define_unreferenced_template_members	Define all template members regardless of whether they are referenced
Direct_ldst_optimize	Improve memory addressing
Double_constants	Use double precision for float constants
Double_math_only	Perform double-precision floating-point arithmetic
Double_return	Make non-prototype functions return type **double**, not type **float**
Downshift_file_names	Convert include file names to all lowercase
Dpfp_div_em	Use DPFP extension when emulating floating-point division
Dsf_hoist_from_target	Fill branch delay slot by hoisting instruction from target
Each_function_in_own_section	Map each function to a separate section
Each_var_in_own_section	Map each variable to a separate section
Ecpp_error	Generate error message when detecting certain non-EC++ features in Embedded C++ code
Emit_core_directives	Emit core-register-extension directives
Emit_line_records	Write a line table in the generated object file
Empty_is_void	Force C++ (void) interpretation of function declarations with no arguments
Enforce_access_control	Allow compiler-enforced access control
Ensure_instantiation	Force instantiation of all template types
Epilog_merge	Use the same epilog for multiple functions
Epilog_trace	Generate information about a function being exited
Exception_aware_class	Compile exception-aware classes
Export_vtabs	Make virtual function tables public and defined
Fast_virtual_bases	Speed access to members of virtual bases

For_dcln_local	Treat loop-initialization variables as local to the loop
ForceDebug	Place named SDI for any in-scope structure in current compilation unit
Func_replacement_prolog	Generate a prolog for use in redirecting a function
Function_template_deduction	Use 10/96 ANSI C++ draft function template matching rules
Inline_common	Solve the single-copy problem for inline functions (COMDAT)
Inline_divide	Generate inline code for integer division
Import_vtabs	Assume virtual-function tables to be external for classes with inlined, pure virtual functions
Interprocedural_optimizations	Determine the best possible order for generating code for functions in a module
Int_function_warnings	Warn about missing and mistyped returns
Keep_global_refs_in_loops	Do not move expressions with global or external variables out of loops
Ldst_from_code	Access instruction memory using load and store instructions
List	Generate a source listing
List_includes	Display names of files included during compilation
List_macros	List all macros to stdout
Long_enums	Map type **enum** variables to an **int**-sized area of memory
Make_externs_global	Make objects with storage class **extern** be global
Masking_analysis	Remove redundant bit-masking instructions
Microsoft	Relax some standards to conform with Microsoft C/C++
Msgno	Display error message numbers
Multiple_var_defs	Use UNIX “common” name strategy to implement semantics
Nested_enum_mangling	Correctly handle the mangling of nested enumeration types
Nested_types	Make nested types invisible outside the class where they are declared
NoDebug	Do not generate SDI for any named structure within toggle’s scope
Normalize_file_names	Remove redundant relative path information from include file names

Overload_info	Show function chosen during overload operation
Packed_means_unaligned	Make packed structs unaligned
Parm_assignment_warnings	Warn when assignments are made into a routine's value parameter
Parm_warnings	Warn when non-prototyped function arguments do not match declared parameters
PCC	Relax ANSI C Standard restrictions to compile PCC-compatible programs
PCC_msgs	Reduce compiler diagnostic capabilities to the PCC level
Pointers_compatible	Assign pointer values to incompatible pointer variables
Pointers_compatible_with_ints	Make any type pointer compatible with **int** types
Postfix_takes_two_args	Force overloaded postfix operators to require two arguments
Pre_allocate_limms	Assign frequently used limms to unallocated machine registers
Predicate_transformations	Transform conditional branches to conditional execution
Print_lattice_members	Include a list of class members in the class lattice display
Print_lattices	Generate a graphical view of any class with one or more bases
Print_layout	Print layout of C++ data structures
Print_protos	Write prototype-style function headers to standard output
Print_template_usage	Display templates used in a C++ compilation
Print_var_info	Display mapping of auto-, register-, and argument-class variables
Prolog_trace	Generate information about a function being entered
Prototype_conversion_warn	Warn when a prototype declaration forces conversion of a function argument
Prototype_override_warnings	Warn when an ANSI Standard C declaration overrides an old-style function definition
Read_only_strings	Store all string constants in the read-only data section
Recognize_library	Inline code for certain functions in the ANSI Standard C library
Reference_all_functions	Create a reference to every non-inline function in the program
Relative_includes	Look up included files relative to the directory of the including file

Respect_far	Treat functions declared to be **_Far** as far calls
RTTI_common	Solve the single-copy problem for RTTI structures (COMDAT)
Run_time_type_info	Generate and store run-time type information (RTTI) for a class
Scalars_in_bss	Place all uninitialized variables in the BSS section
Seat_funcs_in_dcln_order	Base seat of class on source order
Source_code_order	Produce code in source order where possible
Specify_override	Require **_Override** for overriding subclasses
Support_cast_keywords	Enable cast keywords in C++ code
Support_exceptions	Enable exceptions in C++ code
Support_multiple_inheritance	Enable multiple inheritance in C++ code
Support_mutable	Enable **mutable** in C++ code
Support_namespaces	Enable namespaces and **using** features in C++ code
Support_run_time_type_info	Enable run-time type information (RTTI) and **typeid** in C++ code
Support_templates	Enable templates in C++ code
Support_typename	Enable **typename** in C++ code
Support_vbases	Enable virtual bases in C++ code
Switch_pic	Generate only position-independent switch tables
Tail_jumping	Replace calls to certain functions with jumps to those functions
Tail_recursion	Optimize recursive functions where possible
Template_common	Solve the single-copy problem for template functions and classes (COMDAT)
Template_return_mangling	Allow overloading of template functions by return type
Template_trivial_conversions	Allow trivial conversions for matching arguments to function templates
Thread	Specify that variables are to be stored in thread-local storage
Trace_global_ctors	Indicate construction and destruction of global objects
Treat_exports_as_locals	Change public functions to local static functions
Trigraphs	Enable trigraphs

Try_static_cast	Replace old-style type casts with **static_cast**
Unaligned_means_packed	Pack unaligned structs
Uncached_in_own_section	Place uncached data in separate section
Undefined_preprocessor_id	Warn on undefined **#if** and **#elif** identifiers
Unroll_loops	Allow unrolling of loops
Unsized_arrays_in_small_data	Allocate external unsized arrays in small data
Use_millicode	Generate calls to millicode functions to reduce size of prolog and epilog
Use_reciprocal_for_divide	Convert a floating-point divide-by-constant to a multiply-by-reciprocal
Use_sdata	Respect designation of `%gp` as the small-data base register
Use_UP_rules	Widen unsigned char and unsigned short to unsigned **int**
Volatile_cache_bypass	Bypass the data cache when accessing volatile data
VP_UP_warn	Warn about expressions with more than one value
Vtable_common	Solve the single-copy problem for virtual-function tables (COMDAT)
Wait_for_license	Wait until compiler license is available
Widen_float_args	Widen float arguments to **double**
Xmac_d16	Support dual 16-bit XMAC extensions in compiler
Xmul_32x16	Support 32x16 multiply instructions in compiler

Appendix I — Quick Pragmas List

This section presents a quick-access list of compiler pragmas. For complete listings of compiler pragmas and information on how to use them, see Chapter 6 — Using Compiler Pragmas

`Alias`(*Internal_name*,"*External_name*")	Assign an external name to an internal identifier
`Align_members`([*n*])	Set the maximum boundary for structure-member alignment
`Align_to`(*alignment*, *var*[, *var*...])	Specify the alignment of global or static variables
`Alloc_text`(*seg_name*, *function*[, *function*...])	Group functions in specified text segment
`Asm`("*assembly_string*")	Insert a line of assembly code
`Asm_field_offsets`(["*filename*",]["*naming_spec*",] *struct1* ...)	Find field offsets
`Aux_register`(*reg_num* [, *property*...])	Define an auxiliary register
`BSS`(["*BSS_name*"])	Specify an alternate BSS section name
`Called_infrequently` (*string* [, *string*])	Designate as "called infrequently" all functions whose names contain a specified string
`Calling_convention` ([*Attributes* [, *_DEFAULT*]])	Allow MetaWare C/C++ modules to call functions not compiled with MetaWare C/C++
`Code`(["*Section_name*"])	Specify the code section name
`Condition`(*condition_code* [, *property*...])	Define an extension condition code
`Core_register`(*reg_num* [, *property*...])	Define an extension core register
`Data`([[*Section_type*,]"*Sect_name*"])	Allocate named blocks for data storage
`Define_field_offsets`(["*filename*",]["*naming_spec*",] *struct1* ...)	Find field offsets
`Ensure_instantiation`(*T1*, *T2*,...*Tn*)	Ensure instantiation of specified template types
`Exclude_instantiation`(*T1*, *T2*,...*Tn*)	Prevent instantiation of specified template types
`Exit_expr`(*Expression* [,*Level*])	Evaluate the Expression argument at program termination

Function_alignment(*align*, *func*, ...)	Align functions to a power of two
Global_aliasing_convention(["*Form*"])	Construct external object names for aliasing internal variable and function identifiers
Ident("*Comment*")	Insert a comment in the generated object file
Initialization_level(*Level*)	Set initialization priority of modules
Interrupt_ext_words(*num_words*)	Provide space for saving extension state
Intrinsic(*function_name* [, *property*...])	Define a compiler intrinsic for an instruction
Literals("*name*")	Name a section for storing literals and static variables
Loop_factor(*Loop_count*, *Size*)	Unroll loops
Naked(*func1*[,*func2*,...*funcN*])	Do not save or restore prolog or epilog
Never_returns(*func*[,*func2*,...*funcN*])	Designate function as never returning
Off(*Toggle_name*)	Turn off a toggle
Off_inline(*function_name*[, *function_name* ...])	Turn off inlining for specific functions
Offwarn(*n1*, *n2*, ...)	Disable specified warning messages
On(*Toggle_name*)	Turn on a toggle
On_inline(*function_name*[, *function_name* ...])	Turn on inlining for specific functions
Onwarn(*n1*, *n2*, ...)	Enable specified warning messages
Onwarn_error(*n1*, *n2*, ...)	Enable specified warning messages as compiler error messages
Pack([*n*])	Control default alignment of data structures
Page(*n*)	Insert page ejects in a source-code listing
Pop(*Toggle_name*)	Reinstate prior status of a toggle
Pop_align_members	Restore alignment prior to earlier Push_align_members
Pop_ignore(*Pragma_name*(Pragma_arg), ...)	Reinstate prior ignore status of listed pragmas
Pop_inline(*function_name*[, *function_name* ...])	Pop top-most stack entry for each specified function

`Pop_small_data()`	Reinstate prior small-data specification
`Popwarn`	Cancel previous pragma `Offwarn` or `Onwarn`
`Push_align_members(`*n*`)`	Set maximum boundary for structure-member alignment and save state
`Push_ignore(`*Pragma_name*`,` *Pragma_name* …`)`	Ignore pragmas listed
`Push_small_data(`*n*[`;`*m*]`)`	Assign data to small-data section
`Reg_remove(`*regs*`)`	Remove one or more registers from use
`Reg_reserve(`*regs*`)`	Reserve one or more registers for application-specific use
`Skip(`*n*`)`	Insert blank lines in a source-code listing
`Startup_expr(`*Expression*[`,` *Level*]`)`	Evaluate expression at startup
`Static_segment("`*Section_name*`")`	Name a section for storing initialized static variables
`STDC` *trait* [`ON` | `OFF` | `DEFAULT`]	Toggle a trait
`Title("`*string_c*`")`	Put title at the top of each page of the source listing
`Warning_level(`*n*`)`	Set the level of warning diagnostics
`Weak{`*x* = *y*|(*x*,*y*)|*x*`}`	Designate a weak global symbol and assign it a value

Appendix J — Quick Inliner Options List

This section presents a quick-access list of options for the outboard inliner. For complete listings and for more information about inlining, see Chapter 8 — Inlining

-Hi	Invoke the inliner during compilation to inline routines type-qualified with **_Inline**
-Hia	Do not inline routines that have their address taken
-Hib=*n*	Specify temporary-file buffer size
-Hic=*n*	Inline routines called n times
-HiC=*t*,*c*	Inline routines with t tree nodes, called c times
-Hih	Show which routines have been inlined and/or defined
-Hir	Do not inline recursive routines
-His=*n*	Inline routines with stack size less than n bytes
-Hit=*n*	Inline routines with fewer than n tree nodes
-Hiw	Invoke inliner in multi-module (wide inlining) mode
-Hix	Do not inline exported routines

Index

Symbols

#assert, 37
#if and **#elif**, warn on undefined identifiers, 122
#if and assertions, 37
#include, 258
  relative to including file, 114
#unassert, 37
%gp, 123, 227
  reserving, 76
%sp, 212
.arcextmap, 252
.bss, 252
.data, 252
.data1, 252
.di suffix, 124, 223
.fini, 229, 252
.fixtable, 101, 252
.got, 252
.heap, 213, 252
.init, 229, 252
.initdata, 252
.pictable, 73, 245, 252
.plt, 252
.pro file, 39
.pro file, 71
  specifying, 39, 73
.rodata, 252
.sbss, 252
.sdata, 252
.so files, 33
.stack, 212, 252
.text, 252
  changing name, 133, 159
.tls, 252
.ucdata, 223, 224, 252, 253
  aligning, 223, 224
@ file, 24
___errno(), 239
__arc_main(), 229
__attribute__, 314
__cplusplus, 36
__CPLUSPLUS__, 36
__DATE__, 36, 259
__declspec(thread), 238, 239, 254
  and errno, 239
__embedded_cplusplus, 36
__EXCEPTIONS__, 36
__FILE__, 37
__HIGHC__, 36, 37
__HIGHC_ANSI__, 37
__LINE__, 37
__PICTABLE__, 245
__PPROFILE__, 37
__start_kernel(), 229
__STDC__, 37
__TIME__, 37, 259
__Xsimd, 49
_ARC, 36
_ARC_BE, 36
_ARC_LE, 36
_ARCOMPACT, 36
_ARCVER, 25
_ASM(), 208
_ATT4, 36
_b symbol prefix, 219
_BE, 36
_CALLED_INFREQUENTLY, 132, 176, 315
_CALLING_CONVENTION, 317
_CC, 315
_DEFAULT_CALLING_CONVENTION, 317
_end, 213
_estack, 212
_Far
  if not declared **_Near**, 94
  treating as far, 114
  with **extern** function declaration, 79
_Far and **_Near** with extern function declaration, 79
_fini(), 229
_FORCE_INLINE, 316
_init(), 229
_init_pic, 228, 245
_initcopy(), 222, 229
_initcopy_pic, 245
_Inline, 65, 179, 186
_INTERRUPT1, 316
_Interrupt1, 316
_INTERRUPT2, 316
_Interrupt2, 316
_LE, 37
_MT, 37
_mwget_tls(), 240
_mwprof_countdown_freq, 243
_mwprof_countdown_ticks, 243
_mwprofile_clock_freq(), 243
_mwprofile_tick(), 243
_mwrtl_init(), 229
_mwstart_profile_clock(), 242, 243
_mwstop_profile_clock(), 243
_Near and **_Far** with extern function declaration, 79
_Near with **extern** function declaration, 79
_NEVER_RETURNS, 317
_NEVER_THROWS, 317
_Override, requring, 116

_Packed, 107, 151, 255
in legacy code, 152
_PRESERVE_FLAGS, 317
_Preserve_flags, 317
_Rarely(), 175
_REENTRANT, 37
_SAVE_ALL_REGS, 317
_Save_all_regs, 317
_sbrk(), 213
`_SDA_BASE_`, 217, 227
`_start`, 228
_swi(), 225, 306
_Unaligned, 152
_Uncached, 224, 253, 258
put in separate section, 122
_Unpacked, 255
_Usually(), 175
'%' on register names, 87

Numerics

16-bit **XMAC**, 50
enable, but do not generate, 126
24-bit **XMAC**, 50
32-by-32 scoreboarded multiplier, 48
32x16 multiply, enable, but do not generate, 126
370 CMS, 75

A

absolute include files, 38, 75
absolute pathname *defined*, 75
accelerator instruction (CRC), 46
accepting only ANSI applications, 54
access control, 99
accessing
misaligned data, 151
small data, 217
volatile objects, 258
adapting library to an RTOS, 226
address, do not inline function if taken, 186
addressing, near for globals, 173
ADDS and **SUBS** instructions, 45
aliasing internal names, 129
aligning
`.ucdata`, 223, 224
code and data, 174
functions, 139
structure members, 150
structures, 150
types and variables, 149
uncached data, 223
variables, 129
alignment, 255
and minimum C++ object size, 260
maximum member boundary, 129, 149
popping member, 144
saving, 145, 151
storage blocks, 149
variables, 149
word, 86
alloca(), 290
alternate run-time library directory, 75
annotating assembly listing with lines from source, 54
anonymous structure-member selection, 108
ANSI
10/96 C++ draft rules, 101
C Standard, 16, 21, 112, 121, 248
and PCC, 108
inlining functions, 113
relaxing some restrictions, 72
requiring for source files, 54
specifying degree of conformance, 83
C++ Standard, 16
application entry point, 228
ARC 600
driver command, 24
specifying, 43
ARC 700
driver command, 24
specifying, 43
ARC Media, 49
ARC *Programmer's Reference*, 16
`arc_main.c`, 229
`arc_main.o`, 229
Arcangel, 242
arcexlib.s, 69
ARChitect configuration files, 25
arcsimd.s, 45, 49, 50, 69
ARCtangent-A5
driver command, 24
specifying, 42
ARCtangent-A5.2 processor, 44
`ARGS`, 64
argument file, 24
arguments
limits, 260
mismatched in assembly functions, 204
passing, 283
warn if converted due to prototype, 112
warning of mismatch with non-prototyped functions, 107
arithmetic, enabling extended, 47
arrays in small data, 122
ASCII, 248, 258
asm and **_Asm**, 202
assembler options, 26
passing to the assembler from the driver, 56
assembly
calling, 133
calling C/C++, 268
calling from C/C++, 266, 267
eliminated through optimization, 209
file, generating instead of object file, 82
files, 24
and extension directives, 64
functions, 202
functions with mismatched arguments, 204

inlining with GCC mechanism, 192
inserting a line, 130, 208
keeping intermediate files, 67
listing with source lines, 54
sharing variables with C, 270
using C-style preprocessor directives in, 55
warn on use of inline macros, 93
assertions, 37
creating, 52
disabling all but predefined, 52
removing, 37
assigning
to value parameter, warn on, 107
variables to a register, 270
associativity, 177
AT&T Portable C Compiler, 72, 108, 124
error message format, 77
atexit(), 229
auto variables, 251, 253
`autoexec.bat`, 323
Automated Overlay Manager
intrinsics, 305
options, 54, 55
auxiliary registers, 130, 163, 274, 307

B

back-trace data structure, 278
barrel-shifter instructions, 43, 44
enabling, 46
base
classes, 261
register for small data, 123
virtual and embedded C++, 118
virtual, enabling (C++), 118
basing seat on source order, 115
batch file, 56
and library paths, 33, 56
without temporaries, 57
big-endian, specifying, 56
binary floating point standard, 21
bit fields, 255
allocation order, 256
boundary alignment of structs and unions, 256
int, 249
mapping, 256
named and unnamed, 256
storage unit boundaries, 256
volatile, 222
with GCC-style inline assembly, 197
bit masking
removing redundant, 105
bit-fields, 256
bool, 60, 88
Boolean expressions, optimizing, 175
Borland
C++, 89
error message format, 77, 338
branch
conditional, 109
delay slot, filling, 97
prediction, 174, 326–332
compiling for, 327
list utility, 330
merge utility, 327
profiling, 174
generating data, 64, 327
using data to recompile, 78
breakpoint
instruction, 302
software, 44
table, 44
BRK instruction, 44, 302
BSS, 158
section name, 160
specifying a section, 158
BSS name, 132, 160
BSS type for user-defined sections, 136
buffer for temporaries, 186
building
an application, 24
non-thread-safe libraries, 71
built-in functions, 302

C

C
header files in C++, 60
macro preprocessor, using UNIX, 58
preprocessor directives in assembly source, 55
sharing variables with assembly, 270
C++
compilation, 30, 58
declarations without arguments, 61, 98
disabling, 70
exceptions
and threads, 33
exceptions, enabling, 63, 116
specifying embedded, 61
specifying level, 27, 58
standard, 21
using C header files, 60
C/C++
calling assembly, 266
calling from assembly, 268
listing format, 298
listing source, 104, 147, 148, 298
cache
data, 224
incoherency, 222
called functions, generating information, 89
callee prolog, 277
call-graph profiling, enabling, 81
calling
assembly from C/C++, 266, 267
C/C++ from assembly, 268

conventions, 133, 314
attributes, 315
non-MetaWare modules, 133
calling conventions
predefined names, 317
calls
emit conditional, 94
preserving flags, 89
replacing with jumps, 118
tracing, 292
call-stack tracing, 90
case labels, limit, 260
case-shifting external names, 156
cast
C++ keywords, 116
C++ keywords and embedded C++, 116
pointer to integer types, 251
using static (C++), 121
CFA information, 90
changing warnings to errors, 339
char
compatibility, 248
default, 91
unsigned by default, 90
character
constants, 248
limits, 259
set, 248
size, 248
source sequences, 258
characters, three as one, 121
checking for returns, 91
classes
basing seat on source order, 115
defining static members implicitly, 95
direct and indirect bases, 261
exception-aware, 99
instantiating templates
automatically, 87
selectively, 138
lattice display, 110
layout examples, 262
size, 255
storage mapping, 260
virtual bases, 100
virtual function tables
generating always, 91
generating when defined, 92
clearing interrupt-mask bits, preventing, 90
code
compressing, 221
condition, 133, 166
epilog, 284
generating in source order, 115
pages in listing, 144, 298
portability, 21
position-independent, 228, 244
prolog, 284
section name, 133, 159
size and inlining, 182
smaller, 97
COMDAT, 57, 102, 114, 119, 125, 138
disabling, 69
command-line access to pragmas, 128
command-line help, 28, 53
comments in object files, 139
common
name strategy for semantics, 105
variable definitions in small data, 93
COMMON type for user-defined sections, 136
COMMON_DATA type for user-defined sections, 136
compact double precision floating-point, 46
compact single precision floating-point, 49
compare-and-branch (ARCompact), 89
compares with zero, eliminating, 93
compilation
conditional by core, 25
finding errors without generating object code, 71
suppressing (preprocess only), 53
compiler
calling into code from another, 133
information in object code, 93
pragmas, 129–148, 357–359
return code, 340
specifying output file name, 80
compiler-intrinsic functions, 302–306
and auxiliary registers, 307
and library functions, 306
defining, 140, 164, 306
for extensions, 306
compile-time constants, 302
compiling
C++ modules, 30, 58
object file without linking, 52
pre-ANSI C programs, 83
single-threaded programs, 71
thread-safe programs, 76
with branch profiling data, 78
without linking, 29
compiling and linking, 30
C and C++ modules, 30
embedded C++, 30
in one step, 29
in two steps, 29
compressing code, 221
concatenating intermediate source, 69
condition code, 303
defining, 133, 166
condition flags, 317
conditional
branches, changing to execution, 109
calls, 94
compilation by core, 25
jumps, 175
configuration file, 36, 38, 176
const variables, 253
section, 142, 161
const_cast

enabling (C++), 116
`Constant list longer than aggregate being initialized`, 64
constant, compile-time, 302
constants
character, 248
read-only, 175
single-character, 258
storing strings in read-only data, 113
wide-character, 249
construction
and initialization level, 139
before **main()**, 153
warning if global, 120
constructors, global static, 229
controlling inlining with pragmas, 185
controls
displaying passed, 33, 82
precedence, 34
scope, 34
conventions
document, 17
terminology, 17
conversions, trivial, 120
copying ROM to RAM, 221
copyright, suppressing, 70, 76
core registers, 134, 163
extension, 98
core, specifying version, 43, 44
counter size for profiling, 295
CRC accelerator instruction, 46
creating **#assert** directive, 52
`crt1.o`, 227, 228
`crt1pt.o`, 228, 245
`crti.o`, 229
customizing the compiler, 39, 128

D

data
back-trace structure, 278
position-independent, 244
section name, 148
structure layout, 111
data cache, 224
bypassing, 124
DATA type for user-defined sections, 136
data types
aligning, 149
date and time translation, 259
debugging, 31
format, 31, 53
hardware, 31
information, 53, 292
for structs, 87, 101, 106
reducing, 292
variables in registers, 94
declarations, standard and old-style function, 112
declarators, limits, 259
declared functions, checking for implementation, 114
decoder, Viterbi, 46, 50
deduction, function templates, 101
default
`.pro`file name, 39
application entry point, 228
char, 91
configuration file, 176
floating-point option, 51
heap size, 65
processor, 42
rounding mode, 250
stack, 212
stack size, 76
defining
a name to the preprocessor, 52
global variables multiply, 105
registers, 130, 134, 163
system environment, 36
template class when needed, 94
template members if not referenced, 95
degree of ANSI C conformance, 83
delay slots, filling, 97
delay-slot filling, 305
dependency file directory, 61
destruction, warning if global, 120
destructors
global, 229
inlining, 261
difference of two pointers, 251
direct bases, 261
direct non-virtual bases, 261
direct register assignment, 270
directives, preprocessing, 258
directory
for makefile dependency file, 38, 61
for temporary files, specifying, 77
specifying alternate for include files, 78
specifying default for library, 84
specifying search path for libraries, 33, 78
disabling
C++, 70
C++ features for embedded C++, 61
run-time extension checking, 45
displaying
controls, subprocesses, paths, 33, 82
layout of data, 111
names of files included in compilation, 54
templates, 111
variable mapping, 111
division
inline, 177
negative operands, 249
DLLs, 33
document conventions, 17
documentation, 15
double precision
floating-point math, 51, 96

extension, 46, 97
speeding extension, 97
for **float** constants, 96
double-to-**float** conversions, 250
DPFP
speeding, 97
driver, 24, 229, 310–312
and assembler options, 26
and linker options, 26
command, 24
configuration file, 64
input files, 27
specifying file names, 27
syntax, 24
using an argument file, 28
driver options, 24
for extensions, 45
for floating-point code, 51
for target processor, 42
in configuration file, 64
listing all, 29
passing to the linker, 67
quick-access list, 343–349
reference list, 42–84
driver options, command-line
defined, 42
DSP
multiplier extensions, 47
DSP options, 46
DSP Pack A, 46
DSP-compatible multiplier, 48
dual 16-bit **XMAC**, 50
dual Viterbi butterfly instruction, 46, 50
duplicate definition of function templates, 88
DWARF, 16
DWARF 1, 52
DWARF 2, 53
dynamic
link libraries, 33
stack allocation, 290
`Dynamic array allocation/reallocation failed`, 334
dynamic_cast, enabling (C++), 116

E

eliminating
compares with zero, 93
compiler warnings, 108
embedded C++, 21, 30
and cast keywords, 116
and exceptions, 116
enforcing, 98
specifying, 61
enabling
exceptions, 63
left margin setting, 68
right margin setting, 75
endianness
specifying big, 56
specifying little, 67
enhanced system timer 0, 49
enhanced system timer 1, 50
enhanced system timers, 242
entry
point, 228
tracing, 112
enum, 255
mapping to **int**-sized area of memory, 104
names nested in a class, 106
enumeration constants, limits, 260
environment variable for include file path, 67
epilog, 284, 314
eliminating, 142, 143, 154
errors, 99
optimizing, 99
trace information, 99
`errno`
and **__declspec(thread)**, 239
and thread safety, 239
`errnog.c`, 239
error messages
AT&T format, 77, 338
Borland format, 77, 338
displaying numbers, 105
for warnings, 144, 339
Microsoft format, 69, 338
specifying format, 62, 337
errors
constraint, 336
finding without generating object code, 71
lexical, 336
message format, 337, 338
PCC, 108
syntactic, 336
evaluate
expression before **main\(\)**, 148, 153
expression on exit, 139, 153
example program for small data, 220
`exc_off.h`, 71
`exc_pop.h`, 71
exception handling, 100
compiling files without, 71
exception-aware classes, 99
exceptions
and embedded C++, 116
enabling, 63, 116
executable, 24
file name, 28
specifying maximum size, 64
execution profiling, 293
execution speed, 152, 214
exit
evaluating, 139, 153
tracing, 99
exit(), 229
expansion of inlined functions, 66

exported functions, inlining, 188
expression
evaluating before **main()**, 148, 153
`Expression has no side-effects`, 340
extended arithmetic instructions, 47
extended Asm, 192
extension
core registers, 98
defining condition code, 133, 166
directives in assembly files, 64
instructions, intrinsics for, 140, 164, 306
options, 45
enabling most important simultaneously, 47
with DSP compatibility, 47
preprocessor macros, 45
registers, 130, 134, 163
state, 304
extension of source file name, 58
extensions, 164
compiling for, 98
condition code, 303
disabling run-time checking, 45
file names, 27
source file name, 27
extern, 254, 270
function declaration with **_Near** or **_Far**, 79
in loops, 104
variables, making global, 105
external
name, 266
symbol, 155
extra-ANSI calls without underscore, 69

F

far calls with **_Far**, 114
far data with near or far code, 79
fast double precision floating-point, 46
fast single precision floating-point, 49
faster code, 152, 214
faster DPFP, 97
field offsets
defining, 138, 162
finding, 130, 162
file
extensions, 27
I/O errors, 334
list names, 104
listing source to, 68
filenames
default executable, 28
extension, 27
changing, 58
reading as lowercase for **#include**, 96
removing redundant path information from includes, 106
files
included in compilation, displaying, 54
inlining across, 188, 189
non-exception-aware, wrapping, 71
temporary, specifying directory, 77
filling delay slots, 97
FLAG, 304
flags, preserving, 317
FLEX*lm* license manager, 125
float
constants, double precision, 96
use multiply-by-reciprocal for division, 123
widen to **double**, 125
suppressing, 51
floating point
conversions, 250
default, 51
double math only, 96
double precision, 51
enabling double precision extension, 46, 97
enabling single-precision, 49
options, 51
representations, 250
single-precision, 51
speeding double precision extension, 97
float-to-**int** conversions, 250
format
for error messages
AT&T, 77, 338
Borland, 77, 338
specifying, 62, 337
for warning messages
specifying, 78, 337
strings for output filenames, 84
FORTRAN, 133, 314
common blocks, 157
frame pointer register, 277
free chain, 321
free(), 320
functions, 178
_Inline, 65
_Rarely() and **_Usually()**, 175
assembly, 202
C++
standard templates, 101
instantiating automatically, 88
called but not defined, 338
called infrequently, 132, 176, 315
called, information, 89
calling C/C++ from assembly, 268
changing public to local static, 120
checking for implementation of declared, 114
checking for return, 91
defining intrinsic for extensions, 140, 306
displaying inlining, 187
do not inline if address taken, 186
each in separate section, 97
eliminating prolog and epilog, 142, 143, 154
extra-ANSI without underscore, 69
for copying volatile data, 223
grouping in a segment, 129
inlined member, 261

inliner, 183
inlining, 143, 145, 179, 185
ANSI C, 178
by calls and tree nodes, 187
by number of calls, 186
by number of tree nodes, 188
by stack size, 188
C++, 178
template classes, 189
inlining exported, 188
intrinsic for extension instructions, 164
intrinsic *see* compiler-intrinsic functions
millicode, 123
mismatched assembly arguments, 204
names
C, 266
C++, 267
never returning, 91, 143
never throwing, 317
non-prototype, 107
optimizing recursive, 118
overriding in subclasses, 116
pure virtual, 261
recursive, 179
reentrancy, 237
replacing calls with jumps, 118
replacing on execution, 318
return values, 284
rewiring, 101
sharing epilog, 99
specifying alignment, 139
specifying maximum inlined size (C++), 66
tables for virtual, 262
template
and single-copy problem, 119
overloading by return type, 119
tracing
calls, 292
on entry, 112
on exit, 99
virtual, 102
virtual tables, 100, 260
warn if overridden by ANSI declaration, 112

G

GCC
built-in functions, 302
GCC compatibility, 64
GCC inline assembly, 192
general-purpose multiply (ARC 700), 47
generated code, compiler information in, 93
generating
assembly file instead of object file, 82
branch-profiling data, 64, 327
debugging information, 53
extension instructions, 140, 164, 306
line numbers for debugging, 53
listings, 31
object file without linking, 52
global
aliasing, 64, 139, 155
and undefined attributes, 254
destructors, 229
extern inlinefunctions, 102
extern variables, 105
identifiers, 129
static constructors, 229
subobject count, 99
variables and objects near, 173
variables in loops, 104
weak symbol, 148
gmake, 240
GNU compatibility, 64
granularity of profiling, 294

H

hardware
accessing data, 222
debugging, 31
interfacing from C/C++, 222
Harvard architecture *see* closely coupled memory
header files, C in C++, 60
heap, 213, 320–323
and RTOS, 320
chain, 320
integrity checking, 322
manager, 320
size, 65, 213
specifying, 213
help
command-line, 28, 53
list all options, 29
host, object byte limits, 260
hostlink
buffer size, 65

I

I/O functions, 230
IDE, 15
`Identifier is undeclared`, 337, 338
identifiers, 139, 155
aliasing, 129
as weak global symbols, 148
case-sensitivity, 248
limits, 259
IEEE
754 Standard, 250
binary floating-point, 21
ignoring sequence numbers, 75
improving speed, 175
inboard macro preprocessor, 70
include directory, 83
include files, 258

in listings, 299
specifying alternate directory, 78
specifying environment variable name, 67
including
additional libraries, 33
C++ header files and run-time library, 30, 58
current directory first, 114
incompatible pointer variables, 109
indirect base, 261
infinite loops from interrupts, 225
infrequently called functions, 132, 176, 315
inheritance, multiple, 116
INITDATA (linker), 221, 229, 245
initialization
data structures, 153
level and construction, 139
priority, 139
`Initialization of ... causes generation of temporary`, 340
initialized global variable storage, 253
initializing
library, 229
memory, 221
RTOS, 229
`initpic.o`, 228
inline
assembly macros, warning on use, 93
certain standard functions by default, 113
division, 177
functions, 102
expansion, 66
within classes, 102
integer division, 103
inline, 178, 183, 188, 261
inline assembly, 192
GCC-style, 192
inline assembly and register names, 87
`Inline...exceeds size...`, 340
inlining, 182–190
a function, 143, 185
a line of assembly, 208
across files, 188, 189
and code size, 182
and optimizations, 182
and speed, 182
and **throw**, 189
assembly functions, 202
built-in *vs*. outboard, 183
by calls and tree nodes, 187
by function stack size, 188
by number of calls, 186
by number of tree nodes, 188
C++ functions, 144
containing a **throw**, 189
categories of functions, 65
class destructors, 261
command-line options, 183, 186–188, 361
disabling for a function, 143, 185
displaying, 187
exported functions, 188
functions, 178
with address taken, 186
member functions, 261
only **_Inline**, 65
popping specification, 145
recursive functions, 187
semi-automatically, 183
specifying maximum size (C++), 66
suppressing, 66
suppressing (C only), 70
template classes, 189
wide, 188, 189
with pragmas, 185
install_dir, *defined*, 17
instantiating
class templates
automatically, 87
by type, 138
function templates automatically, 88
instantiation, forcing for templates, 99
instruction scheduling, 305
instructions
defining intrinsics for, 140, 164, 306
memory space, 104
instrumentation for profiling, 242, 294
int bit fields, 249
integer
conversions, 249
division, 48, 250
inline, 103
multiply, 50
representations, 249
signed, 249
bitwise operations, 250
interfacing to Pascal and FORTRAN, 133
intermediate code
concatenating, 69
internal name, 266
interprocedural optimizations, 103
interrupt
extension space in frame, 140, 226
frame, 304
frequency for profiling, 294
handlers, 224, 279, 316
and thread safety, 240
macros, 226
reserving registers, 225
levels, 225
masking, 303, 304
stack frame, 279
unmasking, 304
vectors, setting, 226
intrinsics *see* compiler-intrinsic functions
int-to-**float** conversions, 250
invoking
outboard inliner, 65
pragmas from command line, 73
ISRs, 224

J

jumps
reducing conditional, 175

K

Kanji, characters, recognizing, 67
keeping
intermediate assembly files, 67
object file, 67

L

lattices, including class members, 110
law of associativity, 177
left margin, 68
level numbers in listings, 299
lexical errors, 336
`libaom.a`, 54, 55
libraries
adapting to an RTOS, 226
alternate directory for, 75
batch file for paths, 33, 56
do not link supplied, 71
eliminating, 227
functions for interrupt vectors, 226
host (run-time system calls), 65
I/O requirements, 230
initialization, 229
linking alternate, 68
location, 33, 75
millicode functions, 123
non-thread-safe, 32
paths, 33, 82
semaphores in, 240
specifying default directories, 84
specifying directory, 30
specifying for link, 33, 79
thread-local storage, 240
thread-safe, 32
thread-safe and not, 32, 71
library
rebuilding, 241
license, 125
limit
arguments, 260
case labels, 260
characters, 259
characters in an identifier, 248
declarators, 259
enumeration constants, 260
host system object bytes, 260
identifiers, 259
members in a structure, 260
members in a **union**, 260
nesting, 259
number of identifiers, 248
parameters, 260
limited character sets, 121
limms, preallocating frequently used, 109
line
numbers only (debugging information), 53
of assembly, inserting, 130
table in object file, 98
lines of source in assembly listing, 54
linker
conflicting options, 26
external symbols, 155
passing options to, 67
linking
alternate library object files, 68
libraries, 32, 79
one object file first, 64
small data, 218
suppressing, 52
thread-safe libraries, 32
without the driver, 229
linking rebuilt run-time libraries, 241
Linux kernel, 64
listing
include files, 104
macros, 104
source code, 104, 298
format, 298
specifying lines per page, 68
symbols, 227
listings
generating, 31
skipping lines, 147, 298
titles, 148, 298
LIT type for user-defined sections, 136
literals
section, 142, 161
storing in read-only data, 113
little-endian, specifying, 67
load/store
and instruction memory, 104
direct address substitution, 96
local
time on preprocessed files, 37
location of Run-Time Libraries, 33, 75
longjmp(), 253
loops
optimizations, 104
treating initialization variables as local, 101
unrolling, 77, 122, 142
variables and C++ level, 60
lowercase, **#include**filenames as, 96
low-level system calls, 230
LR, 304

M

macro **_ARCVER**, 25
macros

for interrupt handlers, 226
inboard preprocessor, 70
inline assembly, 192
listing, 104
symbols, 30
warn on use of inline assembly, 93
makefile, 38
creating, 38, 68, 77
dependency file directory, 38, 61
writing dependencies, 38, 68, 77
malloc(), 239, 240
and heap, 320
`MALLOC_LEVEL`, 322
mangling of nested **enum** names, 106
mapping
displaying, of certain variables, 111
of variables, 252
variables to a section or block, 137
margin
left, enabling and setting, 68
right, enabling and setting, 75
masking
interrupts, 303, 304
removing redundant, 105
members
defining offsets, 138, 162
finding offsets, 130, 162
non-static data, 261
memory
addressing, 96
enabling XY, 51
heap, 213
initializing, 221
misaligned accesses, 151
reclaiming (RTOS), 320
memory-mapped data, 222
Microsoft
Visual C++, 105
Microsoft error message format, 69, 338
millicodes, 123
MIN and **MAX** (ARCtangent-A4), 47
minimizing
conditional jumps, 175
minimum
C++ object size and alignment, 260
object size in C++, 260
misaligned
data, 151
memory accesses, 151
objects, 152
mixing C and C++ modules, 98
`mlo` register
mapping to `r58`, 48
`mmid`, replacing with `mlo`, 48
monitor(), 242
MQX and **volatile**data, 223
MULMAC instructions, 48, 50
multi-bit shifting (ARCtangent-A4), 46
multi-character constants, 248
multi-module inlining, 188, 189
multiple
definitions of global variables, 105
inheritance, 116
multiplier
for DSPs, 48
multiply extension with XMAC, 48
multiply, general-purpose (ARC 700), 47
mutable
and embedded C++, 117
enabling (C++), 116
mutex, 33
`mutexg.c`, 239
MVS systems, 75

N

`Naked` (pragma), 142, 154
name
BSS section, 132, 160
code section, 133, 159
data section, 148
named storage blocks, 136, 157
names
C ++ functions, 267
C functions, 266
in inlined code, 189
internal and external, 266
nested C++ **enum**, 106
of all included files, listing, 104
sections, 136, 157
namespaces
and embedded C++, 117
enabling (C++), 117
native floating point *see* floating point, 49
nested types, making available outside class, 106
nesting limits, 259
`Never_returns` (pragma), 143, 154
non-prototype functions and **double** returns, 96
non-static data members, 261
non-thread-safe libraries, 32
NOP, 305
NORM instruction, 43
and division, 48
and inline multiply, 48
enabling, 48

O

object
aliasing external name, 139, 155
finding referenced before constructed, 120
size of C++, 260
object code
compiler information, 93
suppressing generation, 71
object files
alternate library, 68

comments, 139
creating without linking, 52
keeping, 67
linking one first, 64
specifying directory, 72
specifying extension, 76
offsets of structure members, 130, 138, 162
operands
order in inline assembly, 194
operators
postfix, requiring two arguments if overloaded, 109
optimizations
and inlining, 182
interprocedural, 103
small data, 54
specifying level, 80
user-defined extension instructions, 140, 164, 306
options
outboard inliner, 186, 186–188, 361
options, *see* driver options
outboard
C macro preprocessor (UNIX), 58
inliner, 65, 183
output file
format strings for names, 84
specifying name, 80
overload
requiring two arguments for postfix operators, 109
showing which function chosen, 107
overriding
compiler defaults, 128
subclass functions, 116

P

packed
making structures unaligned, 107
structures, creating, 150
unaligned implies, 121
padding, 255
uncached data, 87
page ejects, 144
parameters
limits, 260
passing, 283
passing “by var”, 107
passing PL/1-style, 107
Pascal, 133, 314
passing
arguments, 283
options to the assembler, 56
options to the linker, 67
PCC mode, 72
performance bottlenecks, 175
PL/1 parameter passing, 107
PL/M, 133
pointers
assigning values to incompatible variables, 109
making compatible with integer types, 109
to virtual-function table, 262
virtual base, 263
well-behaved, 88
popping a toggle, 86
portability, 21
position-independent
code, 73, 118, 244
data, 244
startup, 228
switch statements, 118, 244
postfix operators, requiring two arguments if overloaded, 109
pragma `STDC`, 148
pragma *vs.*#**pragma**, 128
pragmas, 259
command-line option, 128
for inlining, 185
ignoring, 145, 259
invoking from command line, 73
parameters, 128
popping status, 145
quick-access list, 357–359
reference list, 129–148
syntax, 128
unsupported, 259
warnings, 259
preallocating limms, 109
precedence of compiler controls, 34
`Preceding type qualifier is superfluous`, 340
predefined
calling-convention names, 317
macro symbols, 36
names, 317
pre-fetch instructions, 305
preprocessing
directives, 258
without compiling, 27, 35, 53, 81
preprocessor
macro symbols, 30
macros, 45
macros defined by extensions, 45
predicates, 37
preprocessor identifiers, undefined, 122
preserve flags on calls, 89
printf() suppressing floating point, 72
printing
layout of data structures, 111
templates, 111
variable mappings, 111
priority of initialization, 139
profiling, 293
at application exit, 73
branch, 174
branch, generating data, 64, 327
enabling, 80
enabling (call-graphing), 81
granularity, 294

instrumentation code, 242, 294
integrated, compiling, 73
interrupt frequency, 294
parameters, 294
recompiling with branch-profiling data, 78
size of counters, 294
statistical, 293
support, 242
timers, 242
using timer 0, 49
using timer 1, 50
program entry point, 228
prolog, 284, 314
eliminating, 142, 143, 154
generate for redirecting functions, 101
trace information, 112
prolog, callee, 277
prototypes
warn if argument converted, 112
ptrdiff_t, 251
public
functions, changing to local static, 120
pure virtual member functions, 261

R

`r0`, 284
`r1`, 284
race conditions, 238
RAM, 221
read-only
constants, 175
data, 253
static variables, 142, 161
string constants, 142, 161
realloc(), 320
rebuilding libraries, 241
reciprocal for **float** division, 123
reclaiming memory (RTOS), 320
recognize
only macros with two underscores, 52
only predefined assertions, 52
recursive functions, 179, 187
optimizing, 118
redirecting
a function, 318
`stderr` to `stdout` or file, 62
reducing
conditional jumps, 175
debugging information, 53
reentrancy
and thread safety, 17
defined, 17
functions, 237
referencing
all non-inline functions, 113
register
16 file, 44
small-data base, 123
register names in inline assembly, 87
register variables, 251
registers, 274–276
and **_ASM()**, 209
and memory-mapped peripherals, 222
assigning variables directly, 270
auxiliary, 130, 163, 307
core, 134, 163
defining, 130, 134, 163
for interrupt handlers, 225
for return values, 284
placing objects in, 251
reading, 303
reduced set (ARC 600), 44
removing, 74, 146
reserving, 74, 147, 219
saving, 317
saving on interrupt, 225
writing, 303
reinterpret_cast, enabling (C++), 116
relative includes, 114
removing
preprocessor assertions, 37
registers, 74, 146
replacing a function, 318
reserving registers, 74, 147, 219
response file, 24
restoring
structure alignment, 151
warnings, 339
return
code, 340
eliminating save of address, 143
missing or mistyped, 103
type in template function overloading, 119
values, 284
rewiring a function, 318
right margin, 75
ROM, 221
RTOS
and libraries, 226
and thread safety, 238
initializing, 229
interface for threads, 238, 239
reclaiming memory, 320
specifying, 62
specifying path to library, 63
RTTI
and COMDAT, 114
and embedded C++, 117
enabling (C++), 117
generating for a class, 115
ruler in listings, 298
run-time
instrumentation code for profiling, 242
libraries, eliminating, 227
specifying startup file, 60
stack checking, 212

run-time type information *see* RTTI

S

saturating addition and subtraction, 45
saving
  all registers, 317
  maximum structure alignment, 151
  registers, 225
  state from interrupt handlers, 140, 226
sbrk(), 239
scalars, 115
scanf(), suppressing floating-point, 76
scope
  and pragma `Data`, 157
  block, of identifiers, 248, 259
  compiler controls, 34
  extern objects, 105
  in listings, 299
  loop-initialization variables and C++ level, 60
  nested C++ types, 106
  outer, in prolog code, 277, 279
  small-data specifications, 216
  struct,**union**, or **class** and C++ level, 59
scoreboarded multiplier, 48
scratch RAM, 48
scratch registers, 202
seat of a class, 115
section
  const variables, 142, 161
  for literals, 142, 161
  names, 136, 157
section `_end`, 213
sections
  listed, 252
  one per variable, 97
  per function, 97
segment, grouping functions in, 129
semaphores, 33, 240
  in run-time library, 240
semaphores in library, 240
separate section
  per function, 97
  per variable, 97
sequence numbers, ignoring, 75
serializing updates to the global state variables, 238
setjmp(), 253
setting
  compiler pragmas, 128
  interrupt vectors, 226
  left margin, 68
  of interrupt-mask bits, preventing, 90
  right margin, 75
shared function epilog, 99
shared-object files, 33
sign bit, 250
signal functions, 104
signed and unsigned
  bit fields, 255
  integers, 249
signed **char**, 248
SIMD extensions, 49
single-character constants, 258
single-copy problem, 57, 102, 119
  and RTTI structures, 114
  and virtual function tables, 125
single-precision floating point, 51
  extension, 49
size
  and alignment of enumeration types, 255
  and alignment of structures and unions, 255
  C++ objects, 260
  classes, 255
  code, 97
  enumerations, 255
  executable, specifying maximum, 64
  inlined functions, specifying maximum (C++), 66
  structures, 255
  temporary buffer, 186
`Size of .sdata+.sbss segments exceeds ... byte limit`, 218
SLEEP, 305
small data, 146, 214–221
  accessing, 217
  adding sections, 218
  and `%gp`, 227
  and common variable definitions, 93
  and linker, 218
  and **static** variables, 253
  avoiding but reserving `%gp`, 76
  base register, 76, 123, 217, 218, 219
  example program, 220
  for all data, 54
  popping specification, 145, 215
  suppressing, 71
  test program, 220
software
  breakpoints, 44
  interrupts, 225
software breakpoints, 44
source
  character sequences, 258
  listing, 104, 147, 148, 298
  listing format, 298
  listing to stdout or to a file, 68
  order in generated code, 115
  pages in listing, 144, 298
space
  for extensions to save state in interrupts, 140, 226
specifying
  a `.pro` file, 39
  alternate library directory, 30
  big-endian, 56
  C++ level, 27
  directory for temporary files, 77
  driver options, 42
  format for error messages, 62

format of error messages, 337
function alignment, 139
host library (run-time system calls), 65
level of C++ compilation, 58
library to link, 33
lines per listing page, 68
little-endian, 67
name of the environment variable for include search path, 67
object file directory, 72
RTOS, 62
RTOS library path, 63
run-time start-up file, 60
single-precision floating point, 51
stack size, 76
toggles, 86, 148
speed
and inlining, 182
improving, 152, 175, 214
virtual base accesses, 100
speeding DPFP, 97
SR, 306
stack, 212
aligning variables, 212
allocating dynamically, 290
back-trace data structure, 278
checking, 212
conventions, 276
default, 212
frame, 276
for interrupts, 279
pointer, 212
pointer register, 277
size, 76, 212
inlining functions by, 188
specifying, 76
standard C++, 21
start-up
do not link supplied, 70
startup
modules, 227
position-independent, 228
state, saving in interrupts, 140, 226
static
casts (C++), 121
constructors, global, 229
data members, defining implicitly, 95
instances of a class, constructing, 153
variables, uninitialized, 132
static, 253
and small data, 253
static_cast
enabling (C++), 116
treating as (C++), 121
statically bound thread-local variables, 254
statistical profiling, 293
STDC (pragma), 148
`stderror`
redirecting to `stdout` or file, 62
`stdout`, 56, 57
for `stderr` data, 62
listing source to, 68
storage, 252
arrays, 251
class objects, 260
classes, 260
named blocks, 136, 157
virtual bases, 261
storage blocks
alignment, 149
`String constant not terminated`, 336
string constants, storing in read-only data, 113
struct, 255
structures
aligning, 150
debugging information if in scope, 101, 106
defining member offsets, 138, 162
finding member offsets, 130, 162
maximum member alignment, 129, 144, 149
named debugging information, 87
number of members, 260
saving alignment, 145, 151
subprocesses, displaying, 33, 82
SUBS and **ADDS** instructions, 45
support, product, 14
suppressing
copyright message, 70
inlining, 66
C only, 70
linking of supplied library, 71
object code and ink, 71
printf() floating-point, 72
reading of `.pro` file, 71
scanf() floating-point, 76
screen output, 76
supplied start-up code, 70
use of small-data areas, 71
SWAP instruction, 43
enabling, 49
swapping, intrinsics for, 306
SWI, 225, 306
switch,, 244
switches *see* toggles
symbols
aliasing in the symbol table, 267
listing, 227
table entry, 266
undefining, 82
syntax errors, 336
system
calls, 230
errors, 334

T

tail
jumping, 118

recursion, 179
target processor options, 42
Technical Support, 14
template classes
defining only when needed, 94
inlining, 189
preventing instantiation, 139
template functions
and single-copy problem, 119
overloading by return type, 119
template instantiation
COMDAT, 57
template members
defining even if not referenced, 95
templates
and embedded C++, 117
displaying, 111
enabling, 117
forcing instantiation, 99
function, 101
temporary files, 77
terminology conventions, 17
testing for a preprocessor assertion, 37
TEXT type for user-defined sections, 136
thread control block, 239
thread safety, 76
and `errno`, 239
and exceptions, 33
and interrupt
handlers, 240
and RTOSs, 238
and **try**, 239
compiling without, 71
vs. reentrancy, 17
thread-local variables, 254
statically bound, 254
usage in startup and library, 240
three characters as one character, 121
throw
and inlining, 189
never occurs, 317
time, 37
timer
extensions, 242
for profiling, 242
timer 0, 49
timer 1, 50
`tlsctorg.c`, 239
toggles
defined, 86
controlling in source or `.pro` files, 86
popping status with pragma, 144
quick-access list, 351–356
reference list, 86–126
reinstating prior status, 86
turn off with pragma, 143, 148
turn on with pragma, 143, 148
turning off from command line, 72
turning on from command line, 72
`trace.c`, 292
tracing
at function entry, 112
at function exit, 99
function calls, 292
treating C++ files as C header files, 60
trigraphs, 258
enabling, 121
trivial conversions, 120
try
and thread safety, 239
never throws, 317
two-character constant, 248
twos-complement binary form, 249
type of `size_t`, 251
typeid, 115
and embedded C++, 117
enabling (C++), 117
typename
and embedded C++, 118
enabling (C++), 118
types, aligning, 149

U

unaligning and packing, 107, 121
uncached data, 87
placing in separate section, 122
undeclared functions, 340
undefined preprocessor identifiers, 122
undefining a symbol, 82
underscores, requiring two on macros, 52
`unexpected symbol`, 336
uninitialized
global variable storage, 253
static variables, 132
variables, storing in BSS, 115
union, 255
number of members, 260
UNIX
“common” name strategy, 105
macro preprocessor, 58
unmasking interrupts, 304
unreferenced variables, 97
unrolling loops, 77, 122, 142
unsigned
char, 90, 248
widen to **int**, 124
unwinding loops, 77
UP (unsignedness preserving) rules, 108, 124
warn on possible difference, 125
using
and embedded C++, 117
enabling (C++), 117

V

values returned by **sizeof()**, 251

variable polynomial CRC, 46
variables
  aligning, 129, 149
  aligning stack-based, 212
  allocating to registers for debugging, 94
  and memory-mapped peripherals, 222
  auto, 251
  default source-file extension, 27
  displaying mapping, 111
  `errno`, 239
  loop-initialization, 101
    and C++ level, 60
  mapping of, 252
  placing common definitions in small data, 93
  register, 251
  separate section, 97
  sharing in C and assembly, 270
  static uninitialized, 132
  statically bound thread-local, 254
  uninitialized, storing in BSS, 115
  unreferenced, 97
version 4.*x* (ARCtangent-A4 compiler), 78
virtual
  base members, 100
  bases
    and embedded C++, 118
    enabling (C++), 118
    pointers, 263
    storage, 261
  function tables, 102, 260
    always generate, 91
    generate when defined, 92
    make public and defined, 100
  functions, 102
virtual, 261
Viterbi butterfly dual-word instruction *see* dual Viterbi butterfly instruction
void
  assumed if no C++ arguments, 98
  and older code, 340
volatile
  data
    functions for copying, 223
volatile, 223, 253, 258
  accessing objects, 258
  data, 122, 124
    and MQX, 223
  variables, 104
    separate section, 223
VP rules, 124
  warn on possible difference, 125
VRaptor, 50

W

wait for license, 125
warn on undefined **#if** and **#elif**, 122
warnings
  add count to compiler return code, 75
  constraint, 336
  disabling, 145, 339
  disabling by number, 72, 143, 339
  displaying numbers, 105
  enabling, 145, 339
  enabling as errors, 144, 339
  enabling by number, 144, 339
  levels, 148, 338, 339
    specifying, 83
  messages, 338
    controlling, 339
    specifying format, 78, 337
  PCC, 108
  popping on/off specification, 145, 339
  reporting as errors, 339
  superfluous, 340
wchar_t size, 78
weak global symbol, 148
well-behaved
  pointers, 88
wide
  character
    constant, 249
    size, 78
  inlining, 188, 189
    and public names, 189
widening an operand, 124
word alignment, 86
wrapper files, 60, 71
wrapping
  C header files for C++, 98
  non-exception-aware files, 71
writing makefile dependencies
  to a `.u` file, 38, 77
  to a file, 38, 68
  to standard output, 38, 68

X

XMAC with multiply extension, 48
XMAC, 24-bit, 50
XMAC, dual 16-bit, 50
XY memory, 51