

C/C++ Compiler

*User's Guide and Reference
for the 68000 and ColdFire Families*

206889

102554-003

TRADEMARKS

The following names are trademarks, registered trademarks, and service marks of Mentor Graphics Corporation:

3D Design™, A World of Learning(SM), ABIST™, Arithmetic BIST™, AccuPARTner™, AccuParts™, AccuSim®, ADEPT™, ADVance™ MS, ADVance™ RFIC, AMPL™, Analog Analyst™, Analog Station™, AppNotes(SM), ARTgrid™, ArtRouter™, ARTshape™, ASICPlan™, ASICVector Interfaces™, Aspire™, Assess2000(SM), AutoCells™, AutoDissolve™, AutoFilter™, AutoFlow™, AutoLib™, AutoLinear™, AutoLink™, AutoLogic™, AutoLogic BLOCKS™, AutoLogic FPGA™, AutoLogic VHDL®, AutomotiveLib™, AutoPAR®, AutoTherm®, AutoTherm Duo™, AutoThermMCM™, Autowire Station™, AXEL™, AXEL Symbol Genie™, Bist Architect™, BIST Compiler(SM), BIST-In-Place(SM), Block Station®, Board Architect™, Board Designer™, Board Layout™, Board Process Library™, Board Station®, Board Station Consumer™, BOLD Administrator™, BOLD Browser™, BOLD Composer™, BSDArchitect™, BSPBuilder™, Buy on Demand™, Cable Analyzer™, Cable Station™, CAECO Designer™, CAEFORM™, Calibre®, CAM Station™, Capture Station®, Celaro™, Cell Builder™, Cell Station®, CellFloor™, CellGraph™, CellPlace™, CellPower™, CellRoute™, Centricity™, CEOC™, CheckMate™, CheckPlot™, CHEOS™, Chip Station®, ChipGraph™, ChipLister™, Circuit PathFinder™, Co-Verification Environment™, Co-Lsim™, CodeVision™, CommLib™, Concurrent Board Process(SM), Concurrent Design Environment™, Connectivity Dataport™, Continuum™, Continuum Power Analyst™, CoreAlliance™, CoreBIST™, Core Builder™, Core Factory™, CTIntegrator™, DataCentric Model™, Datapath™, Data Solvent™, dBUG™, Design Architect®, Design Architect Elite™, Design Evolution™, Design Only™, Design Manager™, Design Station®, Design Viewer™, DesignWide™, DFTAdvisor™, DFTArchitect™, DFTCompiler™, DFTInsight™, DirectConnect(SM), Documentation Station™, DSS (Decision Support System)™, Eldo™, ePartners™, EParts®, E3LCable™, EDGE (Engineering Design Guide for Excellence)(SM), Empowering Solutions™, Engineer's Desktop™, EngineerView™, ENRead™, ENWrite™, ESim™, Expert2000(SM), Explorer CAECO Layout™, Explorer CheckMate™, Explorer Datapath™, Explorer Lsim™, Explorer Lsim-C™, Explorer Lsim-S™, Explorer Ltime™, Explorer Schematic™, Explorer VHDLsim™, ExpressI/O™, FabLink™, Falcon™, Falcon Framework®, FastScan™, FastStart™, FastTrack Consulting(SM), First-Pass Design Success™, First-pass success(SM), FlexSim™, FlexTest™, FDL (Flow Definition Language)™, FlowXpert™, FORMA™, FormalPro™, FPGA Advantage™, FPGAAdvisor™, FPGA Builder™, FPGASim™, FPGA Station™, FrameConnect™, Galileo®, Gate Station®, GateGraph™, GatePlace™, GateRoute™, GDT®, GDT Core®, GDT Designer™, GDT Developer™, GENIE™, GenWare™, Geom Genie™, Hierarchy Injection™, HIC rules™, Hardware Modeling Library™ (HML), HotPlot®, Hybrid Designer™, Hybrid Station®, IC Design Station™, IC Designer™, IC Layout Station™, IC Station®, ICbasic™, ICblocks™, ICcheck™, ICcompact™, ICdevice™, ICextract™, ICGen™, ICGraph™, ICLink™, IClist™, ICplan™, ICRT Controller Lcompiler™, ICrules™, ICrules™, ITrace™, ICverify™, ICview™, ICX Custom Model™, ICX Custom Modeling™, ICX Project Modeling™, ICX Standard Library™, IDEA Series™, Idea Station®, INFORM®, IFX™, Inexia™, Integrated Product Development®, Integration Tool Kit™, INTELLITEST®, Interactive Layout™, Interconnectix™, IntraStep(SM), Inventra™, Inventra Soft Cores™, IP Engine™, IP Evaluation Kit™, IP Factory™, IP-PCB™, IP QuickUse™, IPSim™, IS_Analyzer™, IS_Floorplanner™, IS_MultiBoard™, IS_Optimizer™, IS_Synthesizer™, ISD Creation(SM), ITK™, It's More than Just Tools(SM), Knowledge-Sourcing(SM), LAYOUT™, LNL™, LBIST™, LBIST Architect™, Lc™, Lcore™, Leaf Cell Toolkit™, Led™, Leonardo™, LED LAYOUT™, LIBRARIAN™, Library Builder™, Logic Analyzer on a Chip(SM), Logic Builder™, Logical Cable™, LogicLib™, logio™, Lsim™, Lsim DSM™, Lsim-Gate™, Lsim Net™, Lsim Power Analyst™, Lsim-Review™, Lsim-Switch™, Lsim-XL™, Mach PA™, Mach TA™, Manufacture View™, Manufacturing Advisor™, Manufacturing Cable™, MaskCompose™, MaskPE®, Master Model®, MBIST™, MBISTArchitect™, Mc™, MCM Station®, MDV™, MegaFunction™, Memory Builder™, Memory Builder Conductor™, Memory Builder Mozart™, Memory Designer™, Memory Model Builder™, Mentor™, Mentor Graphics®, Mentor Graphics Support CD(SM), Mentor Graphics SupportBulletin(SM), Mentor Graphics SupportCenter(SM), Mentor Graphics SupportFax(SM), Mentor Graphics SupportNet-Email(SM), Mentor Graphics SupportNet-FTP(SM), Mentor Graphics SupportNet-Telnet(SM), MicroPlan™, MicroRoute™, Microtec®, Mixed-Signal Pro™, ModelEditor™, ModelSim®, ModelSim LNL™, ModelStation®, ModelViewer™, ModelViewerPlus™, Monet®, Mslab™, Msview™, MS Analyzer™, MS Architect™, MS-Express™, MSIMON™, MTPI(SM), Nanokernel®, NetCheck™, NETED™, OpenDoor(SM), Opsim™, OutNet™, PACKAGE™, PARADE™, ParallelRoute-Autocells™, ParallelRoute-MicroRoute™, PathLink™, Parts SpeciaList™, PCB-Gen™, PCB-Generator™, PCB IGES™, PCB Mechanical Interface™, PDLsim™, Personal Learning Program™, Physical Cable™, Physical Test Manager.SITE™, PLA Lcompiler™, PLDSynthesis™, PLDSynthesis II™, Power Analyst™, PowerAnalyst Station™, Pre-Silicon™, ProjectXpert™, ProtoBoard™, ProtoView™, QNet™, QualityIBIS™, QuickCheck™, QuickFault™, QuickGrade™, QuickHDL™, QuickHDL Express™, QuickHDL Pro™, QuickPart Builder™, QuickPart Tables™, QuickParts™, QuickPath™, QuickSimII™, QuickStart™, QuickUse™, QuickVHDL®, RAM Lcompiler™, RC-Delay™, RC-Reduction™, RapidExpert™, REAL Time Solutions™, Registrar™, Reinstatement 2000(SM), Reliability Advisor™, Reliability Manager™, REMEDI™, Renoir™, RF Architect™, RF Gateway™, RISE™, ROM Lcompiler™, RTL X-Press™, Satellite PCB Station™, ScalableModels™, Scalable Verification™, SCAP™, Scan-Sequential™, Scepter™, Scepter DFF™, Schematic View Compiler, SVC™, Schemgen™, SDF™ (Software Data Formatter), SDL2000 Lcompiler™, Seamless®, Seamless Co-Designer™, Seamless CVE™, Seamless Express™, Selective Promotion™, SignalMask OPC™, Signature Synthesis™, Simulation Manager™, SimExpress™, SimPilot™, SimView™, SiteLine2000(SM), SmartMask™, SmartParts™, SNX™, SOS Initiative™, Source Explorer™, Spectra®, SpiceNet™, SST Velocity®, Standard Power Model Format (SPMF)™, Structure Recovery™, Super C™, Super IC Station™, Support Services BaseLine(SM), Support Services ClassLine(SM), Support Services Latitudes(SM), Support Services OpenLine(SM), Support Services PrivateLine(SM), Support Services SiteLine(SM), Support Services TechLine(SM), Support Services RemoteLine(SM), Symbol Genie™, Symbolscript™, SYMED™, System Architect™, System Design Station™, System Modeling Blocks™, Systems on Board Initiative™, Target Manager™, Tau™, TeraCell™, TeraPlace™, TeraPlace-GF™, TechNotes™, Test Station®, Test Structure Builder™, The Ultimate Site For HDL Simulation™, TimeClosure™, Timing Builder™, TNX™, ToolBuilder™, TrueTiming™, Vlog™, V-Express™, V-Net™, VHDLnet™, VHDLwrite™, Verinex™, ViewCreator™, ViewWare®, Virtual Library™, Virtual Target™, Virtual Test Manager:TOP™, VR-Process(SM), VRTX®, VRTXmc™, VRTXoc™, VRTXsa™, VRTX32®, Waveform DataPort™, We Make TMN Easy™, WorkXpert™, xCalibre™, xCalibrate™, Xconfig™, Xpert™, Xpert API™, XpertBuilder™, Xpert Dialogs™, Xpert Profiler™, XRAY®, XRAY MasterWorks®, XSH®, Xtrace®, Xtrace Daemon™, Xtrace Protocol™, Zeelan®, Zero Tolerance Verification™, Zlibs™

All other trademarks mentioned in this document are trademarks of their respective owners.

RESTRICTED RIGHTS LEGEND

U.S. Government Restricted Rights. The software programs and related documentation have been developed entirely at private expense and are commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which the software programs and documentation were obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Mentor Graphics Corporation
880 Ridder Park Dr.
San Jose, CA 95131

Copyright © 1987-2000 Mentor Graphics Corporation. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Mentor Graphics Corporation.

IMPORTANT - USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE.

This license is a legal "Agreement" concerning the use of Software between you, the end user, either individually or as an authorized representative of the company purchasing the license, and Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Mentor Graphics (Singapore) Private Limited, and their majority-owned subsidiaries (collectively "Mentor Graphics"). **USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT.** If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within 10 days after receipt of Software and receive a full refund of any license fee paid.

END USER LICENSE AGREEMENT

1. GRANT OF LICENSE. The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, and documentation ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics or its authorized distributor grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' then-current standard policies, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be communicated and technically implemented through the use of authorization codes or similar devices); (c) eligibility to receive updates, modifications, and revisions; and (d) support services provided. Current standard policies are available upon request.

2. ESD SOFTWARE. If you purchased a license to use embedded software development ("ESD") Software, Mentor Graphics or its authorized distributor grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of Mentor Graphics' real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

3. BETA CODE.

3.1 Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

3.2 If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and test-

ing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3 You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or makes during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

4. RESTRICTIONS ON USE. You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than your employer's employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by the European Union Software Directive or local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it without Mentor Graphics' prior written consent. The provisions of this section shall survive the termination or expiration of this Agreement.

5. LIMITED WARRANTY.

5.1 Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LOANED TO YOU FOR A LIMITED TERM OR AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

5.2 THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6. LIMITATION OF LIABILITY. EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE STATUTE OR REGULATION, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR

GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

7. LIFE ENDANGERING ACTIVITIES. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH SUCH USE.

8. INFRINGEMENT.

8.1 Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright in the United States, Canada, Japan, Switzerland, Norway, Israel, Egypt, or the European Union. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the claim, provided that you: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the claim; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the claim.

8.2 If an infringement claim is made, Mentor Graphics may, at its option and expense, either (a) replace or modify Software so that it becomes noninfringing, or (b) procure for you the right to continue using Software. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

8.3 Mentor Graphics has no liability to you if the alleged infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you design or market; (f) any Beta Code contained in Software; or (g) any Software provided by Mentor Graphics' licensors which do not provide such indemnification to Mentor Graphics' customers.

8.4 THIS SECTION 8 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

9. TERM. This Agreement remains effective until expiration or termination. This Agreement will automatically terminate if you fail to comply with any term or condition of this Agreement or if you fail to pay for the license when due and such failure to pay continues for a period of 30 days after written notice from Mentor Graphics. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

10. EXPORT. Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export in any manner any Software or direct product of Software, without first obtaining all necessary approval from appropriate local and United States government agencies.

11. RESTRICTED RIGHTS NOTICE. Software has been developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

12. THIRD PARTY BENEFICIARY. For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth in this Agreement.

13. CONTROLLING LAW. This Agreement shall be governed by and construed under the laws of Ireland if the Software is licensed for use in Israel, Egypt, Switzerland, Norway, South Africa, or the European Union, the laws of Japan if the Software is licensed for use in Japan, the laws of Singapore if the Software is licensed for use in Singapore, People's Republic of China, Republic of China, India, or Korea, and the laws of the state of Oregon if the Software is licensed for use in the United States of America, Canada, Mexico, South America or anywhere else worldwide not provided for in this section.

14. SEVERABILITY. If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

15. MISCELLANEOUS. This Agreement contains the entire understanding between the parties relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement which are physically signed by you and an authorized agent of Mentor Graphics. This Agreement may only be modified by a physically signed writing between you and an authorized agent of Mentor Graphics. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.

Rev. 10/99

REV.	REVISION HISTORY	DATE	APPD.
-001	Merged MCC68K (100131-018) and CCC68K (100704-008) manuals. Updated for software versions 5.0 (C compiler) and 3.0 (C++ compiler).	9/98	S. F.
-002	Updated for MCC68K/CF 5.1 and CCC68K/CF 3.1.	12/98	M. W.
-003	Updated for C compiler version 5.2 and C++ compiler version 3.2.	7/00	D.G.

Contents

Preface

About This Manual	xxiii
Microprocessor References	xxiv
Related Publications	xxiv
Notational Conventions	xxvi
Questions and Suggestions	xxvi

1 Introduction

C/C++ Compilers	1-1
Compiler Features	1-1
ASM68K and ASMCF Assemblers	1-2
LNK68K and LNKCF Linkers	1-3
LIB68K and LIBCF Object Module Librarians	1-3
XRAY Debugger (Optional)	1-3
Data Flow	1-4

2 Using the Compilers

Invoking the Compilation Driver	2-1
Command Line Syntax	2-1
Filename Extensions	2-2
Input and Output File Locations	2-2
Environment Variables	2-3
Setting Environment Variables for UNIX	2-4
Setting Environment Variables for DOS Shell under Windows	2-5
Special Considerations for Windows	2-5

3 Using Command Line Options

About Command Line Options	3-1
Command Line Options — Summary	3-1
Command Line Options — Extended Descriptions	3-18
Enabling Microtec Compiler Extensions	3-19
Setting Compatibilities	3-20
ANSI Compatibility	3-20
Object Code Compatibility	3-20
Controlling Include File Search Paths	3-20
Add Search Path for Nonstandard Include Files	3-20
Use Precompiled Headers	3-21
Change Search Path Order for Include Files	3-22
Add Search Path for Standard Include Files	3-22

Controlling Listing Files	3-23
Specify Format of Listing Files	3-23
Generate Listing File	3-24
Defining Preprocessor Names	3-24
Controlling the Preprocessor and Its Output	3-25
Driver Options	3-25
Produce Object File Only	3-25
Read Options From File (not a driver option)	3-26
Pass Command File to Linker	3-26
Save Assembly File	3-26
Name the Output File	3-26
Produce Code for Specified Processor (not a driver option)	3-27
Pass Default Library to Linker	3-29
Produce Assembler Source File	3-29
Check Program Syntax	3-29
Control Verbose Output Information	3-30
Perform Extra Checking (not a driver option)	3-30
Pass Options to Tools	3-30
Modify Alignment (not a driver option)	3-31
Modifying Naming Conventions	3-32
Generating Floating-Point Processor Instructions	3-33
Supporting ColdFire Multiple and Accumulator Unit	3-33
Producing Debug Information	3-33
Support HP 64000	3-35
Controlling Diagnostic Messages	3-35
Redirect Diagnostics to File (Windows only)	3-35
Redirect Diagnostic Messages	3-36
Change Diagnostic Message Severity	3-36
Handling Uninitialized Global Data	3-38
Optimizing Code	3-38
Generate Optimizer Information Messages	3-38
List of Code Optimizations	3-39
Producing Minor Code Generation Variations	3-47
Generating Position-Independent Code and Data	3-51
Choose Address Mode for Sections	3-53
Name the Sections	3-55
Enabling C++ Extensions	3-58
Suspension of Cleanup	3-58
Control C++ Virtual Table Generation	3-58
Specify Language	3-59
Control Template Instantiation	3-59
Enable C++ Features	3-60
Accept Nonstandard C++ File Suffix	3-62

Using Options	3-62
Option Combinations	3-62
ANSI Extensions (C only)	3-63
Using <code>__STDC__</code>	3-63
Function Prototyping	3-63
ANSI and Floating-Point Variables	3-64
Microtec Compiler Extensions	3-65
typeof() Operator	3-66
Exception Handling	3-66
Customizing the Compilation Driver	3-66
Pragmas and Options	3-67
Locating Header Files	3-69
Finding Source Files	3-69
Determining Option Defaults	3-70
Producing Listing Files	3-70
Command Line Examples	3-71
4 Microtec C++ Compiler Extensions	
Keywords	4-1
interrupt	4-1
packed, unpacked	4-3
Include File Optimization	4-4
asm Support	4-5
asm Examples	4-6
Assigning asm to a Variable	4-7
Returning a Typed Value	4-7
Using #define for Readability	4-8
Variable Names Inside asm	4-8
#pragma asm Versus asm	4-10
Considerations for Assembler In-lining	4-10
5 Using Libraries	
C++ I/O Library	5-1
UNIX Level 1+ Elements	5-2
I/O Buffering	5-3
Buffered I/O	5-3
Unbuffered I/O	5-4
Unit Buffered I/O	5-4
C++ Function Groups and Include Files	5-4
mriehs.h File	5-4
new.h File	5-6
Library Extensions	5-6
C Function Groups and Include Files	5-6
mriext.h	5-6

Non-Reentrant Extensions	5-8
I/O System Functions	5-9
Intrinsic System Functions	5-10
Library Function Definitions	5-12
alloca — Allocates Data Space on Stack.....	5-13
chdir — Changes Working Directory	5-14
close — Closes a Specified File.....	5-15
connect — Initiates a Socket Connection on the Host System.....	5-16
creat — Creates a Specified File.....	5-17
_cxxfini — Calls Static Destructors to Destroy Static Objects	5-18
_ehs_cleanup_area — Cleans up the Exception Handling State Buffer.....	5-19
_ehs_init_area — Initializes the Exception Handling State Buffer	5-20
_ehs_restore_from_area — Restores the Exception Handling State	5-21
_ehs_save_restore_area — Saves and Restores the Exception Handling State	5-22
_ehs_save_size — Returns Memory Size Required to Save Exception Handling State.....	5-23
_ehs_save_to_area — Saves the Exception Handling State	5-24
fprintf — Provides Formatted Print to Standard Error	5-25
_exit — Terminates a Program	5-26
fileno — Gets File Descriptor.....	5-27
ftoa — Converts Floating-Point Number to ASCII String	5-28
getcwd — Returns Current Working Directory.....	5-29
getl — Reads a Long Integer From a Stream	5-30
getw — Reads a Short Integer From a Stream	5-31
isascii — Tests for an ASCII Character.....	5-32
itoa — Converts Integer to ASCII String	5-33
itostr — Converts Unsigned Integer to ASCII String.....	5-34
lseek — Sets the Current Location in a File	5-35
ltoa — Converts Long Integer to ASCII String.....	5-36
ltostr — Converts Unsigned Long Integer to ASCII String	5-37
_main — Calls Static Constructors to Create Static Objects.....	5-38
max — Returns the Greater of Two Values.....	5-39
memcpy — Copies Characters in Memory	5-40
memset — Clears Memory Bytes	5-41
min — Returns the Lesser of Two Values.....	5-42
open — Opens a Specified File	5-43
_pclose — Closes a Pipe from a Process	5-45
_popen — Opens a Pipe to a Process.....	5-46
__pure_virtual_function_called — Handles Abnormal Conditions With Pure Virtual Functions	5-47
putl — Writes a Long Integer to a Stream.....	5-48
putw — Writes a Short Integer to a Stream.....	5-49
read — Reads Bytes From a Specified File.....	5-50

sbrk — Allocates Memory Space	5-51
set_new_handler — Sets the Memory Exception Handler for operator new	5-52
set_terminate — Sets the Termination Function to Terminate Execution.....	5-53
set_unexpected — Sets the Function to Handle Unexpected Exceptions.....	5-54
socket — Creates a Communication Endpoint on the Host System.....	5-55
stat — Gets Information about File	5-56
swab — Swaps Odd and Even Bytes in Memory	5-57
sys_in — Reads Characters from Standard Input Window	5-58
sys_out — Sends Characters to Standard Output Window	5-59
sys_stat — Checks for Input Characters Waiting	5-60
terminate — Calls abort() or the Last Function Set.....	5-61
toascii — Converts a Byte to ASCII Format	5-62
_tolower — Converts Characters to Lower-case Without Argument Checking.....	5-63
_toupper — Converts Characters to Upper-case Without Argument Checking.....	5-64
unexpected — Calls terminate() or the Last Function Set.....	5-65
unlink — Unlinks a Filename	5-66
write — Writes Bytes to a Specified File	5-67
zalloc — Allocates Data Space Dynamically	5-68

6 C Library Customizer

When to Create a Custom Library	6-1
Directories in the C Library Customizer	6-1
Windows System Requirements	6-2
Using the C Library Customizer	6-2
Step 1: Creating a Custom Configuration File	6-3
Step 2: Building a Custom Library	6-6
Step 3: Testing a Custom Library	6-7
Assessing Test Results	6-9
Debugging Custom Libraries	6-9
Using Custom Libraries	6-10

7 Optimizations

General Optimizations	7-1
Algebraic Simplification	7-1
Redundant Code Elimination	7-2
Strength Reduction	7-2
Global Optimizations	7-2
Dead Code Elimination	7-2
Factorization	7-4
Global Constant Propagation	7-4
Global Copy Propagation	7-5
Global Value Propagation	7-5

Register Allocation	7-6
Unused Definition Elimination	7-6
Loop Optimizations	7-6
Array Operator Synthesis (C complier only)	7-7
Loop Invariant Code Optimization	7-7
Loop Rotation	7-8
Strength Reduction and Index Simplification	7-9
Local Optimizations	7-10
Common Subexpression Elimination	7-10
Constant Folding	7-10
Generating Code for switch Statements	7-10
Redundant Load and Store Elimination	7-11
Jump Optimizations	7-11
Branch Tail Merging	7-11
Code Hoisting	7-12
Cross-Jump Optimization	7-13
Multiple Jump Optimization	7-14
Redundant Jump Elimination	7-14
Short/Long Displacement Optimization	7-15
Function In-Lining	7-15
inline Keyword	7-16
Instruction Scheduling	7-17
Machine-Dependent Optimizations	7-17
Generating Code for Function Prologue and Epilogue	7-17
In-Line Library Function Expansion	7-17
Grouping Stack Adjust Instructions	7-18
Indexing Arrays	7-18
Char Operations Where Possible	7-19

8 Template Instantiation

Templates Versus Macros	8-1
In-Line Macro	8-1
Macro Function	8-2
Function Template	8-2
Compile-Time Template Instantiation	8-3
Declare the Class Template Stack in stack.h	8-4
Define the Class Template in stack.def	8-4
Use Template Class in Source File user1.cc	8-5
Automatic Instantiation	8-5
Scope of Template Instances	8-5
Manual Template Instantiation	8-6
Specialization	8-7

9 Precompiled Header Files

Precompiled Header File Processing	9-1
Using Options With Precompiled Header Files	9-3
Using the Preprocessor With Precompiled Headers	9-4
Precompiled Header File Usage Guide	9-4

10 Interlanguage Calling

Parameter Passing	10-1
Pushing Parameters	10-1
Setting Short Integers	10-1
Implicit Parameter for Structure/Union Return Value	10-3
Alignment of Structure/Union in Parameter Area	10-4
Function Return Values	10-4
Popping Parameters	10-5
Register and Stack Usage With Functions	10-5
Stack Frames	10-5
The Prologue	10-6
Local Variables in the Prologue	10-7
The Epilogue	10-7
Assembly Language Interfacing	10-8
Assembly Language Routine Example	10-9
Variable References from Assembly Routines	10-10
Interrupt Handlers	10-11
Defining a Function	10-12
Calling C Functions From C++	10-12
Passing Arguments	10-13
Calling C++ Functions From C	10-14
Overloaded Functions	10-14
Member Functions	10-14
Exception Handling Between C++ and C	10-18
Exception Handling Between C++ and Assembly	10-18

11 Internal Data Representation

Storage Layout	11-1
Data Type Summary	11-2
Pointers	11-3
Type Conversion	11-4
Mixed Operands	11-5
Type Casting	11-6
Function Operations	11-6
Passing Prototyped Parameters Smaller Than int	11-6
Function Declaration With interrupt Keyword	11-6
Structure Operations	11-7
Structure Alignment	11-7

Determining Structure Size	11-8
Bit Fields	11-9
Alignment and Packing	11-12
Alignment of Bit Fields	11-12
Aggregates	11-14
Size of Aggregates	11-15
Packed Structures	11-16
Structure Padding	11-17
Structure Alignment When Default Alignments Differ	11-21
Packed Enumeration Types	11-22
Tips About Packing	11-23
Allocation of Variables	11-25
Register	11-26
Local Variables	11-26
Static Variables	11-26
Memory Allocation	11-26

12 Run-Time Organization

Code Organization	12-1
Changing Default Addressing	12-3
Compiler-Generated Sections	12-3
code Section (Type CODE)	12-5
strings Section (Type CODE)	12-5
literals Section (Type CODE)	12-5
const Section (Type CODE)	12-5
pixinit Section (Type CODE)	12-5
cxx_edt Section (Type CODE)	12-5
cxx_rtti Section (Type DATA)	12-6
ioports Section (Type DATA)	12-6
vars Section (Type DATA)	12-6
zerovars Section (Type DATA)	12-6
tags Section (Type DATA)	12-7
syshost Section (Type DATA)	12-7

13 Embedded Environments

Considerations for Embedded Systems	13-1
In-Line Assembly	13-2
Examples	13-3
Features of Assembler In-Lining	13-4
Assigning asm to a Variable	13-4
Returning a Typed Value	13-5
Using #define for Readability	13-5
Variable Names Inside asm	13-5
#pragma asm or asm	13-7

Considerations for Assembler In-Lining	13-8
Addressing	13-9
Direct Memory Addressing	13-9
Calling a Function at an Absolute Address	13-9
Reentrant Code	13-10
Calling Non-Reentrant Functions from Multiple Threads	13-11
Multitasking/Multi-Threaded Environments	13-11
First Approach	13-13
Second Approach	13-16
Position-Independent Code and Data	13-18
Position-Independent Versus Position-Dependent	13-18
Compiler Considerations	13-19
Programmer Considerations	13-19
Assembler and Linker Considerations	13-22
Absolute Versus Register-Relative Addressing	13-23
Absolute Addressing	13-23
PC-Relative Addressing	13-25
User-Modified Routines	13-27
User-Modified Routines for Embedded Systems	13-27
In Character Routine	13-27
Out Character Routine	13-27
Start Routine	13-28
Exit Routine	13-29
Initialization Routine	13-29
Heap Management Routine	13-29
System Functions and SysHost feature	13-29
Removing Unneeded I/O Support	13-31
Removing Unneeded Floating-Point Support	13-32
Reserving a Register	13-32
Shared Program	13-33
Shared Data	13-34
System Data	13-35
Special Purposes	13-36
Data Initialization	13-37
Saving Initialized Variables in ROM	13-37
Using the Linker	13-38
Default Sections	13-38
Libraries	13-38
Messenger Symbols	13-38
INITDATA Command	13-38
Linker Command Example	13-39
Linker Command Example (ROM-Based System)	13-41
Interrupt Handlers	13-43

Static Initialization and Termination	13-44
Static Constructor	13-44
Static Destructor	13-44
initfini Section	13-44
pixinit Section	13-47
I/O Static Initialization and Termination	13-48
Exception Handling	13-48
Type Database	13-49
Save/Restore Functions	13-49

14 Compiler Output Files

Assembly Source File	14-1
Advantages of Producing an Assembly File	14-1
Variable Names	14-1
External and Public Variable Names	14-1
Static Variable Names	14-2
Line Numbers	14-2
Code and Data Sections	14-3
Compiler Output Listing	14-3

Appendix A: Error Statements

Message Format	A-1
Message Severity Levels	A-2
Error Position Marker	A-3
Suppressing Error Messages	A-4
Reporting Problems	A-6
Preparing a Test Case	A-6
Calling Technical Support	A-6
C Compiler Error Messages	A-6
C++ Compiler Messages	A-73
Expanded Descriptions	A-105

Appendix B: Glossary	B-1
----------------------------	-----

Figures

Figure 1-1.	Cross Environment Data Flow Through the Microtec Toolkit.....	1-5
Figure 5-1.	C++ I/O Levels	5-2
Figure 10-1.	Parameter Area of the Stack	10-1
Figure 10-2.	Parameter Area and Short Integers	10-2
Figure 10-3.	Parameter and Return Value Areas	10-3
Figure 10-4.	Parameter Area and Structure/Union	10-4
Figure 10-5.	C/C++ Interlanguage Calling	10-16
Figure 10-6.	Example of an Assembly Function in an Exception Handling Call Chain.....	10-19
Figure 11-1.	Memory Layout	11-1
Figure 11-2.	Loading Dependent on Processor Type	11-2
Figure 11-3.	Byte Ordering in Words	11-2
Figure 11-4.	Sample Bit Field Allocation (Over 8 Bytes)	11-11
Figure 11-5.	Sample Bit Field Allocation (Over 2 Bytes)	11-11
Figure 11-6.	Sample Packed Struct Bit Field Allocation (Over 2 Bytes)	11-12
Figure 11-7.	Packing Bit Fields	11-13
Figure 11-8.	Signed Bit Fields	11-14
Figure 11-9.	Packing of Aggregates (Big-Endian)	11-15
Figure 11-10.	Packing of Aggregates (Little-Endian)	11-15
Figure 11-11.	Packing of Aggregates (Big-Endian)	11-16
Figure 11-12.	Unpacked and Packed Structures (Big-Endian)	11-17
Figure 11-13.	Unpacked Structure: struct s	11-21
Figure 11-14.	Structure Alignment Example.....	11-22
Figure 11-15.	Structure three_tight_bytes	11-23
Figure 11-16.	Sample Data Structure	11-25
Figure 13-1.	Memory Configuration for Multi-Threaded Environments (Approach 1)	13-15
Figure 13-2.	Memory Configuration for Multi-Threaded Environments (Approach 2)	13-17
Figure 13-3.	Position-Independent Code (Example 1)	13-20
Figure 13-4.	Position-Independent Data (Example 2)	13-22
Figure 13-5.	Absolute Addressing for Code and Data	13-23
Figure 13-6.	Absolute Addressing Example	13-24
Figure 13-7.	PC-Relative Addressing.....	13-25
Figure 13-8.	PC-Relative Addressing Example	13-26
Figure 13-9.	Shared Programs	13-33
Figure 13-10.	Shared Data	13-34
Figure 13-11.	System Data	13-35
Figure 13-12.	Reserved Register as a Pointer to Data	13-36
Figure 13-13.	Reserved Register as a Storage Area	13-36
Figure 13-14.	Memory Configuration	13-40
Figure 13-15.	Memory Configuration (ROM Based)	13-42

Figure 13-16. initfini Section Before Linking	13-45
Figure 13-17. initfini Section After Linking	13-46
Figure 13-18. C++ Initialization Call Sequence	13-46
Figure 13-19. C++ Termination Call Sequence	13-47
Figure 13-20. pixinit Section before Linking	13-48

Tables

Table P-1.	Notational Conventions	xxvi
Table 2-1.	Default Filename Extensions	2-2
Table 2-2.	Default Environment Variables and Defaults for 68000 Processors	2-3
Table 3-1.	UNIX/Windows Command Line Option Summary	3-2
Table 3-2.	Command Line Option Function Categories	3-18
Table 3-3.	Processor Identification	3-27
Table 3-4.	Interaction of -Og and -OR Options	3-44
Table 3-5.	UNIX/Windows Allocation of Code and Data	3-53
Table 3-6.	Common Option Combinations	3-62
Table 3-7.	Keywords (Microtec Compiler Extensions)	3-65
Table 5-1.	Exception Handling Functions in mriehs.h Include File	5-5
Table 5-2.	Errors During Exception Handling	5-5
Table 5-3.	Functions Associated With operator new	5-6
Table 5-4.	mriext.h Functions	5-7
Table 5-5.	mriext.h Macros	5-8
Table 5-6.	Non-Reentrant Extensions	5-8
Table 5-7.	System Functions	5-9
Table 5-8.	Intrinsic System Functions	5-10
Table 5-9.	Mode Values for the open Function	5-43
Table 6-1.	C Library Customizer Preprocessor Symbols	6-4
Table 10-1.	Instructions for Removing Bytes from Stack	10-6
Table 11-1.	C/C++ Scalar Data Types	11-3
Table 11-2.	Complex (Aggregate) Types	11-4
Table 11-3.	Mixed Operand Conversion	11-5
Table 11-4.	Integral Promotion	11-5
Table 11-5.	Natural Boundary Alignment	11-19
Table 11-6.	Effect of Compiler Options on Structure Layout	11-22
Table 12-1.	Sections Generated by the Compiler	12-2
Table 12-2.	Example Modules	12-3
Table 12-3.	Sections Contained in module1.o and module2.o	12-4
Table 12-4.	Sections Contained in the Executable File	12-4
Table 13-1.	Position-Independent Code Options	13-18
Table 13-2.	Absolute Versus Register-Relative Addressing	13-23
Table 13-3.	Commands for Absolute Addressing Example	13-25
Table 13-4.	Commands for PC-Relative Addressing Example	13-26
Table A-1.	Message Severity Levels	A-2
Table A-2.	Using Compiler Options to Suppress Diagnostic Messages	A-4
Table A-3.	C++ Compiler Error Messages	A-73

The Embedded Software Division of Mentor Graphics Corporation provides tools that can be used individually to enhance existing development environments or together to provide a complete and highly integrated embedded software development solution.

About This Manual

This manual serves as a user's guide and reference for the Microtec C and C++ compilers for the 68000 and ColdFire families of microprocessors. It is organized as follows:

- Chapter 1, *Introduction*, describes the components of the compilers.
- Chapter 2, *Using the Compilers*, describes how to begin using the compilers on a UNIX system or a DOS shell under a Windows operating system.
- Chapter 3, *Using Command Line Options*, contains a summary of the command line options available, and in-depth explanations on how to use each option.
- Chapter 4, *Microtec C++ Compiler Extensions*, describes the Microtec extensions to the C++ language.
- Chapter 5, *Using Libraries*, explains how to use the libraries distributed with the compilers, and contains a list of available library functions.
- Chapter 6, *C Library Customizer*, describes how to build and use a custom library created from a general distribution C library.
- Chapter 7, *Optimizations*, describes the various optimizations that the C and C++ compilers can perform.
- Chapter 8, *Template Instantiation*, details the fundamentals of Microtec C++ template instantiation.
- Chapter 9, *Precompiled Header Files*, explains how to create and use precompiled header files to shorten C++ code compilation time.
- Chapter 10, *Interlanguage Calling*, provides information on interlanguage calling and exception handling between C, C++, and assembly language.

- Chapter 11, *Internal Data Representation*, describes the internal data formats of C and C++ and the run-time organization of C and C++ programs for the 68000 and ColdFire families.
- Chapter 12, *Run-Time Organization*, describes the memory organization of C and C++ programs for the 68000 and ColdFire families.
- Chapter 13, *Embedded Environments*, provides instructions and tips on using the C/C++ compilers to create code for embedded systems.
- Chapter 14, *Compiler Output Files*, describes how to pass variable name and data type information from the compilers directly to the XRAY Debugger.

The *Appendix* section of this manual is organized as follows:

- Appendix A, *Error Statements*, describes error and diagnostic messages produced by the C and C++ compilers.
- Appendix B, *Glossary*, contains a list of terms used in this manual relevant to the compilers and their use.

Microprocessor References

The 68000 family of microprocessors includes the 68000, 68008, 68010, 68012, 68020, 68030, 68302, 68040, 68060, 68881, 68882, 68851, 68HC000, 68HC001, 68EC000, 68EC020, 68EC030, 68EC040, 68EC060, and the CPU32/CPU32+ families: 68330, 68331, 68332, 68333, 68340, 68349, 68360. The ColdFire family includes: 5102, 5200, 5202, 5204, 5206. This document refers to these microprocessors collectively as 68000 and ColdFire family microprocessors.

Related Publications

This documentation is written for the experienced program developer and assumes the developer has a working knowledge of the Motorola 68000 and ColdFire families of microprocessors. Although it provides several useful and informative program examples, this documentation does not describe the microprocessor itself. For such information, refer to the following Motorola publications:

- *Programmer's Reference Manual*, M68000PM/UD.
- *M68000 Family Resident Structured Assembler Reference Manual*, M68KMASM/D8.
- *68000 16/32-Bit Microprocessor User's Manual*, M68000UM/AD4.
- *68020 32-Bit Microprocessor User's Manual*, MC68020UM/AD.

- *68030 32-Bit Microprocessor User's Manual*, MC68030UM/AD.
- *MC68040 32-Bit Microprocessor User's Manual*, MC68040UM/AD.
- *MC68060 32-Bit Microprocessor User's Manual*, MC68060UM/AD.
- *68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD.
- *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD.
- *CPU32 Central Processor Unit Reference Manual*, CPU32RM/AD.
- *ColdFire MCF5102 User's Manual*, MCF5102UM/AD
- *MCF5200 ColdFire Family Programmer's Reference Manual*

For further information about the Microtec extended IEEE-695 format, refer to:

- *IEEE-695 Document*, Microtec, December 1992.

For information about other Microtec 68000 and ColdFire families toolkit components, refer to the following Microtec publications:

- *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.
- *Product Installation (UNIX or Windows Hosts)*

Notational Conventions

This manual uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Enclose a list from which you must choose an item.
[]	Square Brackets	Enclose optional items.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
	Vertical Bar	Separates alternative items in a list.
	Punctuation	Punctuation such as commas (,) and colons (:) must be entered as shown.
	Typewriter Font	Represents code or user input in interactive examples.
	<i>Italics</i>	Represents a descriptive item that should be replaced with an actual item.
	Bold	Represents elements that need to stand out from the main body of text.
->	Arrow	Refers to a sequence of windows. For example, Window1 -> Window2 -> Window3 refers to Window3, which is opened from Window2, which is opened from Window1.

Questions and Suggestions

Mentor Graphics is committed to providing its customers with quality software development and RTOS tools and support services. Our commitment continues beyond your purchase of the product throughout your development life cycle.

If you have questions or suggestions regarding this product, please contact your Mentor Graphics support representative. Contact numbers are listed on the back cover of this document.

Introduction 1

The Microtec C++ and ANSI C compilers are part of the Microtec Toolkit for the Motorola 68000 and ColdFire families of microprocessors. Other components of the toolkit include a relocatable cross assembler, a linker, and an object module librarian. The XRAY Debugger is available as an option.

The components of the toolkit are described in more detail in the following sections.

C/C++ Compilers

The Microtec MCC68K/CF and CCC68K/CF compilers help you develop efficient, portable, and easy-to-maintain programs for microprocessor-based systems. Between the two compilers, all of the standard features are supported.

The C compiler driver offers a subset of the capabilities of the C++ compiler driver, with the following differences:

- The C compiler driver can only compile C programs.
- The C compiler driver ignores or rejects options specific to C++.

Compiler Features

The Microtec C/C++ compiler package provides the following features:

- Supports traditional C and the C++ features described in *The Annotated C++ Reference Manual*, such as: templates, exception handling, multiple inheritance, overloading resolution, memberwise assignment and initialization, static member functions, abstract classes with pure virtual functions, pointer to member, and type-safe linkage
- Supports the latest ANSI C and C++ features
- Provides a compiler driver that invokes the C or C++ compiler, depending on the source language, and the assembler and linker, as needed
- Provides a simple interface to assembly language
- Compiles large C/C++ source modules
- Generates standard or optimized code for supported microprocessors
- Supports in-line assembly instructions, including referencing program variables by their C names

- Issues comprehensive warning messages
- Provides options to specify search paths for standard and nonstandard **#include** files
- Contains a run-time library that supports all ANSI C functions applicable to embedded systems and many standard C++ functions
- Generates reentrant code
- Locates code and constants in ROM and data in RAM
- Provides a fully ANSI-compatible C preprocessor
- Generates a listing file containing C or C++ code intermixed with assembly code
- Generates line number and symbolic debugging information for use with the XRAY Debugger
- Supports source-level debugging of optimized code with the XRAY Debugger
- Adds Microtec Compiler C extension support to the C and C++ languages
- Includes C++ I/O class libraries
- Integrates C++ name demangling support with the Microtec assembler and linker
- Supports Microtec Compiler extensions such as **packed** structures and **interrupt** procedures
- Loads only necessary library modules
- Supports pre-compilation of header files in C++ to save recompilation time

ASM68K and ASMCF Assemblers

The ASM68K and ASMCF Assemblers convert assembly language programs into relocatable object modules that conform to the IEEE-695 standard format. The object modules can be linked with other object modules or with libraries. The assemblers also process macros and conditional assembly statements and can generate cross-reference and standard symbol tables. Code and data can be placed in multiple named or numbered sections.

After assembly, LNK68K/LNKCF links the object modules, which can then be downloaded and run on any 68000 or ColdFire family microprocessor or the XRAY Debugger (if the modules are in the appropriate format).

The ASM68K/ASMCF Assemblers accept source program statements that are syntactically compatible with those accepted by assemblers for the 68000 and ColdFire families of microprocessors. The two assemblers behave identically.

LNK68K and LNKCF Linkers

The LNK68K and LNKCF Linkers combine relocatable object modules into a single, relocatable absolute object module that can be relinked with other modules. This feature is called incremental linking. The module can be in the following formats:

- Microtec Linker extended IEEE-695 format
- Motorola S-record format

If an input file is a library of object modules, the linker automatically loads any modules from that library that other named object modules reference. The linker produces a link map that shows the final location of all modules and sections and the final absolute values of all symbols. A cross-reference listing shows which modules refer to each global symbol.

You can run the linker either from the command line or in batch mode by using a command file. The linker reports unresolved external symbols and errors that occur at link time and prints these messages on the link map or on the screen.

LIB68K and LIBCF Object Module Librarians

The LIB68K and LIBCF Librarians create and maintain program libraries. A program library is a file containing relocatable object modules. The librarian lets you automatically load frequently used object modules without your having to know the specific names and characteristics of the modules.

The LIB68K and LIBCF Librarians let you format and organize library files that will later be used by the LNK68K and LNKCF Linkers.

XRAY Debugger (Optional)

The XRAY Debugger lets you monitor and control execution of programs at the source and assembly levels from a window-oriented user interface. You can examine or modify the value of program variables using the same terms, definitions, and structures you defined in the original source code.

The debugger gives you complete interactive control of the program by directing an execution environment such as a simulator, in-circuit emulator, or target monitor.

The debugger executes your program while you access variables, procedures, source line addresses, and other program entities.

The powerful XRAY Debugger command language allows simple and complex breakpoint setting, single-stepping, and continuous variable monitoring. Input and output can be directed to and from files, buffers, or windows. In addition, a sophisticated macro facility lets you automate tasks and associate complex command sequences with events such as executing a statement or accessing a specific data location. These features allow you to isolate errors and patch your source code.

Data Flow

Figure 1-1 shows the components of the toolkit and the flow of data among them.

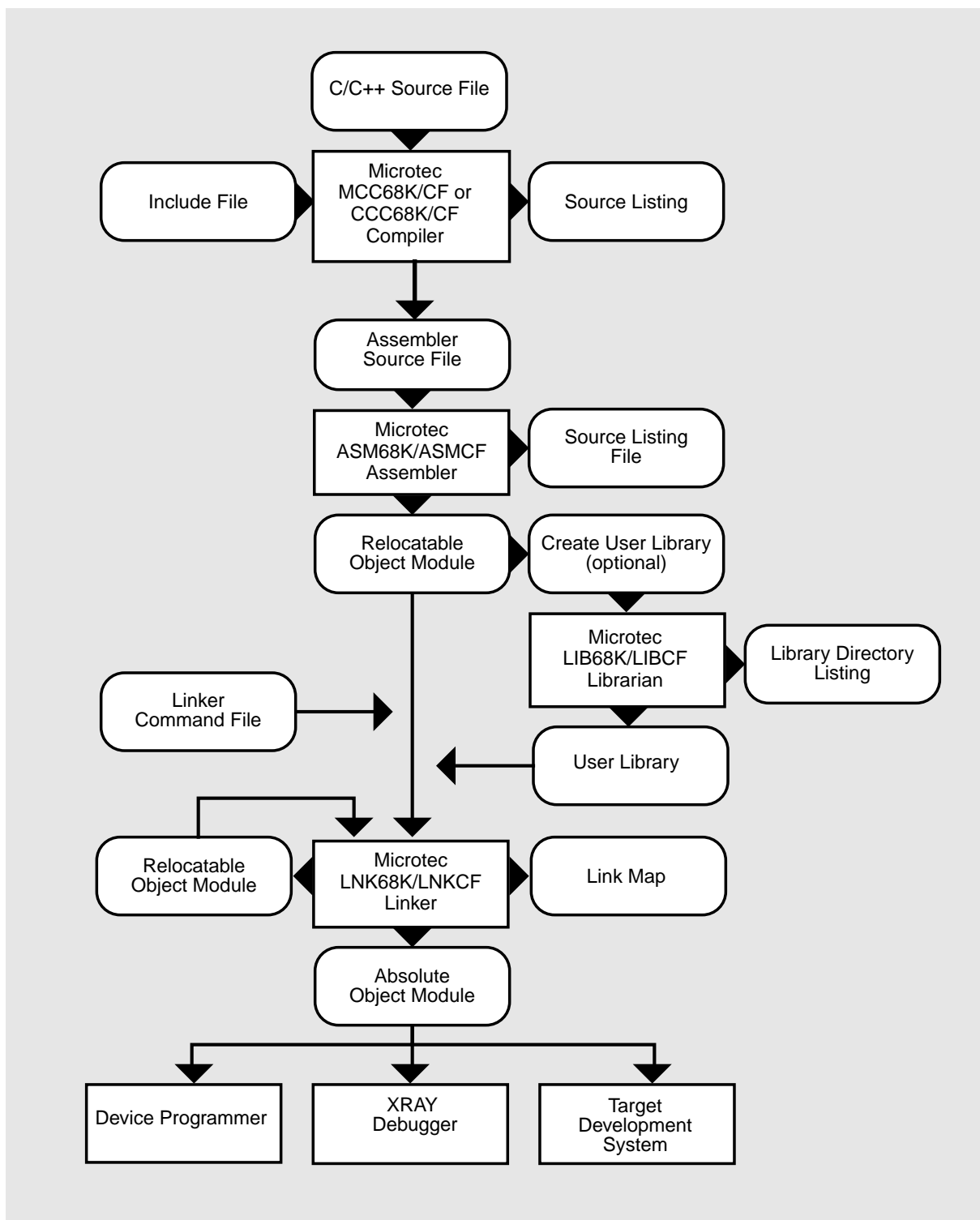


Figure 1-1. Cross Environment Data Flow Through the Microtec Toolkit

Using the Compilers 2

This chapter describes how to invoke the Microtec C and C++ compilers on a host computer running a UNIX operating system or a DOS shell under Windows. The syntax is identical for these systems, except where specifically noted.

Invoking the Compilation Driver

The Microtec MCC/CCC (C/C++) compilers are distributed with a UNIX-style driver program. The driver program accepts multiple input files and automatically invokes the appropriate sequence of compiling, assembling, and linking commands to produce an executable file.

This section describes the compiler's invocation command, filename extensions, and default file locations.

Command Line Syntax

The following commands invoke the compilation driver:

```
xcc68k [-option | source_filename] ...  
xcccf [-option | source_filename] ...
```

Description:

xcc68k or *xcccf* The name of the compilation driver (one of the following: *mcc68k*, *mcccf*, *ccc68k*, or *cccf*).

option Any of the command line options described in the section *UNIX/Windows Command Line Option Summary* in Chapter 3. You may specify *options* and *source_filenames* in any order. A prefix (- or +) must precede each option. If no options are specified, the driver uses the default settings.

source_filename The name of a file containing one of the following:

- C or C++ program source file
- 68000 or ColdFire family assembly language file
- Object file
- Object library

Filename Extensions

The compilers identify input file types and determine how files should be processed based on the files' extensions. Table 2-1 lists the rules that the driver uses to determine how to process input files. If a file has a different extension (or no extension at all) then the driver assumes it is an object file or a library, and it passes it to the linker. The driver displays a warning message if the file does not have a **.o**, **.obj**, or **.lib** extension.

When an output file is created by the compiler, assembler, or linker, it is given a default extension based on its type.

The extensions for input and output files are listed in Table 2-1.

Table 2-1. Default Filename Extensions

File	UNIX Extension	Windows Extension	How Input Files are Processed
C preprocessed file	.i	.i	Compiled with C compiler
C++ preprocessed file	.ixx	.ixx	Compiled with C++ compiler
C source file (from user)	.c	.c	Compiled with C compiler
C++ source file	.cc, .cxx, .cpp	.cc, .cxx, .cpp, .ixx	Compiled with C++ compiler
Assembly language file	.s	.src or .asm	Assembled with assembler
Relocatable object file	.o	.obj	Passed to linker
Object file library	.lib	.lib	Passed to linker
Executable file	.x	.abs	Executed

Input and Output File Locations

If an input filename does not include a directory specification, the driver looks for the file in the current directory.

If an output filename is not specified or does not include a directory specification, the file is placed in the current directory.

Environment Variables

The driver and compiler have default directories that they use to write temporary files and to find toolkit executables, run-time libraries, and standard include files.

To override these default directories, set the environment variables as shown in Table 2-2. Except for in **COMPILER_HOME**, multiple directories can be specified by separating entries with a colon (:) for UNIX hosts or a semicolon(;) for Windows hosts. This establishes the order in which the directories will be searched.

In Table 2-2, replace **68k** with **cf** and **68K** with **CF** to set environment variables for ColdFire targets (for example, **MRI_68K_BIN** becomes **MRI_CF_BIN**).

For a list of all environment variables that affect the search paths for other Mentor Graphics products, refer to the appropriate *Getting Started* manual.

Table 2-2. Default Environment Variables and Defaults for 68000 Processors

Environment Variable	Description	UNIX Default	Windows Default
COMPILER_HOME	Directory containing executables, include files, and library directories	/opt/MGC/embedded	\MGC\embedded
MRI_68K_BIN	Directories to search for executable files	\$COMPILER_HOME/bin or /opt/MGC/embedded/bin	%COMPILER_HOME%\bin or \MGC\embedded\bin
MRI_68K_INC	Directories to search for standard include files	\$COMPILER_HOME/ include/mcc68k or /opt/MGC/embedded/include/ mcc68k	%COMPILER_HOME%\ include\mcc68k or \MGC\embedded\include\ mcc68k
MRI_68K_LIB	Directories to search for libraries and the default linker command file	\$COMPILER_HOME/lib or /opt/MGC/embedded/lib	%COMPILER_HOME%\lib or \MGC\embedded\lib
MRI_68K_TMP	Directories to use for temporary files	\$TMP or /tmp	Value of TMP environment variable or current directory

Note

If you invoke the compiler using an absolute pathname (such as **/opt/MGC/embedded/bin/mcc68k**), then the driver will ignore the **\$COMPILER_HOME** environment variable when looking for other executables, such as the linker and assembler. It will use the path specified when invoking the compiler (such as **/opt/MGC/embedded/bin**) instead. However, if the **\$MRI_68K_BIN** environment variable is set, that value will override this behavior.

Setting Environment Variables for UNIX

The following examples show how to set environment variables for a UNIX host.

Example:

If the distribution files are installed under **/usr2/mri**, follow these steps:

1. Include **/usr2/mri/bin** in your current search path.
2. Set the **COMPILER_HOME** environment variable.

If you are using the C shell (**cs**h), enter the command:

```
setenv COMPILER_HOME /usr2/mri
```

If you are using the Bourne (**sh**) or Korn shell (**ksh**), enter these commands:

```
COMPILER_HOME=/usr2/mri
export COMPILER_HOME
```

The above commands set the search path for executable files to **/usr2/mri/bin**, for include files to **/usr2/mri/include/mcc68k**, and for library files to **/usr2/mri/lib**.

Example:

If the executable files are in a directory other than **/opt/MGC/embedded/bin** or **\$COMPILER_HOME/bin** (for example, **/usr3/mri/bin**), follow these steps:

1. Include **/usr3/mri/bin** in your current search path.

2. Set the **MRI_68K_BIN** environment variable.

If you are using the C shell (**csh**), enter the command:

```
setenv MRI_68K_BIN /usr3/mri/bin
```

If you are using the Bourne (**sh**) or Korn shell (**ksh**), enter these commands:

```
MRI_68K_BIN=/usr3/mri/bin
export MRI_68K_BIN
```

Set **MRI_68K_INC**, **MRI_68K_LIB**, and **MRI_68K_TMP** in a similar manner.

Setting Environment Variables for DOS Shell under Windows

The following examples show how to set environment variables for a Windows host.

Example:

If you want to install the distribution files in **C:\MCC68K**, but you want to invoke the compiler while your default drive is **D:**, follow these steps:

1. Include **C:\MCC68K** in your current search path.
2. Enter these commands from the terminal:

```
set COMPILER_HOME=C:\MCC68K
or:
set MRI_68K_BIN=C:\MCC68K\BIN
set MRI_68K_INCLUDE=C:\MCC68K\INCLUDE\MCC68K
set MRI_68K_LIB=C:\MCC68K\LIB
```

3. Edit the **AUTOEXEC.BAT** file to include the commands listed in the previous step.

Special Considerations for Windows

If **MRI_68K_BIN** is not defined, then the following directories are searched for executables in this order:

1. The drive and directory containing **mcc68k.exe**.
2. The directory **%COMPILER_HOME%\bin** if the **COMPILER_HOME** environment variable is set.
3. The directory **boot_drive:\MGC\embedded\bin**.

If **MRI_68K_LIB** is not defined, then the following directories are searched for libraries and the default linker command file in this order:

1. The directory **%COMPILER_HOME%\lib**.
2. The directory *boot_drive*:\MGC\embedded\lib.

If **MRI_68K_INC** is not defined, then the following directories are searched for standard include files in this order:

1. The directory **%COMPILER_HOME%\include\mcc68k**.
2. The directory *boot_drive*:\MGC\embedded\include\mcc68k.

If **MRI_68K_TMP** is not defined, then the following directories are used as temporary files in this order:

1. The drive and directory set by the environment variable **TMP**.
2. The current directory.

Using Command Line Options 3

This chapter describes the syntax of Microtec C and C++ compiler options, which are supported on host computers running a UNIX or Windows operating system. The syntax is identical for these systems, unless specifically noted.

The chapter includes:

- Quick-reference tables summarizing the command line options
- Expanded descriptions of command line options and defaults
- Tips on how to use options wisely
- Sample command line options

About Command Line Options

Command line options specify the names of output files and turn features on and off.

Each option begins with a prefix, either a minus sign (–) or a plus sign (+). These prefixes distinguish options from filenames. Options preceded by a plus sign (+) are used specifically to control the C++ compiler.

A single-letter option without arguments can be followed immediately by another option letter. For example, the combination of options **-c** and **-g** is **-cg**.

Some options have a negative form that disables or turns off the option. Specify the negative form by entering **n** before the name of the option.

If conflicting options are specified in the same command line, the last option on the line overrides the earlier options. If a command line specifies two filenames for the same output file, the second name is used.

All of the following command line options can be given to the driver program. Some of the options control the behavior of the driver itself, and the others are passed on to the compiler.

Command Line Options — Summary

Table 3-1 summarizes, in alphabetical order, the C/C++ compiler command line options. The (C only) and (C++ only) parenthetical statements next to some of the

option names indicate that the option is only available in either the C or the C++ compiler.

Table 3-1. UNIX/Windows Command Line Option Summary

Option	Meaning
-A (C only)	Sets ANSI-compliant mode. (default: -A)
-acc	References items in the const section according to the -Mc option. (default: -acc)
-acd	References items in the const section according to the -Md option. (default: -acc)
-aic	References items in the vars section according to the -Mc option. (default: -aid)
-aid	References items in the vars section according to the -Md option. (default: -aid)
-alc	References items in the compiler-generated literals section according to the -Mc option. (default: -alc)
-ald	References items in the compiler-generated literals section according to the -Md option. (default: -alc)
+array_new_and_delete (C++ only)	Enables the array new and array delete operators. The +no_array_new_and_delete option disables these operators. (default: +array_new_and_delete)
-asc	References items in the strings section according to the -Mc option. (default: -asc)
-asd	References items in the strings section according to the -Md option. (default: -asc)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-azc (C++ only)	References items in the compiler-generated zerovars section according to the -Mc option. (default: -azd)
-azd (C++ only)	References items in the compiler-generated zerovars section according to the -Md option. (default: -azd)
+bool (C++ only)	Enables the bool keyword. The +no_bool option disables the bool keyword. (default: +bool)
-C	Saves comments in preprocessor output (see -E , -P). (default: -nC)
-c	Produces an object file but not an executable file. (default: -nc)
-Dname[=value]	Defines the value of a preprocessor macro. (default: No preprocessor macro values supplied.)
-doption_file	Reads options from the specified file. (default: Options supplied by command line.)
+delete_std (C++ only)	Cleanup action is not generated for file scope and function scope static variables. (default: Cleanup action is generated for these variables.)
-E[s]	Invokes only the preprocessor without removing #line or #pragma directives. Sends output to standard output (see -C , -P). -Es disables echoing of newline and backslash. (default: -nE)
-ecommand_file	Passes the command file to the linker. (default: -ecc68k.cmd)
+Efilename (C++ only)	Redirects diagnostic output to file (Windows only). (default: Diagnostic messages sent to standard error device.)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
+e (C++ only)	Generates references to externally declared virtual function tables. (default: The compiler generates a virtual function table in the compilation unit that defines a non-in-lined and non-inherited virtual function.)
+ne (C++ only)	Generates uninitialized virtual function tables that are externally declared. (default: The compiler generates a virtual function table in the compilation unit that defines a non-in-lined and non-inherited virtual function.)
-Fee	Writes diagnostic messages to standard error. (default: -Fee for Windows systems; -Feo for UNIX systems.)
-Feo	Writes diagnostic messages to standard output. (default: -Fee for Windows systems; -Feo for UNIX systems.)
-Fli	Shows the contents of #include files in the listing file. (default: -nFli)
-Flp <i>number</i>	Sets the page length of the listing file. (default: -Flp55)
-Flt <i>string</i>	Specifies the title for the listing file. (default: No title string appears on the listing file.)
-Fsi	Expands the contents of #include files in assembly output file. (default: -nFsi)
-Fsm	Includes high-level source code as comments in the assembly output file. (default: -nFsm)
-Fsr <i>number</i>	Specifies the radix for printing constants in the assembly output file. (default: -Fsr10)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-f	Generates code that uses instructions provided by the floating-point coprocessor. (default: -nf)
-Gd	Generates debugging information for preprocessor macros. (default: -Gd)
-Gf	Generates fully qualified path names for input files. (default: -nGf)
-Gl	Generates line number labels. (default: -nGl)
-Gm	Generates debugging information for multistatement lines. (default: -Gm)
-Gr	Generates restricted debugging information. (default: -nGr)
-GS (C only)	Enables the data browsing facility of the XRAY Source Explorer. (default: -nGS)
-Gs	Generates debugging information for use by XRAY Source Explorer. (default: -nGs)
-g	Generates debugging information. (default: -ng)
-H	Saves assembly file. (default: -nH)
-h	Generates code for HP 64000 Series Development Systems. (default: -nh)
-i (C++ only)	Enables informational reports on compiler optimizations. Equivalent to -ic -id -it . (default: -ni)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-ic (C++ only)	Enables reports on constant propagation. (default: -nic)
-id (C++ only)	Enables reports of dead assignment elimination. (default: -nid)
-it (C++ only)	Enables reports of tail recursion optimization. (default: -nit)
-I <i>dir</i>	Specifies the search path for nonstandard #include files. (default: See the full description of this option later in this chapter for a list of the directories searched.)
-I@	Changes the location of the source file's directory on the include search path.
-J <i>dir</i>	Specifies the search path for standard #include files. (default: See the full description of this option later in this chapter for a list of the directories searched.)
-jH (C++ only)	Directs the compiler to automatically use and create (if necessary) a precompiled header file. (default: -njH)
-KAx	Alignment of data object. (default: -KAH)
-Kc	Directs the compiler to use the CLR instruction. (default: -Kc)
-KFi	Directs the compiler to store state of FPU on interrupt. (default: -nKFi)
-Kf	Directs the compiler to generate a stack frame for all functions. (default: -nKf)
-Khreg[,reg] . . .	Reserves one or more registers. (default: No registers reserved.)
-KI	Initializes local variables to zero. (default: -nKI)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-KJ	Generates switch statement jump tables in a DATA section. (default: -nKJ)
-Km	Generates messenger symbols for use by other toolkit components. (default: -Km)
-KP	Specifies structs , unions , and enums to be packed by default. (default: -nKP)
-Kq	Makes double variables or constants 32 bits in size. (default: -nKq)
-Kr	Uses the RTS instruction to return from an interrupt function. (default: -nKr)
-Ksnumber	Generates stack overflow check code. (default: -Ks-1)
-Kt	Generates code to tag a function's exit and entry points. (default: -nKt)
-Ku	Treats plain char variables as unsigned . (default: -nKu)
-Kv	Treats bit fields as unsigned . (default: -nKv)
+lac++ (C++ only)	Specifies the C++ language as described in <i>The Annotated C++ Reference Manual</i> . (default: +lac++)
+lc++ (C++ only)	Specifies C++ with anachronisms. (default: +lac++)
+lf2 (C++ only)	Specifies (strict) Cfront 2.1. (default: +lac++)
+lf2o (C++ only)	Specifies Cfront 2.1 with anachronisms. (default: +lac++)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
+lf3 (C++ only)	Specifies Cfront 3.0. (default: +lac++)
+lf3o (C++ only)	Specifies Cfront 3.0 with anachronisms. (default: +lac++)
-l[filename]	Generates a source listing with diagnostics. (default: -nl)
-mac	Directs the compiler to take advantage of the ColdFire MAC unit. (default: -nmac)
-mdiv	Directs the compiler to generate hardware divide instructions in the presence of the ColdFire DIV unit. (default: -nmdiv)
-Mca	Directs the compiler to use absolute addressing for all code references. (default: -Mca)
-Mcp	Directs the compiler to use PC-relative addressing for all code references. (default: -Mca)
-Mdnumber	Directs the compiler to use register-relative addressing for all static data references. (default: -Mda)
-Mda	Directs the compiler to use absolute addressing for all data references. (default: -Mda)
-Mdp	Directs the compiler to use PC-relative addressing for all static data references. (default: -Mda)
-Ml	Directs the compiler to use 32-bit displacements for register <i>An</i> -relative or PC-relative addressing. (default: 16-bit displacements)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-Mlc	Directs the compiler to use 32-bit displacement for An - or PC-relative addressing for code references only. (default: 16-bit displacement)
-Mld	Directs the compiler to use 32-bit displacement for An - or PC-relative addressing for static data references only. (default: 16-bit displacement)
-Mls	Directs the compiler to use 32-bit displacement in switch statement jump tables. (default: 16-bit displacement)
-NCname	Sets the constant variables section name. (default: -NCconst)
-NIname	Sets the initialized data section name. (default: -NIvars)
-NJname	Sets the static constructors/destructors call site section name. (default: -NJinitfini)
-NLname	Sets the compiler-generated literals section name. (default: -NLliterals)
-NMname	Sets the module name. (default: Module name is source filename stripped of any suffix or prefix.)
-NSname	Sets the string section name. (default: -NSstrings)
-NTname	Sets the code section name. (default: -NTcode)
-NZname	Sets the uninitialized static data section name. (default: -NZzerovars)
-noption	Negates the specified option(s).
-O	Enables a standard set of optimizations. Equivalent to -Ocgllr . (default: -nO)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-Ob	Performs optimizations that assume that global memory objects are not changed through aliases. (default: -nOb)
-Oc	Does not pop the stack after each function call. Causes a direct branch to be generated in place of a function call when the last statement of a function is a function call and the function does not otherwise require a stack frame. (default: -Oc)
-Oe	Generates only one exit (return) from a function. (default: -nOe)
-Og	Enables global-flow optimizations. (default: -nOg)
-Oi	In-lines function calls within a module. (default: -nOi)
-Oj	Enables run-time library function in-lining. (default: -Oj)
-OL	Enables all loop options (-OL...). (default: -nOL)
-OL <i>number</i>	Enables forward code motions with loops. (default: -OLf0)
-OLhc	Enables loop constant hoisting. (default: -nOLhc)
-OLhi	Enables loop invariant hoisting. (default: -nOLhi)
-OLiv	Enables elimination of induction variables within loops. (default: -nOLiv)
-OLr	Enables replacement of same family loop expressions. (default: -nOLr)
-OL <i>snumber</i>	Enables strength reduction within loops. (default: -nOLs0)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-OLt	Enables conversion of loop entry testing. Use only in conjunction with -Ot . (default: -nOLt)
-OLul <i>number</i>	Enables loop unrolling. (default: -OLul0)
-OI	Enables local optimizations. (default: -OI)
-OR	Allocates heavily used variables to registers. (default: -OR)
-Or	Enables instruction scheduling optimization. (default: -nOr)
-Os	Optimizes in favor of code size rather than execution time. (default: -nOs)
-Ot	Optimizes in favor of execution time rather than code size. (default: -nOt)
-OT	Dictates the maximum number of temps that can be created during in-lining. (default: -OT2000)
-Ou	Enables the removal of unreferenced functions during the link phase. (default: -Ou0)
-OV	Treats all static data as volatile. (default: -nOV)
-OX <i>number</i>	Enables copy propagation optimization. (default: -OXc0)
-OXcs <i>number</i>	Enables common sub-expression elimination optimization. (default: -OXcs0)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
<i>-OXi</i> <i>number</i>	Enables function in-lining optimization. (default: -OXii0 ; if the -Oi option is enabled, -OXii2 (C language) or -OXii1 (C++ language))
<i>-OXk</i> <i>number</i>	Enables constant propagation optimization. (default: -OXk0)
<i>-OXrep</i> <i>number</i>	Enables aggressive case and constant propagation behavior. (default: -OXrep0)
-OXrf	Enables factorization optimizations. (default: -nOXrf)
-OXsr	Enables scratch register optimization. (default: -nOXsr)
-OXt	Enables tail recursion optimization. (default: -nOXt)
-Oz38	Prevents conversion of JSRs to BSRs with the -Mca option. (default: -nOz38)
-o <i>filename</i>	Names the output file. (default: See full entry for more information.)
-P [<i>s</i>]	Executes the preprocessor only, sending output to either the .i (in C) or .ixx (in C++) file (see -C , -E). (default: -nP)
-pprocessor	Produces code for a specified processor. (default: -p68000 for 68K; -p5200 for CF)
-Q <i>number</i>	Specifies the maximum number of error messages before quitting. (default: -Q20)
-QA (C++ only)	Suppresses all messages. (default: No messages suppressed.)
-Qe	Suppresses error, warning, and informational messages. (default: No messages suppressed.)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-Qfn (C++ only)	Suppresses display of the message number when the diagnostic is displayed. (default: Message number displayed.)
-Qfs (C++ only)	Suppresses display of the source line when the diagnostic is displayed. (default: Source line number displayed.)
-Qi	Suppresses informational messages. (default: -Qi)
-Qme' <i>msgid</i> [, <i>msgid2</i>] . . . '	Makes diagnostic IDs error messages. (C++ only)
-Qmi' <i>msgid</i> [, <i>msgid2</i>] . . . '	Makes diagnostic IDs informational messages. (C++ only)
-Qms' <i>msgid</i> [, <i>msgid2</i>] . . . '	Suppresses issuance of the specified diagnostics. (C++ only)
-Qmw' <i>msgid</i> [, <i>msgid2</i>] . . . '	Makes diagnostic IDs warnings. (C++ only)
-Qo	Suppresses listing of options currently active. (default: -nQo)
-Qs	Suppresses summary message. (default: No messages suppressed.)
-Qw	Suppresses warning and informational messages. (default: No messages suppressed.)
-q	Instructs the driver to pass the default library to the linker in addition to the linker command file. (default: -nq)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
+rtti (C++ only)	Enables the typeid and dynamic_cast keywords and other RTTI-related features. The +no_rtti option disables these keywords. (default: +no_rtti)
-S	Generates code for the assembler. (default: No assembly language file saved.)
+tl (C++ only)	Instantiates all used templates as local. (default: +tu)
+tm (C++ only)	Does not instantiate templates, unless manually created with a pragma. (default: +tu)
+tu (C++ only)	Instantiates all used templates as external, unless declared as static. (default: +tu)
-Uname	Undefines a preprocessor macro. (default: No preprocessor macro is undefined.)
-ui[char]	Changes the insert character for asm pseudofunction calls to <i>char</i> . (default: Back quote (`) character used.)
-upd	Prepends a dot to symbol names. (default: -upu)
-upu	Prepends an underscore to symbol names. (default: -upu)
-us	Suppresses the name modifier. (default: -upu)
+use_ODR (C++ only)	Uses ANSI C++ One Definition Rule. (default: Does not use this rule.)
-utnumber	Truncates the length of program identifiers to <i>number</i> . (default: -ut0)
-Vb	Displays the banner before compiling. (default: Banner not displayed.)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-Vd	Displays commands for invoking components. Commands are not executed. (default: Commands executed but not displayed.)
-Vi	Displays commands for invoking components. Commands are executed. (default: Commands executed but not displayed.)
-Vt	Displays a time stamp for the various stages of the compilation (UNIX only). (default: Time stamps not displayed.)
-Vw	Displays the banner and exits. (default: Compilation occurs without displaying banner.)
-v	Detects and issues warnings for features in the source files that can cause problems. (default: -nv)
-Wa,'option1 [,option2] . . .'	Passes options to the assembler. (default: No options passed to assembler.)
-Wl,'option1 [,option2] . . .'	Passes options to the linker. (default: No options passed to linker.)
+wchar_t (C++ only)	Enables the wchar_t keyword. The +no_wchar_t option disables the wchar_t keyword. (default: +wchar_t)
-X0	Allocates space for global variables that have not been explicitly initialized and initializes them to zero. (default: -Xc)
-Xc	Treats uninitialized global variables as “weak externals.” (default: -Xc)
-Xp	Allocates space for global variables that have not been explicitly initialized. (default: -Xc)

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-x	Enables Microtec Compiler extensions to C and C++. (default: -x)
-y	Checks source syntax but does not compile. (default: -ny)
-Za2	Uses even alignment for structure members larger than char . (default: -Za2)
-Za4	Uses quad alignment for structure members larger than short . (default: -Za2)
-ZBA (C++ only)	Generates object code compatible with CCC68K 1.X with the -A option.
-ZBnA (C++ only)	Generates object code compatible with CCC68K 1.X with the -nA option.
-Ze	Passes parameters of size char on even stack addresses. (default: -nZe)
-Zle (C++ only)	Makes struct layout compatible with Release 4.5 and earlier versions of MCC68K. (default: -Zle)
-Zmnumber	Forces packed structure sizes to be a multiple of <i>number</i> . (default: -Zm1)
-Znnumber	Forces unpacked struct sizeof to be a multiple of <i>number</i> .
-Zo	Passes parameters of size char on odd stack addresses. (default: -nZo)
-Zp2	Passes parameters of size short or smaller as short . (default: -nZp2)
-Zp4	Passes parameters smaller than int as int . (default: -nZp4)
-zc	Allows // comments in C code.

(cont.)

Table 3-1. UNIX/Windows Command Line Option Summary (cont.)

Option	Meaning
-ze	Enables C++ exception handling. (default: -nze)
+z <i>suffix</i> (C++ only)	Compiles files with <i>suffix</i> extension as C++ files. (default: Only files with standard C++ extension are compiled as C++ files.)

Command Line Options — Extended Descriptions

The following pages describe in greater detail the command line options, listed according to function, allowed on the UNIX and Windows operating systems for the C/C++ compilers and their drivers. Please refer to Table 3-2 for the order of the options.

Table 3-2. Command Line Option Function Categories

Section Headers	Options
Enabling Microtec Compiler Extensions	-x
Setting Compatibilities	-A, -ZBA
Controlling Include File Search Paths	-I, -jH, -J
Controlling Listing Files	-Fl, -Fs, -l
Defining Preprocessor Names	-D, -U
Controlling the Preprocessor and Its Output	-C, -E, -P, -c, -d, -e, -H, -o, -p, -q, -S, -y, -V, -v, -W, -Za
Modifying Naming Conventions	-u, +use_ODR
Generating Floating-Point Processor Instructions	-f
Supporting ColdFire Multiple and Accumulator Unit	-mac, -mdiv
Producing Debug Information	-G, -g, -h
Controlling Diagnostic Messages	+E, -Fe, -Q
Handling Uninitialized Global Data	-X
Optimizing Code	-i, -O
Producing Minor Code Generation Variations	-K, -Z
Generating Position-Independent Code and Data	-M, -a, -N
Enabling C++ Extensions	+array_new_and_delete, +bool, +delete_std, +e, +l, +t, -z, +rtti, +wchar_t, +z

Enabling Microtec Compiler Extensions

-x (default)

-nx

The **-x** option enables the following Microtec Compiler extensions to the C and C++ languages:

- The predefined preprocessor symbol **`_MRI_EXTENSIONS`**
- Acceptance of **`\X`** for hexadecimal sequences
- Additional bit field types **`char`**, **`short`**, and **`long`**
- Assembly in-lining with **`asm`**
- In-lining of selected library functions
- **`interrupt`** procedures (**`interrupt`** keyword)
- **`packed`** structures (**`packed`** and **`unpacked`** keywords) and **`enums`**
- Preprocessor directives **`#warning`** and **`#inform`**

The following library functions are expanded in-line rather than called:

strcpy	memcpy	strlen
--------	--------	--------

When 68881 code is generated (**-f**), the following function calls are replaced by in-line 68881 code:

acos	cosh	sin
asin	exp	sinh
atan	fabs	sqrt
atanh	log	tan
cos	log10	tanh

The **-nx** option disables the Microtec Compiler extensions.

Setting Compatibilities

ANSI Compatibility

-A (default)

-nA

C only.

The **-A** option enables additional features provided by ANSI C. The `__STDC__` preprocessor symbol is predefined.

The **-nA** option disables ANSI features. With the **-nA** option, the compiler will accept pre-ANSI programs. (For example, if **-nA** is used, **volatile** will not be recognized as a keyword.)

The **-nA** option does not guarantee compatibility with earlier versions of Microtec compilers or with other pre-ANSI compilers.

Object Code Compatibility

-ZBA

-ZBnA(default)

C++ only.

The **-ZBA** option instructs the compiler to generate object code compatible with code produced by CCC68K 1.2 with the **-A** option.

The **-ZBnA** option instructs the compiler to generate object code compatible with code produced by CCC68K 1.2 with the **-nA** option.

Controlling Include File Search Paths

Add Search Path for Nonstandard Include Files

-Idir

The **-I** option specifies a search path to be scanned to locate user-supplied **#include** files. If an include filename in your source file is enclosed in double quotes (" "), the compiler searches for the include file in the following directories:

1. The directory containing the source file that has the **#include** directive.
2. The directory containing the top-level source file.

3. The directories specified by the **-I** option.

The compiler searches multiple directories if you precede each directory name with **-I**. For example, if you enter **-Idir1 -Idir2 -Idir3**, the compiler searches for **#include** files in the directory containing the source file, the *dir1* directory, the *dir2* directory, and the *dir3* directory, in that order.

4. The directories specified with the **-J** option.
5. The directories specified by the environment variable **MRI_68K_INC**.
6. The directory **\$COMPILER_HOME/include/mcc68k** on UNIX or the directory **%COMPILER_HOME%\include\mcc68k** on Windows, if **MRI_68K_INC** is not defined.
7. The standard include file directory if neither **MRI_68K_INC** nor **COMPILER_HOME** is defined.
8. The directory from which the compiler was invoked.

See the **-J** option for information on include filenames enclosed in angle brackets (< >), and see the **-I@** option for information on altering the ordering of the search for the include files.

Use Precompiled Headers

-jH
-njH (default)

The **-jH** option tells the compiler to automatically use and create a precompiled header file (if necessary). Subsequent occurrences of the **-jHc** and **-jHu** options override this option.

-jHcfile_name

The **-jHcfile_name** option tells the compiler to create a precompiled header file with the specified name. Subsequent occurrences of the **-jH** option override this option.

-jHddir (default: .)

The **-jHddir** option specifies the directory in which to search for and create precompiled header files.

-jHufile_name

The **-jHufile_name** option tells the compiler to use the specified precompiled header file as part of the current

compilation. Subsequent occurrences of the **-jH** option overrides this option.

For more information about using precompiled header files, refer to Chapter 9, *Precompiled Header Files*.

Change Search Path Order for Include Files

-I@

-nI@ (default)

The **-I@** option indicates that the **#include** files should be searched for first in the directory containing the current source file and then in the directory containing the top-level source file. Although these directories are placed implicitly at the beginning of the search path, this option enables the user to move them from the beginning of the search path to any other position.

The command line:

```
ccc68k -Ia -I@ -Ib t.cc
```

causes the directory search sequence to be directory *a*, the current directory, and then directory *b*.

Add Search Path for Standard Include Files

-Jdir

The **-J** option specifies a search path to be scanned to locate standard **#include** files. When an include filename in your source file is enclosed in angle brackets (< >), the compiler searches for the file in the following directories:

1. The directories specified by the **-J** option.

The compiler searches multiple directories if you precede each directory name with **-J**. For example, if you enter **-Jdir1 -Jdir2 -Jdir3**, the compiler searches for standard **#include** files in the *dir1* directory, the *dir2* directory, and the *dir3* directory.

2. The directories specified by the environment variable **MRI_68K_INC**.
3. **\$COMPILER_HOME/include/mcc68K** on UNIX or the directory **%COMPILER_HOME%\include\mcc68k** on Windows, if **MRI_68K_INC** is not defined.

4. The standard include file directory if neither **MRI_68K_INC** nor **COMPILER_HOME** is defined.

See the **-I** option for information on include filenames enclosed in quotes ("*name*").

Controlling Listing Files

Specify Format of Listing Files

-Fli	
-nFli (default)	The -Fli option puts the contents of each #include file in the listing file. You must specify the -l option to generate a listing file.
-Flp <i>number</i>	The -Flp option sets the number of lines per page in the listing file. The default page length is 55 lines. A specification of less than 10 defaults to 55 lines. You must specify the -l option to generate a listing file.
	Specify -Flp0 to omit page breaks and the page header from the listing file.
-Flt <i>string</i>	The -Flt option specifies a title string that appears at the top of each page of the listing file. You must specify the -l option to generate a listing file. Enclose <i>string</i> in double quotes if it contains any spaces or punctuation.
-Fsi	
-nFsi (default)	The -Fsi option expands the contents of each #include file in the assembler source file.
-Fsm	
-nFsm (default)	The -Fsm option includes high-level source code as comments in the assembler source file. You must specify the -S or -H option to generate an assembly file.
-Fsr <i>number</i>	The -Fsr option specifies the radix of constants printed in the assembly output file. <i>number</i> can be 2, 8, 10, or 16. The default is 10.

Generate Listing File

`-l[filename]`

The **-l** option writes a listing file containing high-level source code and any diagnostic messages to the standard output device or the specified file.

By default, no listing file is generated.

If the **-l** option is specified with **-P** or **-E**, the **-l** option is ignored.

Defining Preprocessor Names

`-Dname[=value]`

The **-D** option defines the value of a preprocessor macro. If you do not specify a string, the preprocessor macro has the value 1 (equivalent to putting **#define name 1** at the top of the source file).

You can define multiple names by preceding each name with **-D**. For example:

```
-Dabc -DABC
```

is equivalent to using the preprocessor directives **#define abc 1** and **#define ABC 1**.

`-Uname`

The **-U** option undefines the defined preprocessor macro specified by *name* (this is equivalent to putting **#undef name** at the beginning of the source file). For example:

```
-UNAME1 -Uname2 -UNAME3
```

undefines the preprocessor macros NAME1, name2, and NAME3.

Note

The **-U** (undefine macro) option is processed before the **-D** (define macro) option, regardless of the order in which they appear on the command line.

Controlling the Preprocessor and Its Output

-C
-nC (default) The **-C** option keeps high-level source comments in the preprocessor output. (See also the **-E** and **-P** options.)

The **-nC** option excludes comments from the preprocessor output.

-E[s]
-nE (default) The **-E** option tells the driver to execute the preprocessor only. Preprocessor output is sent to standard output. The **#line** and **#pragma** directives are not removed. The **#include** directives are converted to appropriate **#line** directives so that the original structure of the source files can be determined. (See also the **-C** and **-P** options.)

If **-Es** is specified, all the **-E** actions are performed, but the echoing of the backslash and newline is disabled.

This option conflicts with **-c**, **-l**, **-P**, and **-S**.

Driver Options

-P[s]
-nP (default) The **-P** driver option tells the driver to execute the preprocessor only. Preprocessor output is sent to a file having the same name as the source file but with either the suffix **.i** (C) or **.ixx** (C++). This option is useful for debugging complicated macro definitions and deeply nested conditional directives. (See also the **-C** and **-E** options.)

This option conflicts with **-c**, **-E**, and **-l**.

If **-Ps** is specified, all the **-P** actions are performed, but the echoing of the backslash and newline are disabled.

With the **-nP** option, compilation continues after preprocessing; the preprocessed source file is not saved.

Produce Object File Only

-c
-nc (default) The **-c** option produces only an object file having either a **.o** extension for UNIX or **.obj** for Windows. The option tells the driver to produce an object file but not to call the linker

to produce an executable file having either a **.x** extension for UNIX or **.abs** for Windows.

The **-nc** option tells the driver to produce an object file and to call the linker to produce an executable file.

Read Options From File (not a driver option)

-doption_file

The **-d** option directs the driver to read command line options from the specified file.

Pass Command File to Linker

-ecommand_file

The **-e** option directs the driver to pass the specified command file to the linker but prevents the driver from passing the default library and default linker command file as well.

Save Assembly File

-H
-nH (default)

The **-H** option instructs the compiler not to remove the generated assembly file. The **-nH** option removes the generated assembly file.

The name of the assembly file is based on the source filename followed by a **.s** (UNIX) or **.src** (Windows) extension.

The **-ofilename** option does not affect the name of the assembly file.

Name the Output File

-ofilename

The **-o** option names the output file with the specified name instead of its default name. The output file can be one of these:

Preprocessed file	If used with the -P option
Assembly source file	If used with the -S option
Object file	If used with the -c option
Executable file	Assumed by default

By default, the output filename is the source filename stripped of any suffix or prefix.

Produce Code for Specified Processor (not a driver option)

-pprocessor

The **-pprocessor** option produces code for the specified processor.

For 68K, the possible values for *processor* are **68000** (default), **68008**, **68010**, **68020**, **68030**, **68040**, **68060**, **68302**, **68330**, **68331**, **68332**, **68333**, **68340**, **68349**, **68360**, **68EC000**, **68EC020**, **68EC030**, **68EC040**, **68EC060**, **68HC000**, **68HC001**, **CPU32**, and **CPU32P**.

For ColdFire, the possible values for *processor* are **5102**, **5200** (default), **5202**, **5203**, **5204**, **5206**, **5307**, and **5400**.

A predefined preprocessor symbol is defined for each *processor* value.

The **-pprocessor** option affects the alignment of global data, local data, and structures. Modules that share structures must be compiled for the same processor.

Some *processor* values actually use the instruction set of a similar processor. The resulting code does not contain any instructions unique to the *processor* specified or instructions not supported by the similar processor. Table 3-3 shows *processor* values, the default alignment, the processor instruction set used, and the library that should be used with the specified *processor*.

Table 3-3. Processor Identification

Processor Value	Default Alignment	Instruction Set Used	Library To Be Used
68000	2	68000	68000
68008	2	68000	68000
68010	2	68010	68000
68020	4	68020	68020
68030	4	68030	68020
68040	4	68040	68040
68060	4	68060	68040

(cont.)

Table 3-3. Processor Identification (cont.)

Processor Value	Default Alignment	Instruction Set Used	Library To Be Used
68302	2	68000	68000
68330	2	CPU32	CPU32
68331	2	CPU32	CPU32
68332	2	CPU32	CPU32
68333	2	CPU32	CPU32
68340	2	CPU32	CPU32
68349	4	CPU32+	68020
68360	4	CPU32+	68020
68EC000	2	68000	68000
68EC020	4	68020	68020
68EC030	4	68EC030	68020
68EC040	4	68EC040	68020
68EC060	4	68EC060	68020
68HC000	2	68000	68000
68HC001	2	68000	68000
CPU32	2	CPU32	CPU32
CPU32P	4	CPU32+	68020
5102	4	68EC040	5102
5200/02/03/ 04/06	4	5200	5200
5307	4	5307	5200

The default alignment also determines whether the **-Za2** (default alignment 2) or **-Za4** (default alignment 4) option is in effect (see the *Enabling C++ Extensions* section in this chapter for more information).

With **-p5307**, **-mac** and **-mdiv** are always on.

If you do not use the default linker command file (**-e** option), you must include the appropriate library for your processor in your linker command file.

Note

This guide uses the term “68020-class” to refer to the 68020, 68EC020, 68030, 68EC030, 68040, 68EC040, 68060, 68EC060, 68330, 68331, 68332, 68332, 68333, 68340, 68349, 68360, and CPU32/CPU32+ microprocessors.

A predefined preprocessor symbol is defined for each *processor* value.

Pass Default Library to Linker

-q
-nq (default)

Instructs the driver to pass the default library to the linker in addition to the libraries specified in the **-ecmdfile** option.

If **-nq** is specified, the compiler does not pass the default libraries to the linker when a linker command file is specified with the **-e** option.

Produce Assembler Source File

-S

The **-S** option instructs the driver to produce an assembly language file (**.s** extension for UNIX and **.src** for Windows), which can be passed to the assembler to produce an object file (**.o** extension for UNIX and **.obj** for Windows).

This option conflicts with **-c**, **-E**, **-P**, and **-y**.

Check Program Syntax

-y
-ny (default)

The **-y** option checks the program syntax without generating code.

The **-ny** option checks the syntax and generates code.

Control Verbose Output Information

-Vb	The -Vb option displays the copyright notice and version number and continues compilation.
-Vd	The -Vd option displays the commands that are used to invoke each component of the toolkit without executing the commands. If you redirect your output to a file, you can use the output to create a script for future compilations.
-Vi	The -Vi option displays the commands used to invoke each component of the toolkit as they are executed.
-Vt (UNIX only)	The -Vt option displays a time stamp for the various stages of the compilation.
-Vw	The -Vw option displays the copyright notice and version number and then exits.
-nV (default)	The -nV option disables the display of verbose messages.

Perform Extra Checking (not a driver option)

-v	
-nv (default)	<p>The -v option tells the compiler to issue warnings of potential problems in the source file. For example, the -v option issues warnings for the following:</p> <ul style="list-style-type: none"> • Comparison of an unsigned int with zero or a negative integer • Assignments in contexts where an equality comparison might be intended • Pointer conversions that could cause misaligned accesses

Pass Options to Tools

-Wa,' <i>option1</i> [' <i>option2</i>] . . . '	<p>The -Wa option passes the specified options directly to the assembler. This option applies only if the driver is invoking the assembler; that is, if -E, -P, or -S options are not specified.</p>
--	--

Enclosing quotes are not necessary when a simple single command is passed with the **-Wa** option, as shown in the following example:

```
ccc68k -Wa,-g main.s
```

Complex embedded commands passed by **-Wa** often require both single (') and double (") quotes as shown in the following examples:

```
ccc68k -Vi y.cc -Wa,'-f "intfile, case"'
ccc68k '-Wa,-l>main.l' main.cc
```

In these examples, both single and double quotes were necessary because the **-f** command line option uses the assembler command line flags `intfile` and `case`.

For information about assembler command line options, see the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.

-Wl,'option1[,option2] . . . '

The **-Wl** option passes the specified options directly to the linker. This option applies only if the driver is invoking the linker; that is, if **-c**, **-E**, **-P**, or **-S** options are not specified.

Enclosing quotes are not necessary when a simple single command is passed with the **-Wl** option, as shown in the following example:

```
ccc68k -Wl,-M main.cc
```

For information about linker command line options, see the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.

Modify Alignment (not a driver option)

-Za2 (default)
-Za4

The **-Za2** option specifies that non-**char** structure members are allocated on two-byte addresses. The **-Za4** option specifies that four-byte structure members are allocated on four-byte addresses and two-byte structure members are allocated on two-byte addresses. The alignment options do not affect **packed** structures.

Modifying Naming Conventions

The compiler's default naming convention puts an underscore at the beginning of symbol names. The C and C++ run-time libraries are built using the default naming convention. Therefore, if a module that contains a reference to a library item is not compiled with the default naming convention, that item is unresolved at link time.

<code>-ui[<i>char</i>]</code>	The -ui option changes the default insert character for asm pseudofunction calls from a back quote (`) to <i>char</i> . If <i>char</i> is null, no insert character is defined.
<code>-upd</code>	The -upd option prepends a dot (.) to all global symbol names. By default, the compiler prepends an underscore to all public symbols.
<code>-upu</code> (default)	The -upu option directs the compiler to prepend an underscore (_) to all global symbol names.
<code>-us</code>	The -us option directs the compiler to suppress the name modifier (dot or underscore).

Note

The **-upd**, **-upu**, and **-us** options are mutually exclusive.

<code>+use_ODR</code>	C++ only. Each class that is not local to a function must have an identical definition in all compilation modules. This allows a linker to retain only one instance of each member function and virtual function table for each non-local class. All C++ virtual function tables are merged by class name, unless the class is local to a function, and all C++ member functions are merged by name, unless the function belongs to a class local to a function. Therefore, all non-local classes, including nested classes, must follow the ANSI C++ one definition rule. If a program contains two files that define two classes of the same name but with different definitions, one of the two virtual function tables will be used for both classes. This could
-----------------------	---

cause a run-time error, which is not detected by the compiler or linker.

-ut0 (default)

-ut`number`

The **-ut** option truncates program identifiers to the length of *number*. Language keywords are not truncated.

With the **-ut0** option, program identifiers are not truncated.

Generating Floating-Point Processor Instructions

-f

-nf (default)

The **-f** option generates code that uses instructions provided by the 68881 floating-point coprocessor. The preprocessor symbol **_FPU** is predefined.

The **-f** option generates the messenger symbol **___FPU**, which is resolved in the run-time library. See the section *Messenger Symbols* in Chapter 13, *Embedded Environments*, for further information.

The **-nf** option implements all floating-point operations by making calls to the run-time library.

Supporting ColdFire Multiple and Accumulator Unit

-mac

The **-mac** option allows the ColdFire compiler to generate multiple instructions for constant multiplication in place of add and shifts, and allows the **___MAC** messenger symbol to be generated.

It also causes the **_MAC** assembler switch to be passed in ASMCF, allowing MAC instructions to be processed.

-mdiv

The **-mdiv** option directs the ColdFire compiler to generate hardware divide/remainder instructions, and allows the **___DIV** messenger symbol to be generated.

Producing Debug Information

-Gd

-nGd (default)

The **-Gd** option generates debugging information for preprocessor macros.

Use the **-Gd** option in conjunction with the **-g** option.

-Gf -nGf (default)	<p>The -Gf option generates fully qualified filenames for input files, in addition to line number and symbol information. A fully qualified filename has a full path name, even if the full path was not given on the command line. The fully qualified filename lets the XRAY Debugger find the source file even if the XRAY Debugger is not invoked from the directory in which the source file resides. The source file has a .c, .cc, or .cxx extension.</p> <p>Use the -Gf option in conjunction with the -g option.</p>
-Gl -nGl (default)	<p>The -Gl option generates line number labels with the format LLnn, where <i>nn</i> is the line number. These line number labels are not recognized by the XRAY Debugger but can be used in other debugging environments.</p> <p>The -Gl and the -g option are mutually exclusive. If both -Gm and -Gl are specified, -Gm is ignored.</p>
-Gm (default) -nGm	<p>The -Gm option generates debugging information for “stepping” through individual statements on lines that contain more than one statement.</p> <p>Use the -Gm option in conjunction with the -g option. If both -Gm and -Gl are specified, -Gm is ignored.</p>
-Gr -nGr (default)	<p>The -Gr option generates “restricted” debugging information. Information on frame type, push mask, and start/end addresses are generated for each function. No debugging information is generated for line numbers or symbols.</p> <p>Use this option when compiling modules that have already been debugged. The -Gr option lets the XRAY Debugger economically provide call tracing information for the functions in those modules.</p> <p>The -Gr option and the -g option are mutually exclusive.</p>
-Gs -nGs (default)	<p>The -Gs option generates information for use by XRAY Source Explorer.</p> <p>Use the -Gs option in conjunction with the -g option.</p>

-GS	
-nGS (default)	The -GS option enables the data browsing facility in the XRAY Source Explorer.
	The -GS option implies the -Gs option.
-g	
-ng (default)	The -g option generates line number, variable, and symbol information. The _DEBUG preprocessor symbol is pre-defined.
	Information is in IEEE-695 format, which is readable by the XRAY Debugger. With -g , you must use the Microtec assembler and linker to pass complete debugging information to the XRAY Debugger.

Note

Use the **-h** option in conjunction with the **-g** option. If both **-h** and **-ng** are specified, **-ng** is ignored.

Support HP 64000

-h	
-nh (default)	The -h option generates code that includes two special labels, Rlabel and Elabel , which HP 64000 emulators use to identify each function's exit and end points for debugging and timing purposes. Rlabel represents the function's return point, and Elabel represents the function's end address. The <i>label</i> portion of the symbol name is the name of the function.

Controlling Diagnostic Messages

Redirect Diagnostics to File (Windows only)

+E <i>filename</i>	C++ only.
	The +E<i>filename</i> option redirects diagnostic output to the specified file.

Redirect Diagnostic Messages

-Fee (Windows default)

-nFee (UNIX default)

The **-Fee** option writes diagnostic messages to the standard error device (**stderr**). If **-nFee** is specified, the diagnostic messages are written to the standard output device. Note that **-Fee** and **-Feo** are mutually exclusive.

-Feo (UNIX default)

-nFeo (Windows default)

The **-Feo** option writes diagnostic messages to the standard output device (**stdout**). If **-nFeo** is specified, the diagnostic messages are written to the standard error device. Note that **-Fee** and **-Feo** are mutually exclusive.

Change Diagnostic Message Severity

Diagnostic messages have different severity levels: error, warning, or informational. You can change the severity level of messages that have **-D** appended to their message ID number.

-Q*number*

The **-Q** option stops compilation if more than *number* errors occur.
(default: *number* is 20)

-QA

C++ only.

The **-QA** option suppresses all messages.

-Qe

The **-Qe** option suppresses error, warning, and informational messages.

-Qfn

C++ only.

The **-Qfn** option suppresses the display of the message ID number for each diagnostic.
(default: message numbers are displayed)

-Qfs

C++ only.

The **-Qfs** option suppresses display of the source line for each diagnostic.
(default: source lines are displayed)

-Qi (default)

The **-Qi** option suppresses informational messages.

`-Qme'msgid[,msgid2] . . . '`

C++ only.

The **-Qme** option defines as error messages the diagnostics specified by the comma-separated list of diagnostic IDs.

Enclosing quotes are not necessary when a single *msgid* is passed with the **-Qm** option, as shown in the following example:

```
ccc68k -Qme1234 main.cc
```

`-Qmi'msgid[,msgid2] . . . '`

C++ only.

The **-Qmi** option defines as informational messages the diagnostics specified by the comma-separated list of diagnostic IDs.

`-Qms'msgid[,msgid2] . . . '`

C++ only.

The **-Qms** option suppresses the display of ID numbers for diagnostics specified in the comma-separated list.

`-Qmw'msgid[,msgid2] . . . '`

C++ only.

The **-Qmw** option defines as warnings the diagnostics specified by the comma-separated list of diagnostic IDs.

`-Qo`

`-nQo` (default)

The **-Qo** option suppresses the listing (normally produced with **-l** option) of options that are currently active.

The **-nQo** option lists on the output listing options (produced by the **-l** option) that are active.

`-Qs`

The **-Qs** option suppresses the summary of diagnostic messages.

`-Qw`

The **-Qw** option suppresses warning and informational messages.

`-nQ`

The **-nQ** option does not suppress any messages.

Handling Uninitialized Global Data

-X0	<p>The -X0 option initializes all global variables to zero that were not explicitly initialized when allocated.</p> <p>With the -X0 option, weak externals are output as XDEFs, initialized to zero, and placed in the vars section. This section can be renamed on a per-module basis with the -NIname option.</p>
-Xc (default)	<p>The -Xc option treats uninitialized global variables as weak externals or “C common.” The compiler does not allocate any space for these variables. The linker resolves each such variable to its initialized declaration, if one exists. For example:</p> <pre data-bbox="698 819 812 850">int i;</pre> <p>If this variable is not externally defined in another module using XDEF, the linker allocates the variable in the zerovars section. This section can be renamed at link time using the LNK68K ALIAS command. With the -Xc option, weak externals are output as XCOMs.</p>
-Xp	<p>The -Xp option allocates space for global variables that have not been explicitly initialized but assigns them no value.</p> <p>With the -Xp option, weak externals are output as XDEFs for which storage is defined but not initialized.</p> <p>The compiler places these variables in the zerovars section, which can be renamed on a per-module basis with the -NZname option. The zerovars section is set to zero at program start-up time.</p>

Optimizing Code

Generate Optimizer Information Messages

-i -ni (default)	<p>C++ only.</p> <p>Enables all available informational reports of the following compiler optimizations: constant propagation, dead assignment elimination, and tail recursion optimization. This option is equivalent to enabling the -ic, -id, and -it options. The -i option also implies that the -Og option is enabled.</p>
-----------------------------------	---

-ic -nic (default)	C++ only. Enables reports of constant propagation. When the -ic option is enabled and the compiler detects an occurrence of a constant propagation (a variable is replaced by a constant), the compiler issues an informational message.
-id -nid (default)	C++ only. Enables reports of dead assignment elimination by the compiler. When the -id option is enabled and the compiler detects a dead assignment (a variable is set but not used), the assignment is removed, and the compiler issues an informational message.
-it -nit (default)	C++ only. Enables reports of tail recursion optimization by the compiler. When the -it option is enabled and the compiler detects an occurrence of a tail recursion (a recursive call to a function is replaced by a jump), the compiler issues an informational message.

List of Code Optimizations

-O -nO (default)	The -O option turns on the following optimizations: <table> <tr> <td>-Oc</td><td>Optimizes stack processing</td></tr> <tr> <td>-Og</td><td>Optimizes globally</td></tr> <tr> <td>-Oi</td><td>Optimizes in-lining</td></tr> <tr> <td>-Ol</td><td>Performs local optimizations</td></tr> <tr> <td>-Or</td><td>Optimizes instruction scheduling</td></tr> </table> <p>-O is available only from the command line; it cannot be used in the #pragma option mode.</p> <p>This option does not turn on all optimizations.</p>	-Oc	Optimizes stack processing	-Og	Optimizes globally	-Oi	Optimizes in-lining	-Ol	Performs local optimizations	-Or	Optimizes instruction scheduling
-Oc	Optimizes stack processing										
-Og	Optimizes globally										
-Oi	Optimizes in-lining										
-Ol	Performs local optimizations										
-Or	Optimizes instruction scheduling										

Note

To turn on all optimizations, use the following options:

`-O -Ob -Oe -Os`

or

`-O -Ob -Oe -Ot`

To turn off all optimizations, use the following options:

`-nOc -nOl -nOR`

The rightmost option takes precedence if conflicting options are specified on the same command line.

-Ob

-nOb (default)

The **-Ob** option tells the compiler that it can assume that global variables will not be changed by aliases. If the global optimizer (**-Og**) is used and the **-Ob** option is specified, the compiler performs optimizations on global variables. No optimizations are performed on variables declared as **volatile**.

The optimizations performed assume that global variables can be changed only by an “unaliaised” reference to that variable. A variable is considered to be “aliased” if its address is assigned to a pointer variable.

For a loop invariant expression containing a global variable, the optimizer moves the expression out of the loop even if the loop contains a pointer dereference on the left-hand side of an assignment statement.

With the **-nOb** option, the compiler can still perform optimizations on a non-**volatile** variable if there is no pointer dereference on the left-hand side of the assignment statement within the loop in question. Note that **-nOb** does not have the same effect as declaring a variable to be **volatile**.

Warning

The **-Ob** option should not be used on programs that use global variables as operating system memory locations or I/O device registers.

If both the **-Ob** and **-nOg** options are specified, **-nOg** is ignored.

-Oc (default)
-nOc

The **-Oc** option eliminates unnecessary stack processing. The **-Oc** option does not pop the stack after each function call. The stack is allowed to grow and is popped after several function calls. This optimization is known as combining stack pops.

The **-nOc** option generates code to pop the stack after every function call.

-Oe
-nOe (default)

The **-Oe** option generates only one exit (return) from a function. By default, the compiler can generate more than one return instruction if that is more efficient.

-Og
-nOg (default)

The **-Og** option enables global optimization, which consists of the loop optimizer (**-OL**) and the optimizations invoked by the **-OX** options.

-Oi
-nOi (default)

The **-Oi** option enables the in-lining optimization.

In-lining is a code-generation technique that expands the code of a called function inside the caller. This technique eliminates the procedure call overhead (branching and stack allocation) and allows for the possibility of improved local code optimization such as constant folding and loop optimizations.

The **-Oi** option tells the compiler to prescan a module looking for functions, usually small functions with only simple local variables, where in-lining can enhance performance

without costing too much code space. When the module is compiled, the code for these functions is expanded in-line.

-Oi can be used in conjunction with **-OXi***number* to control the level of in-lining.

If an in-lined function invokes another function, that function can also be in-lined.

When considering functions for in-line optimization, preference is given to functions declared with the **inline** keyword.

-Oj (default)

-nOj

The **-Oj** option in-lines certain run-time library functions when Microtec Compiler extensions are enabled.

The **-nOj** option disables in-lining of run-time library functions without disabling other Microtec Compiler extensions.

Use the **-Oj** option in conjunction with **-x** options.

-OL

-nOL (default)

The **-OL** option performs the following optimization loops in the program:

- Forward code motion (**-OLf***number*)
Enables forward code motion within loops. **-OLf1** enables forward motion; **-OLf0** disables it.
- Constant hoisting (**-OLhc**)
Enables hoisting of constants out of a loop.
- Invariant code motion (**-OLhi**)
Enables hoisting of invariant expressions and statements out of a loop.
- Induction variable elimination (**-OLiv**)
Eliminates induction variables within loops.
- Same-family loop expression replacement (**-OLr**)
Replaces same family loop expressions with expressions using the same induction variable.
- Strength reduction of operators (**-OLs***number*)
Performs strength reduction within loops. The number can take the following values:
0: Do not perform strength reduction.

1: Perform strength reduction on expression with “nice” induction variables (not contained within an if statement within the loop).

2: Perform strength reduction on all induction variables.

- Loop exit condition testing at loop entry (**-OLt**)
Perform conversion of loop entry testing. Use only in conjunction with **-Ot**.
- Loop unrolling (**-OLulnumber**)
Perform loop unrolling. **-OLul1** enables loop unrolling; **-OLul0** disables it.

The **-OL** option turns on all of these sub-options. **-nOL** turns them all off. All **-OL** options without a suffix can be turned off with **-nOL** (for example, **-nOLiv**). The default is **-nOL**.

-Ol (default)
-nOl

The **-Ol** option performs local optimizations.

The **-nOl** option disables code hoisting and cross jump optimizations.

This option cannot be used in conjunction with the **-Og** option. If both the **-Og** and **-nOl** options are specified, **-Og** is ignored.

The **-nOl** option implies **-nOc**.

The **-nOl** option implies **-Kf**.

-OR (default)
-nOR

The **-OR** option allocates heavily used variables to registers.

If the global-flow optimizer (**-Og**) is used, local variables are allocated to registers using a register-coloring algorithm. If the global-flow optimizer is not used (**-nOg**), only heavily used local variables are allocated to registers (**-OR**).

Table 3-4 explains what happens when both **-Og** and **-OR** are specified.

Table 3-4. Interaction of **-Og** and **-OR** Options

Option	-nOg (Default)	-Og
-OR (Default)	Heavily used local variables are allocated to registers.	Local variables are allocated to registers using a register-coloring algorithm.
-nOR	Allocation of heavily used local variables to registers is disabled.	Local variables are allocated to registers using a register-coloring algorithm.

The **-nOR** option can be used to disable the allocation of heavily used variables to registers.

-Or

-nOr (default)

The **-Or** option enables instruction scheduling optimization. Instruction scheduling rearranges the instruction sequence for improved throughput in the instruction pipeline.

This scheduling reorders the instructions to minimize the hardware delays that sometimes occur between instructions because of the pipeline design of the architecture.

If you use the **-Or** option, the assembly output can be difficult to understand, and some debugging operations can yield confusing results.

The **-nOr** option disables instruction scheduling.

-Os

-nOs (default)

The **-Os** option optimizes in favor of code size rather than execution time.

-Ot

-nOt (default)

The **-Ot** option optimizes in favor of execution time rather than code size.

If neither the **-Os** nor **-Ot** option is used in conjunction with the **-Og** option, the compiler generates code that is optimized based on a compromise between size and speed.

-OTnumber

The **-OT** option specifies the number of temps that can be created during in-lining. The minimum is 256.

The default is **-OT2000**.

<i>-Ounumber</i>	<p>The -Ou option enables unreferenced function removal optimization, in conjunction with the corresponding linker option -Zu. It can take the following values:</p> <ul style="list-style-type: none"> 0 Do not perform any unreferenced function removal. 1 Perform unreferenced function removal on all functions except virtual member functions. 2 Perform unreferenced function removal on all functions. <p>This option is not invoked by the compiler driver, but directly by the user. The corresponding linker option -Zu is required.</p> <p>This option is for the C++ compiler only.</p>
<p>-OV -nOV (default)</p>	<p>The -OV option tells the compiler to treat all static data as if it were declared volatile. This includes global, file, and local variables.</p>
<i>-OXc number</i>	<p>The -OXc option performs the copy propagation optimization. It can take the following values:</p> <ul style="list-style-type: none"> 0 Do not perform the copy propagation optimization. 1 Perform the optimization only at the basic block level. 2 Perform the optimization across basic blocks. <p>The copy propagation optimization removes unnecessary definitions of the same value introduced by the statement <code>x=y</code>, if the compiler determines that the variable <code>x</code> has the same value as <code>y</code> for each and every use of <code>x</code>.</p> <p>The default is -OXc2 if -Og is on, -OXc0 otherwise.</p>
<i>-OXcs number</i>	<p>The -OXcs option performs the common subexpression elimination optimization. It can take the following values:</p> <ul style="list-style-type: none"> 0 Do not perform the common subexpression elimination optimization. 1 Perform the optimization only at the basic block level. 2 Perform the optimization across basic blocks. <p>There are instances when expressions of the form <code>x+y</code> are constantly recomputed in the code, with neither <code>x</code> nor <code>y</code> being redefined. These expressions are called common sub-expressions and their values can be computed in advance and</p>

stored. This computation is the common subexpression elimination optimization.

The default is **-OXcs2** if **-Og** is on, **-OXcs0** otherwise.

-OXii *number*

The **-OXii** option performs the function in-lining optimization. It can take the following values:

- 0 Do not perform the function in-lining optimization.
- 1 Consider in-line functions for the optimization.
- 2 Consider in-line functions and all small functions for the optimization.
- 3 Consider all functions for the optimization.

If **-Oi** is on, the default is either **-OXii1** (C++) or **-OXii2** (C). Otherwise, the default is **-OXii0**.

See the section *Function In-Lining* in Chapter 7, *Optimizations*, for more information.

-OXk *number*

The **-OXk** option performs the constant propagation optimization. It can take the following values:

- 0 Do not perform the constant propagation optimization.
- 1 Perform the optimization only at the basic block level.
- 2 Perform the optimization across basic blocks.

The constant propagation optimization is similar to that of copy propagation. In this case, a constant value defined at compile time is utilized as such for computation. The intermediate assignment to a variable is eliminated.

The default is **-OXk2** if **-Og** is on; otherwise, the default is **-OXk0**.

-OXrep *number*

The **-OXrep** option specifies the number of times to perform the constant propagation and common subexpression optimizations. It can take a value from 0 through 9.

The default is **-OXrep1** if **-Og** is on; otherwise, the default is **-OXrep0**.

-OXrf
-nOXrf (default)

The **-OXrf** option performs the factorization optimization, which allocates heavily used constants and addresses into

	any available register after all other optimizations have been performed.
-OXsr -nOXsr (default)	The -OXsr option utilizes the unused scratch registers to store heavily accessed variables.
-OXt -nOXt (default)	The -OXt option performs the optimization of tail recursive functions. When the last statement executed in a function body is a recursive call to the same function, the call is said to be tail recursive. The tail recursive call can be optimized to utilize the caller's function stack, speeding up the program execution.
-Oz38 -nOz38 (default)	The -Oz38 option prevents JSR directives from being converted to BSRs on the 68040 processor when the -Mca option is specified.

Producing Minor Code Generation Variations

-KAH (default) -KAO -KAP	The -KAO option tells the compiler to assume all objects pointed to by pointer variables are properly aligned; -KAP tells the compiler to assume they are misaligned. -KAH is the compiler heuristic.
-Kc (default) -nKc	<p>The -Kc option tells the compiler to use the CLR instruction to assign zero to register and memory variables.</p> <p>The -nKc option tells the compiler not to use the CLR instruction to assign zero to nonregister variables. On some 68K family processors, the CLR instruction reads memory before writing a zero to the location. This behavior can cause problems for memory-mapped I/O ports.</p>
-KFi -nKFi (default)	<p>The -KFi option controls the saving of the FPU state when the interrupt keyword is used. The compiler generates code to store and restore the FPU state on an interrupt.</p> <p>With the -nKFi option, the compiler will not save the FPU state on an interrupt.</p>

-Kf

-nKf (default)

The **-Kf** option controls function frame usage.

The **-Kf** option forces frames for all functions. A framed function has a **LINK/UNLK** pair of instructions in the function prologue and epilogue. In a framed function, variables and parameters are accessed on the stack with the FP-relative addressing mode. The stack pointer is not, however, available for your use; user programs should never modify the stack pointer because this can cause unpredictable run-time errors.

With the **-nKf** option, the compiler generates a frame for a function only if the frame is needed.

-Khreg[,reg] . . .

The **-Khreg** option directs the compiler to reserve the registers specified. The compiler does not generate any code that uses a reserved register.

The value for *reg* can be up to three registers from **A2** to **A6** and three registers from **D2** to **D6**.

If you enter **-Khreg** more than once on the command line, the last occurrence of **-Khreg** overrides previous occurrences.

For example, if you enter:

```
-Kha2 -Kha3
```

-Kha2 is discarded, and the compiler does not generate any code that uses the **a3** register.

The compiler generates a warning if the reserved register is required to adhere to function calling conventions.

-KI

-nKI (default)

The **-KI** option initializes all uninitialized local variables to zero. This option does not affect the initial value of local variables that have been explicitly initialized in your code. This option does not affect parameters.

The **-KI** option is designed to aid debugging; use of this option can significantly decrease run-time performance for functions containing a large number of local variables or a large local variable.

	With the -nKI option, no additional code is generated to initialize local variables. User-specified initializations are performed as usual.
-KJ -nKJ (default)	The -KJ option generates switch statement jump tables in a data section (vars).
-Km (default) -nKm	<p>The -Km option generates messenger symbols.</p> <p>The -nKm option suppresses messenger symbols.</p> <p>For more information on messenger symbols, see Chapter 13, <i>Embedded Environments</i>.</p>
-KP -nKP (default)	The -KP option packs structs and unions . When -KP is used with -nZle , enumerations are allocated as the smallest integral type that can hold all their values, rather than int by default.
-Kq -nKq (default)	<p>The -Kq option gives all double variables or constants the size (32 bits) and alignment (4 bytes) of type float. Since -Kq treats long double variables as double, all variables that are float, double, and long double have the size and alignment of type float.</p> <p>If you want to use single-precision math, you must rebuild your libraries with the -Kq option enabled. All of the new libraries must then be linked to all programs built using -Kq.</p>
-Kr -nKr (default)	The -Kr option specifies that the return-from-procedure (rather than the return-from-interrupt) instruction is used to return from a function declared as an interrupt procedure. The -Kr option uses the RTS instruction, and the -nKr option uses the RTE instruction.
-Ksnumber -Ks-1 (default)	The -Ks option checks whether the stack is overflowing at function entry. <i>number</i> is the number of extra bytes acceptable beyond the size of the current stack frame. -1 indicates stack-checking is turned off.

-Kt**-nKt** (default)

The **-Kt** option generates code to tag each function's entry and exit points. These tags can be used by real-time analysis tools to monitor the execution of the program.

For each function, the compiler allocates two words of data in the **tags** section. The tag data is labeled `_r_module_func`, where *module* is the name of the module and *func* is the name of the function. For example:

```

                                SECTION      tags,,D
                                XDEF          _r_main
__r_main:                      DCB.B        4,0

```

The compiler generates the following code in the prologue (function entry):

```
move.w    #10,_r_main
```

The compiler generates the following code in the epilogue (function exit):

```
move.w    #100,_r_main+2
```

The values 10 and 100 are arbitrary. The analysis tool detects and records writes to the tag data.

-Ku**-nKu** (default)

The **-Ku** option tells the compiler to consider **char** variables declared without the **signed** or **unsigned** keyword as **unsigned**. The preprocessor symbol `_CHAR_UNSIGNED` is predefined.

With the **-nKu** option, **char** variables without an explicit **signed** or **unsigned** keyword are treated as **signed**, and the preprocessor symbol `_CHAR_SIGNED` is predefined.

-Kv**-nKv** (default)

The **-Kv** option tells the compiler to consider bit fields declared without the **signed** or **unsigned** keyword as unsigned.

With the **-nKv** option, bit fields without an explicit **signed** or **unsigned** keyword are treated as signed.

-Ze**-nZe** (default)

The **-Ze** option pushes all parameters of size **char** onto the stack in even memory address values.

Use the **-Ze** option in conjunction with the **-Zp2** option.

-Zle (default) -nZle	C++ only.
	The -Zle option instructs the compiler to make structure layout compatible with that produced by MCC68K 4.5 and earlier. The -KP option does not affect enums in this mode.
-Zm <i>number</i>	The -Zm option instructs the compiler to make any packed structure size a multiple of <i>number</i> . The default value of <i>number</i> is 1. Unpacked structures are not affected.
-Zn <i>number</i>	The -Zn option instructs the compiler to make any unpacked structure size a multiple of <i>number</i> .
-Zo (default) -nZo	The -Zo option pushes all parameters of size char onto the stack in odd memory address values.
	Use the -Zo option in conjunction with the -Zp2 option.
-Zp2	The -Zp2 option pushes all parameters smaller than int onto the stack as 16-bit quantities. A parameter smaller than short is extended to short .
	If no function prototype is used to indicate parameter type, parameters are extended to int by default.
-Zp4 (default)	The -Zp4 option extends to int all parameters smaller than int before pushing them on the stack.
	If no function prototype is used to indicate parameter type, parameters are extended to int by default.

Generating Position-Independent Code and Data

-Mca (default)	The -Mca option directs the compiler to use absolute addressing for all code references. If this option is specified for the 68040 processor, all JSR directives are converted to BSRs unless the -Oz38 option is also specified.
-Mcp	The -Mcp option directs the compiler to use PC-relative addressing for all code references. This results in code that is position-independent. The preprocessor symbol _PIC is predefined.
-Mdn	The -Mdn option directs the compiler to use register-relative addressing for all references to static data. This results in

data that is position-independent. The preprocessor symbols `_PID` and `_PID_REG` are predefined. `_PID_REG` contains the name of register number *n* in upper-case letters.

The **-Mdn** option directs the compiler to use **An**-relative addressing. The register number *n* must be in the range 2 to 6. If you use the **A5**-relative addressing option, link with one of the **A5**-relative libraries. For data generated relative to other valid registers, you must build a corresponding library.

-Mda (default) The **-Mda** option directs the compiler to use absolute addressing for all data references. If you use this option, link with one of the absolute libraries.

-Mdp The **-Mdp** option directs the compiler to use PC-relative addressing for all references to static data. This results in data that is position-independent. The preprocessor symbol `_PID` is predefined.

If you use this option, link with one of the PC-relative (**ir**) libraries.

Note

The options **-Mca** and **-Mcp** are mutually exclusive.

The options **-Mda**, **-Mdp**, and **-Mdn** are mutually exclusive.

The rightmost option takes effect if conflicting options are specified on the same command line.

-Ml The **-Ml** option directs the compiler to use 32-bit displacements for **An**- or PC-relative addressing. This option is the same as **-Mlc** and **-Mld** together.

By default, the compiler uses 16-bit displacements for **An**- or PC-relative addressing.

-Mlc The **-Mlc** option directs the compiler to use 32-bit displacement for **An**- or PC-relative addressing for code references only.

- Mld** The **-Mld** option directs the compiler to use 32-bit displacement for **An-** or PC-relative addressing for static data references only.
- Mls** The **-Mls** options directs the compiler to use 32-bit displacement in **switch** statement jump tables.

Choose Address Mode for Sections

The compiler and linker allocate code and data to the sections listed in Table 3-5.

Table 3-5. UNIX/Windows Allocation of Code and Data

Contents	Section Name	Section Type
Code	code	code
Compiler-generated C++ exception-handling data	cxx_edt	data
Compiler-generated C++ initializers for position-independent virtual tables	pixinit	code
Compiler-generated C++ run-time type information	cxx_rtti	data
Compiler-generated C++ static constructor/destructor	initfini	code
Compiler-generated literals	literals	data
Compiler-generated tag data	tags (-Kt option only)	data
Constant variables	const	data
Initialized data	vars	data
Pointer to the start of the section	heap	data
Pointer to the end of the section	stack	data
Strings	strings	data
Uninitialized data	zerovars	data

Items in code sections are referenced according to the address mode chosen with the **-Mc** options (**-Mca** or **-Mcp**).

Items in data sections are referenced according to the address mode chosen with the **-Md** options (**-Mda**, **-Mdp**, or **-Mdh**).

Use the **-a** option to control whether a section name is of type **data** or **code** (or **rom**). The **-N** option controls the name used by the assembler and linker to refer to the section.

Reference items in the remaining sections (**vars**, **const**, **strings**, and **literals**) according to the address mode chosen with the options **-Mc** or **-Md**. If a section is to be placed in ROM (with **code**), reference its items according to the **-Mc** options. If a section is to be placed in RAM (with **zerovars** and **vars**), reference its items with the **-Md** options.

-acc (default)

-acd

The **-acc** option references items in the **const** section according to the **-Mc** option.

The **-acd** option references items in the **const** section according to the **-Md** option.

-aic

-aid (default)

The **-aic** option references items in the initialized data section according to the **-Mc** option.

The **-aid** option references items in the **vars** section according to the **-Md** option.

-alc (default)

-ald

The **-alc** option references items in the **literals** section according to the **-Mc** option.

The **-ald** option references items in the **literals** section according to the **-Md** option.

-asc

-asd (default)

The **-asc** option references items in the **strings** section according to the **-Mc** option.

The **-asd** option references items in the **strings** section according to the **-Md** option.

-azc

-azd (default)

C++ only. The **-azc** option allocates items in the **zerovars** section according to the **-Mc** option.

The **-azd** option allocates items in the **zerovars** section according to the **-Md** option.

Name the Sections

The compiler allocates code and data to sections. Default section names are overridden by these options.

- | | |
|------------------------|--|
| -NC <i>name</i> | <p>The -NC option sets the constant variables section name.</p> <p><i>name</i> can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).</p> <p>The compiler does not generate an error message if you specify an illegal name.
(default: const)</p> |
| -NI <i>name</i> | <p>The -NI option sets the initialized data section name.</p> <p><i>name</i> can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).</p> <p>No error message is generated by the compiler if you give an illegal name.
(default: vars)</p> |
| -NJ <i>name</i> | <p>The -NJ option sets the name of a section where pointers to static constructors are located.</p> <p>If a <i>name</i> other than the default is chosen, the header file targ_mac.h (referenced by the start-up object file cxxconde.cxx) needs to be modified to reflect the new name.
(default: initfini)</p> |
| -NL <i>name</i> | <p>The -NL option sets the compiler-generated literals section name.</p> <p><i>name</i> can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).</p> <p>No error message is generated by the compiler if you give an illegal name.
(default: literals)</p> |
| -NM <i>name</i> | <p>The -NM option sets the module name.</p> <p><i>name</i> can be any sequence of characters.</p> |

No error message is generated by the compiler if you give an illegal name.

(default: source filename stripped of any suffix or prefix)

-NS*name*

The **-NS** option sets the string section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).

No error message is generated by the compiler if you give an illegal name.

(default: **strings**)

-NT*name*

The **-NT** option sets the code section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).

No error message is generated by the compiler if you give an illegal name.

(default: **code**)

-NZ*name*

The **-NZ** option sets the uninitialized static data section name.

name can be one or two decimal digits or a symbol beginning with an alphabetic character, question mark (?), period (.), or underscore (_).

No error message is generated by the compiler if you give an illegal name.

(default: **zerovars**)

Note

If you compile your C++ source file with the **-Xc** option, the linker allocates all uninitialized global data in the **zerovars** section. This section can be renamed at link time using the LNK **ALIAS** command. See the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference Manual for the 68000 and ColdFire Families* for more information.

If you compile your C++ source file with the **-Xp** option, you can rename the **zerovars** section at compile time with the **-NZ** option.

Enabling C++ Extensions

+array_new_and_delete (default)

+no_array_new_and_delete

C++ only.

The **+array_new_and_delete** option enables the **array new** and **array delete** operators. The **+no_array_new_and_delete** option disables these operators.

+bool (default)

+no_bool

C++ only.

The **+bool** option enables the **bool** keyword. The **+no_bool** option disables the **bool** keyword.

Suspension of Cleanup

+delete_std

C++ only.

Cleanup action is not generated for file scope and function scope static variables. This option saves space in programs that never terminate.

Control C++ Virtual Table Generation

+e

+ne

C++ only.

There are three options controlling virtual table generation: the default option (which cannot be specified explicitly), the **+e** option, and the **+ne** option.

By default, the compiler generates a virtual function table in the module that defines the first non-in-lined and non-inherited virtual function.

The **+e** option generates references to existing virtual tables (**.xref** files). This avoids the conflict of trying to create duplicate external table definitions.

The **+ne** option generates external declarations of virtual tables as **.xdef** files. The compiler produces virtual tables for classes that either contain virtual functions or are derived from virtual base classes. This saves space, but duplicate tables defined with this option will cause a conflict. The **+e**

option should be used with no more than one file per executable.

If all non-inherited virtual member functions of a class are declared as in-line, the virtual function table for that class is generated as a local object in every module that requires it.

Specify Language

<code>+lac++</code> (default)	C++ only. The +lac++ option specifies C++ without anachronisms, as described in <i>The Annotated C++ Reference Manual</i> .
<code>+lc++</code>	C++ only. The +lc++ option specifies C++ with anachronisms, as described in <i>The Annotated C++ Reference Manual</i> .
<code>+lf2</code>	C++ only. The +lf2 option specifies (strict) Cfront 2.1 compatibility.
<code>+lf2o</code>	C++ only. The +lf2o option specifies Cfront 2.1 compatibility with anachronisms.
<code>+lf3</code>	C++ only. The +lf3 option specifies (strict) Cfront 3.0 compatibility.
<code>+lf3o</code>	C++ only. The +lf3o option specifies Cfront 3.0 compatibility with anachronisms.

Control Template Instantiation

The **+tl**, **+tm**, and **+tu** options control the generation of template instances. They are effective only if your program uses templates. These options are mutually exclusive. The last option specified on a line overrides the other options on that line.

<code>+tl</code>	C++ only. The +tl option generates all instances of template functions as local functions; static data members of template classes are still generated as public and are shared among modules.
------------------	--

If you use the same template function in multiple files compiled with **+tl**, you end up with duplicated instances in your final program.

+tm

C++ only.

The **+tm** option disables automatic generation of template instances; that is, template instances are only generated when they are explicitly instantiated in the program. This avoids duplication of code at compilation time. When **+tm** is used, template definitions for the templates that are not generated need not be included.

For example, you could compile only one file with **+tu** to get all template instances and compile all other files with **+tm**.

+tu (default)

C++ only.

The **+tu** option automatically generates all necessary template instances in a compilation unit, except those that are manually disabled, and puts them into the object file. You need to include all necessary template declarations and definitions in your source file. Duplicated template instances in object files are removed at link time.

Enable C++ Features

-zc

The **-zc** option allows the use of double slashes (*//*) to denote comments. In C++, this is on by default; in C, it is off by default.

-ze

-nze (default)

In C++, the **-ze** option enables C++ exception handling in C++ code. For further information on exception handling, see Chapter 10, *Interlanguage Calling*. When the **-ze** option is specified, the macro **_CPLUSPLUS_EHS** is predefined as 1; otherwise, it is predefined as 0. Libraries supporting exception handling can be found in *install_dir/lib/ze*.

The **-nze** option turns off exception handling. Libraries not supporting exception handling can be found in *install_dir/lib/nze*.

In C, the **-ze** option turns on the **-Kf** option.

+rtti

+no_rtti (default)

C++ only.

The +**rtti** option enables the **typeid** and **dynamic_cast** keywords. The +**no_rtti** option disables these keywords.

+wchar_t (default)

+no_wchar_t C++ only.

The **+wchar_t** option enables the **wchar_t** keyword. The **+no_wchar_t** option disables the **wchar_t** keyword.

Accept Nonstandard C++ File Suffix

+zsuffix C++ only.

The **+zsuffix** option lets you specify a nonstandard C++ file-name suffix. Any files with this suffix are compiled as C++ files. For example, specifying **+zc** (without a period) treats **.c** files as C++ code.

Using Options

This section discusses how to use some of the compiler options to get the most out of the Microtec C/C++ compilers.

Option Combinations

Table 3-6 shows some common combinations of options.

Table 3-6. Common Option Combinations

Options	Result
-ng -nOg -nOl	Fastest compilation (no debugging information, no global optimization)
-O -Ot	Fastest generated code (advanced optimizations, optimized for time rather than space)
-O -Os	Smallest generated code size (advanced optimizations, optimized for space rather than time)
-x -A	Broadest C language definition accepted (both Microtec Compiler and ANSI extensions enabled)
-x +lc++	Broadest C++ language definition accepted (both Microtec Compiler and ANSI extensions enabled, but without ANSI C++ rules)

ANSI Extensions (C only)

By default, the C compiler enables both the ANSI (-A) and Microtec Compiler extensions (-x). When the ANSI compiler option is disabled, the compiler attempts to behave like a traditional C compiler (K & R). However, the preprocessor is ANSI in nature, so its evaluation mechanisms are different from pre-ANSI preprocessors.

Using `__STDC__`

The `__STDC__` macro is defined by all ANSI-compliant compilers to indicate that the ANSI Standard C language definition is being enforced. Use of the `-nA` option undefines this macro.

The `__STDC__` macro lets you identify sections of your code so that different actions can be performed based upon whether ANSI checking is enforced.

Example:

```
#ifdef __STDC__
    int other_name;
#else
    int const;
#endif
```

This example declares a variable named `const` when ANSI rules are not in effect. When ANSI rules are applied (`__STDC__`), `const` is recognized as a keyword, so the variable cannot be named `const`.

Function Prototyping

Function prototypes let you include parameter type information within C function declarations. With access to such information, the C compiler can detect mismatched parameter passing in terms of parameter type, size, and number. This optional enforcement of parameter type checking detects some of the most common C programming errors during compilation and helps create problem-free code.

Standard library headers include ANSI and pre-ANSI declarations, which are selectively included by setting the ANSI compiler option. This mechanism can be copied if you do not want to give up ANSI features or lose portability with pre-ANSI compilers.

Example:

```
#ifdef __STDC__
extern int remove (const char *filename);
#else
extern int remove ();
#endif
```

Function prototypes are accepted regardless of the ANSI compiler option. If the compiler is invoked while ANSI is specified to not be enforced, prototype information is flagged with an informational **(I)** diagnostic message.

ANSI and Floating-Point Variables

If the compiler is invoked with the ANSI compiler option, **float** variables are not converted to **double** unless necessary.

Example:

```
#ifdef __STDC__
    assert(sizeof(float)==4);
#else
    assert(sizeof(float)==sizeof(double));
#endif
```

This example shows the ANSI C definition (**__STDC__**) enforcing **float** values to be 32 bits. If the assertion is untrue (that is, **float** values are not implemented as 32 bits), the program terminates.

If you are migrating to an ANSI compiler, be careful if software floating-point libraries are used without a floating-point unit (FPU) available. Use **float** instead of **double** (32 bits instead of 64 bits), to improve efficiency.

Example:

```
1      #pragma option -nQ
2      float a,b;
3      foo()
4      {
5          a = b+2.0;
              ^
>> (I) [ANSI] inefficient evaluation in "double" because of
"double" constant
6      }

1 Informational
```

The informational message in this example indicates that the **double** operand (that is, the constant 2.0) inhibits the use of simpler floating-point arithmetic. The variable **b** is promoted to **double** for the purposes of evaluation, which may generate less than optimal code. You can modify the source file to impose the type **float** on the constant either by using the ANSI qualifier **f** or by casting:

```
a = b+2.0f;          /* ANSI suffix denotes a float constant */
a = b+(float)2.0; /* casting of the constant to float */
```

Microtec Compiler Extensions

The **-x** option activates Microtec Compiler extensions to the C language that ease embedded and systems programming. These extensions include:

- The **asm()** pseudofunction to in-line assembly code
- The **interrupt** keyword for interrupt procedures
- The **packed** keyword to avoid padding in structures and reduce the size of small enumerations
- The support of **char**, **short**, and **long** bit fields
- In-lining of several library functions (see the **-x** option in section *Command Line Options — Summary* in this chapter for more information)

Table 3-7 lists the additional Microtec Compiler extension keywords recognized by the C compiler regardless of whether the **-x** option is enabled. Other identifiers cannot conflict with them.

Table 3-7. Keywords (Microtec Compiler Extensions)

Keyword	Description
interrupt	Declares a function as an interrupt handler.
packed	Reduces the amount of storage needed by not forcing padding between members. Used in struct , union , and enum declaration statements.
typeof	Determines the data type of a variable.
unpacked	Forces normal padding between members. Used in struct , union , and enum declaration statements.

typeof() Operator

The **typeof()** operator is a Microtec Compiler extension that can find the type of any variable. It can be used anywhere a type is needed. For example, the following macro can be used to swap two variables of the same type.

```
#define swap(x,y) do { \
    typeof(x) tmp = x; \
    x = y; \
    y = tmp; \
} while (0)

int a, b;
struct foo c, d;

main()
{
    swap(a, b);
    swap(c, d);
}
```

Exception Handling

You must use the **-ze** option to compile all C and C++ modules that use C++ exception handling or that might call a C++ function that throws exceptions.

When the **-ze** option is specified, the C compiler (invoked by the C++ driver) creates stack frame pointers for functions in the given C file. This information is used by the C++ exception handling system. For further information on C++ exception handling, refer to Chapter 10, *Interlanguage Calling*.

Customizing the Compilation Driver

The compiler features a powerful compile/assemble/link driver that automatically invokes the appropriate product. The driver simplifies compiler invocation, and can be used to customize your compilation environment.

Compiler options are read from left to right. If an option is specified multiple times on the command line, the last occurrence of that option is enforced. Options specified apply to all files being compiled (the position of filenames on the command line is not significant).

Example:

```
ccc68k -g -l -Flp44 main.cc read.s scr.lib -ng -Flp100 -g -ng
```

This example compiles `main.cc` without debugging information (the rightmost option is `-ng`) and uses a listing page length of 100 lines (the rightmost option is `-Flp100`).

Example:

```
#!/bin/csh
set extraoptions="$argv[2- $#argv]"
ccc68k $1 -c -g -O -Flp44 $extraoptions
```

This UNIX C shell script, named **compile**, assigns all the invocation arguments that follow the first argument (the filename) to **extraoptions**:

<code>compile test.cc</code>	Compiles <code>test.cc</code> with normal options
<code>compile test.cc -g -l</code>	Compiles <code>test.cc</code> with normal options plus <code>-g</code> and <code>-l</code> options
<code>compile test.cc trick.cc</code>	Compiles <code>test.cc</code> and <code>trick.cc</code> with normal options

If you are not satisfied with the default settings of the options, use the driver option **-dfilename** to customize your compilation. This option also lets a system administrator establish installation defaults.

The driver reads the lines in *filename* as if they were specified on the command line. The file can be several lines long; the new-line character is read as white space.

Example:

```
-g
-Flp66
-e /usr/misc/myproject_linkfile
-DDEBUG_MODE -DHOST=SUN4
```

If the file **/usr/misc/myproject_defaults** contains the lines shown in the preceding example, and if **-d/usr/misc/myproject_defaults** is specified on the command line, then all these options are applied to the compilation. Alternatively, a shell **alias** command can be used to force customization:

```
alias ccc68k "ccc68k -d/usr/misc/myproject_defaults"
```

Pragmas and Options

With the **#pragma option** construct, you can set local compiler options that are normally set on the command line within the source file itself. These options override options on the invocation line because they are the last encountered and affect the compilation of the entire file, not just the code after their occurrence.

Note that not all options have meaning in a **#pragma option** construct. This construct is used mostly for code generation options. File generation options such as **-t**, **-o**, and **-s** should only be used at the command line.

Example:

```
#pragma option -g
funct1()          /* always include debugging info */
...
#pragma option -ng
funct2()          /* do not include debugging info */
...
```

If the lines shown in this example occur in the same file, the compiler applies the last option encountered (`-ng`) to the entire file. The compiler does not apply different options to `funct1` and `funct2`. The options cannot be turned on and off to affect only portions of a source file.

A typical use of **#pragma option** is with makefiles that compile a related group of sources. One of the files in this group might require particular options, such as optimizing for space or eliminating debugging information. The **#pragma option** directive lets you specify command line options that apply to that file only:

```
#pragma option -ng          /* No debugging info for this file */
```

Driver options such as **-S**, **-e**, or **-l** are not accepted because the driver decides how to proceed before reading any input files, and all driver options have been processed prior to reading your file. The same happens with listing control options and preprocessor options. Generally, it is safe to give options that affect debugging, optimizations, and code generation.

Example:**modulea.c:**

```
#pragma option -NTcode_a -NIdata_a -NCdata_a -NSdata_a
#pragma option -NLdata_a
```

moduleb.c:

```
#pragma option -NTcode_b -NIdata_b -NCdata_b -NSdata_b
#pragma option -NLdata_b
```

This example shows two separate modules with the **#pragma option** directive declaring different names for compiler-generated sections within the different modules. By specifying this information in the separate modules, the sections can be located separately at link time.

Locating Header Files

If you need to know the exact origin of header files included during a compilation, use the **-E** compiler option. It invokes the preprocessor and sends an annotated listing to the standard output device.

Example:

```
#line 1 "input.c"
#line 1 "./x.h"

extern foo(struct tag *p);

#line 2 "input.c"
#line 1 "/usr/mri/include/ccc68k/stdio.h"

extern struct iobuf {
    ...
```

This example shows preprocessor output from the command line:

```
ccc68k -E input.c
```

for a file `input.c` containing:

```
#include "x.h"
#include <stdio.h>
```

By viewing the full pathnames of the include files used, you can eliminate any uncertainty about the actual header files used with your program. Refer to the **-J** compiler option for information on overriding the standard include file (<>) location.

Finding Source Files

It is very common to keep sources and objects in the same directory. However, the Microtec compilers also let you maintain different object versions of those sources.

Example:

```
cd /mydir/no_debug_dir
ccc68k -ng /mydir/new/*.c
cd /mydir/debug_dir
ccc68k -Gf -g /mydir/new/*.c
```

This example shows how you can leave your source files in one directory and create special object versions of those files in different directories. In this case, the source files are located in the directory `/mydir/new`. Before compilation, two different subdirectories were created: `/mydir/debug_dir` and `/mydir/no_debug_dir`.

Once you move to the appropriate subdirectory, you can invoke the compiler with the path name to the source files.

To generate object files that do not contain debugging information, specify the `-ng` option with the path name to the source files. The resulting object files do not contain debugging information and are in the current directory `/mydir/no_debug_dir`.

To generate object files that contain debugging information, specify the `-g` and `-Gf` options with the path name to the source files. The `-Gf` option places full path names in the debug file, making it possible for the XRAY Debugger to locate the source files. The resulting object files contain debugging information and are in the current directory `/mydir/debug_dir`.

Your source files remain in their original directory (`/mydir/new`).

Determining Option Defaults

The options that you specify for a given compilation represent only a subset of all the options applied to your file(s). Many compiler options are turned on by default, and you should be aware of how these defaults can affect your compilation.

An easy way to determine which options are in effect is to compile an empty file and produce a listing file:

```
ccc68k -l -y empty.c
```

All of the options in force are listed. The `-y` option limits the compiler to checking the syntax of the program so that errors produced by an empty file are kept to a minimum.

Producing Listing Files

The listing files produced by the compiler are intended for line printer output with a default page length of 55 lines. You can change this number with the **-Flpnumber** option. Specifying **-Flp0** tells the compiler to prepare the listing for a continuous printout with no page breaks.

The **#pragma list** directives can turn the listing option on and off. Because these directives apply to code following the directive, you can turn the listing option on and off several times within the same file.

Example:

```
foo() {  
#pragma list off  
    int invisible, visible;  
    invisible=1;  
#pragma list on  
    visible=99;  
#pragma list resume  
    invisible=1;  
#pragma list resume  
    visible=99;  
}
```

The listing produced from this example is:

```
foo() {  
#pragma list off  
    visible=99;  
#pragma list resume  
    visible=99;  
}
```

The **#pragma list resume** directive restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive. The listing option **-l** must be specified for the **#pragma list** directives to have any effect.

To produce a listing file that contains both assembly and C++ code, use the **-Fsm** compiler option (C++ code appears in the assembly source file as comment lines):

```
ccc68k -S -Fsm hello.c
```

Command Line Examples

This section gives examples of the use of the compiler and driver.

For more information about the libraries described in these examples, see Chapter 5, *Using Libraries*, in this guide.

Example:

```
ccc68k program.cc
```

This command compiles the C++ source file `program.cc` to produce the object file **program.o** (UNIX) or **program.obj** (Windows). The object file is then linked with the default linker command file and the default library to produce the executable file **program.x** (UNIX) or **program.abs** (Windows).

Example:

```
ccc68k module1.cxx module2.cc module3.cc -l listing
```

This command compiles three C++ source files `module1.cxx`, `module2.cc`, and `module3.cc` to produce three object files: **module1.o**, **module2.o**, and **module3.o** (UNIX) or **module1.obj**, **module2.obj**, and **module3.obj** (Windows). A compiler-generated listing of each source file is written to the file `listing` (`-l` option). The object files are then linked with the default linker command file and library to produce the executable **module1.x** (UNIX)/ **module1.abs** (Windows). The name of the executable file comes from the first source file that appears on the command line with the default extension added. Here, the object files are not deleted; the object file is deleted only if a single source file is successfully linked.

Example:

```
ccc68k -c module.s
```

This command assembles the assembly file `module.s` to produce the object file **module.o** (UNIX) or **module.obj** (Windows). The `-c` option prevents the driver from linking this module to produce an executable file.

Example:

```
ccc68k main.cc asmfile.s extra.o utils.lib
```

This command:

- Compiles `main.cc` to produce **main.o** (UNIX) or **main.obj** (Windows).
- Assembles `asmfile.s` to produce **asmfile.o** (UNIX) or **asmfile.obj** (Windows).
- Links the object files for `main.cc` and `asmfile.s` with the default linker command file, the default library, and `extra.o` to produce an executable file named **main.x** (UNIX) or **main.abs** (Windows). Modules from `utils.lib` are also linked if necessary.

Example:

```
ccc68k -ze -c file1.c
```

This command compiles the C file `file1.c` so that it contains the necessary information for handling exceptions. An alternative method is to compile all the application files with the following command line:

```
ccc68k -o application -ze main.cc file1.c
```

The C++ compiler automatically links the appropriate C++ run-time library that contains the support for run-time exception handling.

Microtec C++ Compiler Extensions 4

This chapter describes Microtec Compiler extensions to the C++ language: keywords, including file optimization, and support for assembler statement intermixing with code.

Keywords

The Microtec C++ compiler supports the following keywords:

- **interrupt**
- **packed**
- **unpacked**

interrupt

The **interrupt** keyword instructs the compiler to generate a special function prologue and epilogue. The function prologue saves the scratch registers used within the function; the function epilogue restores the contents of these registers and generates an appropriate “return-from-interrupt” instruction. The compiler issues a warning if an interrupt function is declared with parameters or with a return value other than void.

An interrupt function is declared as follows:

```
interrupt void int_handler(void);
```

The function takes no arguments, since it will be entered as the result of a machine-specific interrupt and not as the result of a normal function call sequence. The user is responsible for installing the address of the function in interrupt vectors.

By default, a function or a friend function declared as an interrupt handler returns to the calling function with the return-from-interrupt instruction rather than with the ordinary return instruction. However, if the **-Kr** compiler command line option is specified, the interrupt handler returns with an ordinary return instruction.

Example:

```

// This example illustrates how to declare a C++ interrupt
// handler and how to use it with C++ classes and associated
// access rules.
typedef void (*FUNC) ();
const int UPAGECNT = 20;
struct Page;

// class SegU_Info contains some information necessary during
// signal interrupt processing or handling.
//
// class SegU_Info also permits a user interrupt handler
// (that is, void handler()) to access its private members.

class SegU_Info {
    long uaddr[UPAGECNT];
    struct Page *diskaddr[UPAGECNT];
    static int user_handler_exist;
    FUNC catcher;
public:
    friend interrupt void handler (void);
};

// a handle to SetU_Info:
class SegU_Info segu_handle;

// This flag indicates whether or not a user handler exists
int SegU_Info::user_handler_exist = 1;

// This is a very simple sample of user interrupt handler.
// The key points to illustrate are:
//     - The handler takes NO argument and returns NOTHING.
//     - The handler, as a friend function to class SegU_Info,
//       can access members (including private members
//       such as SegU_Info::catcher) of class SegU_Info.
// The purpose of this example is not to illustrate general
// interrupt handler algorithms and similar devices.
interrupt void handler (void) {

    /* some code */

    if (SegU_Info::user_handler_exist) {
        // Clear user signal catcher
        segu_handle.catcher = (FUNC) 0;
    }
    /* some code */
}

```

This example declares an interrupt function handler that uses the return-from-interrupt return convention. An interrupt handler cannot be a member function

because an interrupt handler must not have any arguments. An interrupt handler can be a friend function without arguments or a static member function without arguments, because neither has a **this** pointer (or object handle), although either can be scoped within a class like the static member function.

packed, unpacked

The **packed** and **unpacked** keywords are Microtec Compiler extensions to the C and C++ languages. These keywords can be used with structure (**class**, **struct**, and **union**) and **enum** declarations to control memory requirements. For more information on **packed** and **unpacked** data objects, refer to Chapter 11, *Internal Data Representation*.

Structures that are declared **packed** do not have padding bytes added between members. Bit-field straddling is enabled for **packed** classes. By default, structures are considered **unpacked** unless the **-KP** option is used.

A **packed enum** differs from an **unpacked enum** in that **packed enum** structures are represented by the smallest integral type that can represent all the enumerators, whereas **signed int** is used to represent **unpacked enum** structures. By default, an **enum** is **unpacked**, unless the **-KP** option is used.

A **class** or **enum** can be declared either as **packed** or **unpacked**, but not as both. Multiple declarations of the same **class** or **enum** tag should all be either **packed** or **unpacked**.

Example:

```
unpacked struct packed_struct;
packed struct packed_struct {
// Error: Inconsistent re-declaration
// structure definition
};
```

Declarations without explicit **packed** or **unpacked** qualifiers do not conflict with other declarations.

Example:

```
packed struct no_explicit_qualifier;
struct no_explicit_qualifier; // Not an error
struct no_explicit_qualifier { // This structure is packed
// structure definition
};
```

During **class** definition, if none of the previous declarations has specified a **packed** or **unpacked** qualifier, the default qualifier is assumed, which means that later dec-

larations cannot specify a **packed** or **unpacked** qualifier that is different from the default.

Example:

```
// Assume the -KP option was not specified.
// This structure is unpacked by default.
struct unpacked_struct {
    // structure definition
};
packed struct unpacked_struct ps1;
// Error, unpacked_struct is unpacked !
unpacked struct unpacked_struct ps1; // No error
```

Include File Optimization

The Microtec C++ compiler tracks when files are included in a compilation. When including a file for the first time, the compiler records if the file begins with one of the following directives, ignoring comments and whitespace, and ends with a matching **#endif**:

```
#ifdef A_SYMBOL
```

or

```
#ifndef A_SYMBOL
```

Compilation continues, and if the same file is included again, the compiler checks to see if *A_SYMBOL* is defined (in the first case) or undefined (in the second). In either case, the contents of the included file are not processed any further. This mechanism can be used in header files that can be included multiple times during a single compilation. *A_SYMBOL* can be any symbol; it does not need to match the name of the include file. Make sure that *A_SYMBOL* is not defined in any other include file.

In the unlikely case that your header file has preprocessor directives following the **#endif**, as in this example:

```
#ifndef A_SYMBOL
#define A_SYMBOL
... code to be included only once ...
#endif
#warn about something
```

then this mechanism fails to consider the trailing preprocessor directives on the second and later inclusions. To work around this limitation, change the first line to:

```
#if !defined(A_SYMBOL)
```

asm Support

Embedded applications require close and efficient control of the target architecture. To accommodate this need, the Microtec C++ compiler supports assembler statement intermixing with C++ code.

You can include any number of assembler lines by using the pseudofunction **asm**, which can also be specified as **ASM**.

Syntax:

```
asm([type[,]] [string [,string]...]);
```

Description:

<i>type</i>	Tells the compiler what is returned by the asm invocation and, implicitly, where it is returned. If omitted, the value returned by the asm pseudofunction is of type int .
<i>string</i>	Represents assembler instructions. Generally, each <i>string</i> , separated by a comma, corresponds to a new assembler line. You can insert as many assembler instructions as you like using a single asm statement as long as you follow these rules: <ul style="list-style-type: none">• Since each assembler statement must be on its own line, either use a separate string for each assembler statement or use <code>\n</code> to generate a newline character.• Prefix any assembler mnemonic with at least one white space character (tab or space).

The **asm** pseudofunction behaves like a function in that it takes parameters and returns a value of the specified type. However, **asm** does not generate a procedure call. Instead, it inserts assembler code in-line.

The value of the **asm** pseudofunction is the result placed in the return value register. This value varies for different types and different machines, and it can change depending on the options used. The simplest way to identify the return register is to write a small function that returns a value of the desired type. Compile this function using the **-S -Fsm** options and take note of the register(s) used to return the value. In the following examples, the return register used is **D0**, which is used to return integers or addresses in 68K-based systems.

As with any C++ function, **asm** can be invoked with or without using its returned value. The global optimizer in the compiler disables certain optimizations when **asm** is used.

Follow these guidelines when you add **asm** pseudofunctions to your code:

1. Write your C++ program.
2. Compile the program.
3. Look at the places in the assembler output where you are considering adding **asm** pseudofunctions.
4. Within procedures, add assembly code (also known as “inserts”) in **asm()** pseudofunctions.
5. Outside procedures, use **#pragma asm** and **#pragma endasm** to add assembly code. For a description of these directives, refer to section *#pragma asm Versus asm*, in this chapter.

asm Examples

The following are some simple examples of how to use **asm** to add assembler lines to a C++ program.

Example:

```
main()
{
    asm(" move.l $1000,SP ; initialize stack pointer");
    ...
}
```

In the example above, the compiler inserts the quoted string immediately after the prologue code that it generates for `main()`. A newline is added after the string to avoid concatenation with the next line.

Since no type was specified in the above **asm** statement, the return type is **int**. The returned value is not assigned to any variable and is ignored.

Example:

```
asm( " NOP", " TRAP #0");
asm( " NOP\n TRAP #0" );
asm( " NOP ", " T" "RAP #0 " );
/* no ', 'after T - same as " TRAP #0 " */
asm( " NOP "); asm( " TRAP #0 " );
asm(int, " NOP", " TRAP #0");
```

These statements generate the same code:

```
NOP
TRAP #0
```

The compiler supplies a newline character at the end of every string argument. Thus, in the statement:

```
asm( " NOP\n", " TRAP #0");
```

the `\n` is unnecessary.

Examples:

```
asm("label:"); /*label is in col 1 */
asm("MOVE.L #1,4(A6,D2.W); /* ERROR - MOVE in col 1 */
asm(" ORG.S $100");
asm("thous DC.L $1000,$2000,$3000 " );
asm(" garbage in ...");
```

These five examples show how powerful (and potentially dangerous) the **asm** feature can be. Since the compiler passes the string arguments of **asm** directly to the assembler, it does not check the syntax of the assembler statements. Be careful when you use the **asm** pseudofunction.

Assigning asm to a Variable

You can assign a return value of **asm** to a variable, as with any other C or C++ function. The following example determines the value of the stack pointer.

Example:

```
sp_reg=asm(" move.l SP,D0 ; get value of SP register");
```

The above **asm** statement returns an integer (the default type). According to the calling conventions, integers (such as other scalars) are returned in register `D0`. Since **asm** behaves strictly as a function, the compiler generates code to move the returned value to the target.

The compiler produces this sequence in the assembly file:

```
move.l SP,D0 ; get value of SP register
move.l D0,_sp_reg
```

Returning a Typed Value

You can use the type argument to **asm** to tell the compiler what type is returned by the **asm** invocation and, implicitly, where it is returned.

In the next example, the compiler knows a pointer is returned in **D0**. After pseudo-invocation of **asm**, the compiler generates code to move **D0** to the variable **sp**.

Example:

```
int *sp=asm(int *, " move.l SP,D0");
```

In the next example, floating-point values are returned in **FP0** (assuming a coprocessor). The value of **FP0** is then moved into **d**.

Example:

```
double d=asm(double, " fmove FP1,FP0");
```

No real call to **asm** takes place. The passing of the parameters and the call itself are replaced by the string(s) that you supply. You should store in **D0** (or whatever the calling conventions dictate) the value that interests you.

Using #define for Readability

You can make **asm** statements more readable with the **#define** directive.

Example:

```
#define D4 asm(int, " move.l D4,D0");  
#define D5 asm(int, " move.l D5,D0");  
  
if (D4>D5)  
...
```

This example shows that it is possible to access every single machine device at the C or C++ level. The overhead is confined to a move. This move is not even required for **D0** (and **FP0**).

Variable Names Inside asm

You can insert global and local variables by name inside an **asm** string. At the assembly level, a global variable is represented by prepending an underscore (**_**) to the name. For example, global variable **x** is represented by **_x**. Thus, when accessing a global variable inside an **asm** string, simply add the underscore.

Representation of local variables at the assembler level is more complex. Local variables are represented by frame offsets or by registers that can change from one release of the compiler to the next. Their addresses can even change between two compiles if new code or variables are added to a procedure.

The Microtec C++ compiler lets you insert variable names in any **asm** string by simply quoting the names with a back quote (**`**). Since it is unlikely that this char-

acter will appear in any assembler statement, the back quote has been chosen as the default.

Example:

```
foo() {
    int automobile, garage;
    asm(" move.l `automobile`,`garage`");
    . . .
}
```

The actual addresses of the inserted variables are supplied while conforming to the usual scope rules. According to the memory/register binding in force for the compilation, the above example generates:

```
move.l 12(a6),d4
```

If you want to use a character other than the back quote, use the **-uichar** option to change the insert character to *char*. Use the **-uichar** option inside a **#pragma option** rather than from the invocation line to avoid changing the insert character between compilations.

Inserts are recommended for global variables as well as local variables. Variable names inside a string are not affected by compiler options that modify naming conventions, such as **-upd**, unless they are enclosed in back quotes. However, inserts are option-sensitive.

In the next example, the **-upd** option tells the compiler to prefix global variables with a dot (.). Since the contents of the string in **asm** are not modified, the references to **_global** and **_global2** are not resolved by the assembler.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l _global,_global2");
```

In the next example, **global** and **global2** are enclosed in backquotes. When the string is passed to the assembler, the correct name, prefixed with a dot, is passed to the assembler.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l `global`,`global2`");
```

#pragma asm Versus asm

Since **asm** is a pseudofunction, it can be used only where a normal function can be invoked: inside a procedure. Outside a procedure, you can insert any number of unquoted assembler instructions between **#pragma asm** and **#pragma endasm**. You can use these directives to create your own assembly procedure.

Example:

```
#pragma asm
#if _FPU
    fmove.l    ROUNDING,FPCR ; set rounding mode
#endif
#if _CHAR_SIGNED
    #if (_68020 || _68030 || _68040 || _CPU32)
        extb.l    D1
        move.l    D1,D0
    #else
        ext.w     D1          ; sign extend
        ext.l     D1
        move.l    D1,D0
        #define XXX signed.s
    #endif
#else
    moveq     #0,D0          ; zero extend
    move.b    D1,D0
    #define XXX unsigned.s
#endif
#include XXX
#pragma endasm
```

Note the use of user-defined macros such as `xxx` and the predefined macro `_CHAR_SIGNED`. The full preprocessor functionality is available inside the pair **#pragma asm/#pragma endasm** but you cannot use inserts.

Considerations for Assembler In-lining

When you use **asm**, check the effects of optimizations. Even when some optimizations are disabled, the Microtec C++ compiler can still rearrange, change, and delete code. This behavior can affect your program.

The content of any string is passed to the assembler as is. No control is placed on the string. Direct consequences of this are:

- You must be ready to read errors issued by the assembler.

- The compiler makes pessimistic assumptions on the size of the inserted instructions. Every jump crossing the instructions might be long. Consider the following example:

```
asm( " DS $1000" )
```

- **asm** is not portable. However, any program using direct machine instructions is probably not meant to be portable.

Example:

```
printf( "%x",asm() );
```

This statement prints the contents of register **D0**. The data printed could be completely different if the program is compiled on another compiler.

- By default, the compiler uses the **noabspcadd** assembler command line flag. However, when PC-relative addressing is specified (**-Mdp**, **-Mcp**), the compiler uses **abspcadd**, which means that an absolute expression in conjunction with the mnemonic **PC** [for example, **5(PC)**] refers to an absolute address accessed through PC-relative mode rather than to a relative displacement to the current program counter. **asm** strings that use this addressing mode can be assembled differently, depending on the compiler options specified.

For more information about the **abspcadd** assembler command line flag, refer to the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.

Using Libraries 5

A library is a set of objects, derived from the library sources generated by a compilation. The compilation is set by the parameters of a configuration. The Microtec C/C++ compilers are distributed with a set of libraries that provide a number of general configurations for embedded systems applications.

The compiler driver automatically selects the correct library to pass to the linker if you are using the default linker command file. If you are using a different linker command file or if you invoke the linker directly, you must pass the correct library to the linker yourself.

C++ I/O Library

Figure 5-1 shows the architecture of C++ I/O functions. The C++ architecture is composed of three I/O levels:

- Level 1+ contains I/O elements specific to C++. Level 1+ functions can call C++ I/O level 1 functions or C++ I/O level 2 functions.
- Level 1 contains the common higher level I/O functions between C++ and C.
- Level 2 contains the common system-level I/O functions between C++ and C.

Note

The **iostream.h** header file library is only for use with multiple-inheritance libraries. For single inheritance, use the header file **ostream.h**.

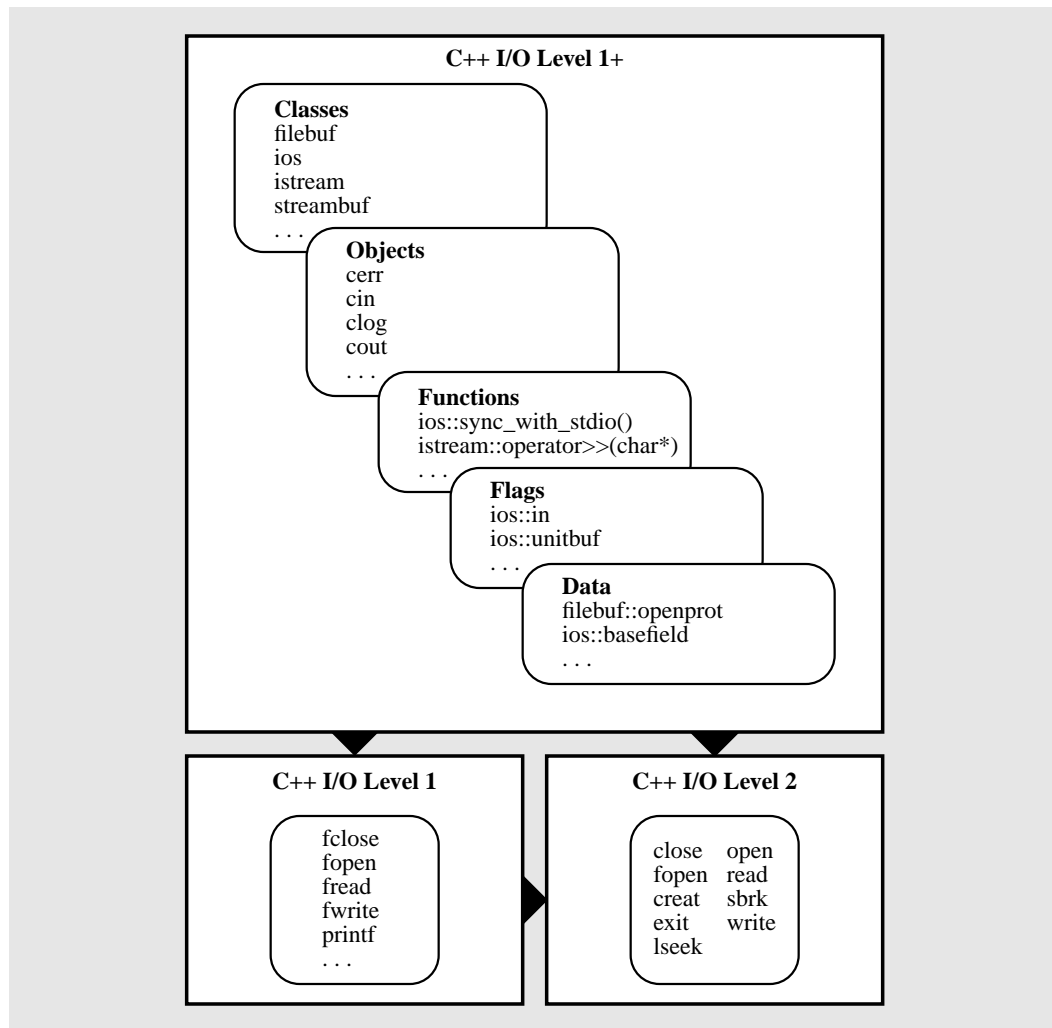


Figure 5-1. C++ I/O Levels

UNIX Level 1+ Elements

UNIX level 1+ contains the following I/O elements specific to C++:

- I/O classes, such as **filebuf**, **ios**, **istream**, and **streambuf**
- I/O objects, such as **cerr**, **cin**, **clog**, and **cout**
- I/O functions, such as **ios::sync_with_stdio()** and **istream::operator>>(char*)**
- I/O flags, such as **ios::in** and **ios::unitbuf**
- I/O data, such as **filebuf::openprot** and **ios::basefield**

These elements are the C++ versions of the level 1 and level 2 objects and functions. For example, for C++ I/O objects, **cout** is the C++ **stdout**. Similarly, **cin**, **cerr**, and **clog** are the C++ **stdin**, **stderr**, and log file, respectively.

I/O Buffering

C++ provides several different methods for I/O buffering:

- Buffered
- Unbuffered
- Unit buffered
- Synchronization with UNIX **stdio** buffering (available only on UNIX systems)

Each of these methods deals with input and output in a different fashion.

Unit buffering is the default. You can specify the I/O buffering behavior that you desire by issuing the proper **ios(3C++)** calls as documented in the *AT&T C++ Library Manual Pages*.

The following sections discuss the buffering methods in more detail. To demonstrate the differences, consider the code fragment:

```
int num = 10;
cout << "abc" << num;
```

This code prints out `abc10`. However, the process differs depending on which I/O buffering method you choose.

Buffered I/O

The buffered I/O method retains all I/O in a buffer until the buffer is flushed. An I/O buffer is flushed when:

- **cout** or **cin** internal buffers overflow
- The program terminates
- You explicitly flush them in your code (for example, using **cout.flush()**)

In the previous example, you do not see `abc10` displayed until the I/O buffer is flushed.

Unbuffered I/O

When I/O is unbuffered, it is not kept in a buffer but is taken one character at a time as it is received. From the previous example, `abc10` is output one character at a time; that is, `a`, `b`, `c`, `1`, and `0`. Unbuffered I/O typically incurs a run-time performance penalty since bytes are flushed one at a time.

Unit Buffered I/O

Unit buffering flushes I/O one unit at a time. From the previous example, `abc` would be considered as one unit, and `num` would be considered as another unit. Therefore, the previous example would first output `abc`, followed by `10`.

In the C++ run-time start-up routine provided, I/O buffering is set to unit buffering by default.

C++ Function Groups and Include Files

Each library function is declared in one of the include files provided with the compiler. Include files define types, constants, and external functions. The files are incorporated using the **#include** directive and angle brackets (for example, **#include <mriehs.h>**). The compiler provides these standard include files for exception handling:

- `mriehs.h`
- `new.h`

mriehs.h File

The **mriehs.h** file must be included if:

- The application uses the **terminate** or **unexpected** functions.
- You set a custom function to handle unexpected exceptions.
- You set a custom termination function.

Table 5-1 lists the functions in **mrriehs.h**.

Table 5-1. Exception Handling Functions in mrriehs.h Include File

Function	Definition
set_terminate	Sets the termination function that is called by the terminate function.
set_unexpected	Sets the function that is called by the unexpected function to handle unexpected exceptions.
terminate	Calls the termination function set by the set_terminate function, or, if no function was set, calls abort .
unexpected	Calls the function set by the set_unexpected function, or, if no function was set, calls terminate .

The exception handling run-time system sets the **errno** value to reflect the error states. Table 5-2 lists the symbols for the **errno** values and the conditions considered as errors by the exception handling run-time system.

Table 5-2. Errors During Exception Handling

Symbol	Set Condition
EHS_EMPTY_THROW	Set if throw with no arguments is called outside a handler.
EHS_FUNC_EXCEP_SPEC	Set if an exception not specified in the exception specifications of the call chain is raised. The unexpected function is called.
EHS_MEMORY_EXCEPTION	Set if the memory exception occurs or the malloc function fails during exception handling processing.
EHS_NO_VALID_CATCH	Set if no valid handler is located for the exception being processed.
EHS_THROW_IN_EHS	Set if an object is thrown during exception handling; for example, a throw from a destructor when processing an exception.
EHS_USER_TERM_RET	Set if a custom function for handling termination returns to its caller.
EHS_USER_UNEXP_RET	Set if a custom function for handling unexpected exceptions returns to its caller.

new.h File

The **new.h** file must be included if:

- The application overrides the default memory exception handler for the **operator new** function. You can call the **set_new_handler** function to set the memory exception handler of the **operator new** to your own handler.
- The application overrides the global **operator new** function.

Table 5-3 lists the functions in **new.h**.

Table 5-3. Functions Associated With operator new

Function	Definition
operator new	Allocates storage space for the given object using the malloc system function.
set_new_handler	Sets the memory exception handler of the operator new function to your custom handler.

Library Extensions

The Microtec C/C++ compilers provide run-time libraries that contain many functions familiar to C programmers. Some low-level system functions are also included for use on target systems; however, these functions may not be applicable to embedded systems.

The last section of this chapter lists all of the C/C++ extensions alphabetically.

C Function Groups and Include Files

In addition to the standard ANSI include files, the Microtec C/C++ compilers also provide extensions beyond the standard ANSI C function set.

mriext.h

The include file **mriext.h** contains Microtec Compiler extensions and macros. A list of the functions is provided in Table 5-4, and a list of the macros is in Table 5-5.

Table 5-4. `mriext.h` Functions

Function	Definition
<code>eprintf</code>	Prints to standard error.
<code>fileno</code>	Gets file number.
<code>ftoa</code>	Converts floating-point numbers to ASCII.
<code>getl</code>	Reads long int .
<code>getw</code>	Reads short int .
<code>isascii</code>	Checks if argument is ASCII.
<code>itoa</code>	Converts integer to ASCII.
<code>itostr</code>	Converts int to ASCII, variable base.
<code>ltoa</code>	Converts long to ASCII.
<code>ltostr</code>	Converts long to ASCII, variable base.
<code>max</code>	Returns maximum value.
<code>memccpy</code>	Copies memory looking for a character.
<code>memclr</code>	Clears memory bytes.
<code>min</code>	Returns minimum value.
<code>putl</code>	Writes long int .
<code>putw</code>	Writes short int .
<code>swab</code>	Swaps bytes in memory.
<code>toascii</code>	Guarantees argument is ASCII.
<code>_tolower (c)^a</code>	Is the same as tolower , except does not perform argument checking.
<code>_toupper (c)^a</code>	Is the same as toupper , except does not perform argument checking.
<code>zalloc</code>	Allocates data space dynamically and initializes to zero.

a. The parameter *c* refers to a character string.

Table 5-5. mriext.h Macros

Macro	Definition
BLKSIZE	Size of I/O buffer
FALSE	Value of 0
NULLPTR	Value of NULL pointer
stdaux	Pointer to a standard auxiliary output device
stdprn	Pointer to a standard printer device
TRUE	Value of 1

Non-Reentrant Extensions

Most Microtec Compiler library extensions are reentrant. However, there are library extensions that are non-reentrant due to the use of static, nonautomatic data. Many standard ANSI C Library functions are also non-reentrant.

Table 5-6 shows the Microtec Compiler non-reentrant extensions.

Table 5-6. Non-Reentrant Extensions

Function	Static Variables Used
alloca ^b	_iob ^a
eprintf	_iob ^a
getl	_iob ^a
getw	_iob ^a
putl	_iob ^a
putw	_iob ^a
zalloc	_membase, _avail, _badlist

a. This extension may have an I/O buffer attached through **_iob**. An I/O buffer is a buffer used for buffered I/O. Any I/O stream that is unbuffered does not access the buffer.

b. C++ extension only.

I/O System Functions

Table 5-7 lists the system functions that perform input/output operations. The functions listed in this table are currently implemented to use XRAY's SysHost mechanism and are non-reentrant.

For more information on system functions, see Chapter 13, *Embedded Environments*.

Table 5-7. System Functions

Function	Definition
chdir	Changes working directory.
close	Closes a specified file.
connect	Connects a socket to a server.
creat	Creates a new file.
_exit	Terminates a program.
getcwd	Gets working directory.
lseek	Sets the current location in a file.
open	Opens a specified file.
pclose	Closes a pipe.
popen	Opens a pipe.
read	Reads from a specified file.
rename	Renames a file.
sbrk	Allocates memory space.
socket	Opens a socket.
stat	Gets information on a file.
sys_in	Gets input from original stdin.
sys_out	Puts output to original stdout.
sys_stat	Checks keyboard status.
system	Executes host system command.

(cont.)

Table 5-7. System Functions (cont.)

Function	Definition
time	Gets system time.
tmpnam	Generates temporary filename.
unlink	Unlinks a filename.
write	Writes to a specified file.

Intrinsic System Functions

Library functions include a set of intrinsic system functions. These system functions support **operator new**, **operator delete**, and the virtual function scheme. These functions are listed in Table 5-8.

Table 5-8. Intrinsic System Functions

Function	Definition
_cxxfini	Calls static destructors to clean up static objects before termination of the program.
_ehs_cleanup_area	Cleans up a user-allocated memory area before it is de-allocated.
_ehs_init_area	Initializes a user-allocated memory area for use by a new thread.
_ehs_restore_from_area	Restores the exception handling context from a memory buffer.
_ehs_save_size	Returns the size of the buffer required to save the exception handling state.
_ehs_save_restore_area	Combination of _ehs_save_to_area and _ehs_restore_from_area for a faster context switch.
_ehs_save_to_area	Stores the exception handling state in a memory buffer.

(cont.)

Table 5-8. Intrinsic System Functions (cont.)

Function	Definition
<code>_main</code>	Calls static constructors to initialize static objects before the first statement of the main function.
<code>_pure_virtual_function_called</code>	Terminates the program if a pure virtual function of an abstract class is called.
<code>set_new_handler</code>	Sets the exception handler for the operator new function.

Library Function Definitions

This section describes all of the library functions in alphabetical order and presents them in a fixed format for easy reference. The format used is described below. Unless specifically noted, the extensions are available in both the C and the C++ compilers.

Function Name

Each section begins with the name of the library function followed by a short, one-line description.

Syntax

The syntax shows the return type of the function and the types of its formal arguments. Declarations for some of these functions are in the standard include file indicated. Alternatively, if a function is not implemented as a macro, it can be declared as an external function with the appropriate return type.

Where the syntax differs between the C and C++ compilers, the code examples appear under separate headers.

Class

Each function is flagged as a Microtec C Compiler extension. Some functions are implemented as a macro or a system function. Functions not implemented as macros can be declared as external functions with the appropriate return type.

Description

The description gives the meaning of the function and its return values.

Notes

This section contains information on abnormal behavior, embedded environment considerations, and other issues that may affect the behavior of the function or macro.

alloca — Allocates Data Space on Stack

This extension is only available for the C++ compiler.

Syntax

```
#include <alloca.h>
void * alloca(size_t size);
```

Class

Microtec C++ Compiler Extension

Description

The `alloca` function allocates *size* bytes of space in the stack frame of the caller, and returns a pointer to the allocated space. This temporary space is automatically freed when the caller returns.

If the requested amount of space cannot be allocated, `alloca` returns a null pointer. The new stack pointer is checked against `__STACK_BASE` to make sure that enough stack space is available for the allocation.

Notes

`alloca` is defined as `_alloca` in the header file `alloca.h`. This is to avoid any conflict with previously defined `alloca` functions used for other purposes.

chdir — Changes Working Directory

Syntax

```
#include <direct.h>
int chdir(const char *dir);
```

Class

System (SysHost) Function

Description

Changes current working directory pathname to name specified by *dir*.

Returns 0 when successful or -1 if an error is detected and sets **errno** to indicate the error.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

close — Closes a Specified File

C Syntax

```
int close (int fd);
```

C++ Syntax

```
extern "C" {  
    int close (int fd);  
}
```

Class

System Function

Description

The `close` function closes a file previously opened by either the **open** or **creat** function. The file descriptor *fd* is returned by **open** or **creat** when the associated file is opened. A file descriptor must be specified (as opposed to a stream address).

The **close** function returns 0 when successful or -1 if the function detects an unknown file descriptor.

Notes

For more information on system functions, see Chapter 13, *Embedded Environments*.

connect — Initiates a Socket Connection on the Host System

Syntax

```
#include <socket.h>
int connect (int sock, struct sockaddr *name, int length)
```

Class

System (SysHost) Function

Description

Passes the arguments to the local host's `connect` or `_connect` system call. *sock* is a socket descriptor returned from a socket call. *name* is an implementation-dependent socket name, and *length* indicates the length of this name.

Returns 0 when successful or -1 if an error is detected and sets **errno** to indicate the error.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

creat — Creates a Specified File

C Syntax

```
int creat (char *filename, int mode);
```

C++ Syntax

```
extern "C" {  
    int creat (char *filename, int mode);  
}
```

Class

System Function

Description

The `creat` function creates and opens the file *filename* for writing. *mode* specifies the file's protection mode. For devices that cannot be created (such as terminals), the `creat` function simply opens the device.

If `creat` is successful, it returns a file descriptor for *filename*. It returns -1 if the file cannot be written to, if the file is a directory, or if there are already too many files opened.

Notes

For more information on system functions, see Chapter 13, *Embedded Environments*.

`_cxxfini` — Calls Static Destructors to Destroy Static Objects

Syntax

```
extern "C" {  
    void _cxxfini();  
}
```

Class

Intrinsic System Function

Description

The `_cxxfini` function calls the static destructors of static objects before the program exits. A static destructor is a compiler-generated destructor used to destroy static objects for each module. All static destructors must be executed before the program exits. More specifically, static destructors must be executed before the file cleanup operations, such as flushing file buffers and closing file channels.

The **`exit`** function calls the `_cxxfini` function.

Notes

The `_cxxfini` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_main`

`_ehs_cleanup_area` — Cleans up the Exception Handling State Buffer

Syntax

```
#include <mriehs.h>
extern "C" {
    void _ehs_cleanup_area(void *area_ptr);
}
```

Class

Intrinsic System Function

Description

The `_ehs_cleanup_area` function de-allocates any dynamically allocated memory associated with the exception handling state buffer. This function must be called before the memory referred to by `area_ptr` is de-allocated. Failure to do so can cause memory leak.

Notes

The `_ehs_cleanup_area` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_init_area`

`_ehs_init_area` — Initializes the Exception Handling State Buffer

Syntax

```
#include <mriehs.h>
extern "C" {
    void _ehs_init_area(void *area_ptr);
}
```

Class

Intrinsic System Function

Description

The `_ehs_init_area` function initializes the user-allocated memory area pointed to by *area_ptr* for use by a new thread. Once this call has been made, the memory buffer can be used by the `_ehs_save_to_area` and the `_ehs_restore_from_area` calls.

Notes

The `_ehs_init_area` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_restore_from_area`
`_ehs_save_size`
`_ehs_save_to_area`
`_ehs_cleanup_area`

`_ehs_restore_from_area` — Restores the Exception Handling State

Syntax

```
#include <mriehs.h>
extern "C" {
    void _ehs_restore_from_area(void *area_ptr);
}
```

Class

Intrinsic System Function

Description

The `_ehs_restore_from_area` function restores the exception handling context from the memory buffer pointed to by `area_ptr`. The buffer should have been previously initialized by the `_ehs_init_area` function call. An exception handling state should have been saved to the buffer by the `_ehs_save_to_area` function call. The current exception handling state is destroyed and then overwritten with the restored state.

Notes

The `_ehs_restore_from_area` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_init_area`
`_ehs_save_size`
`_ehs_save_to_area`

`_ehs_save_restore_area` — Saves and Restores the Exception Handling State

Syntax

```
#include <mriehs.h>
extern "C" {
    void _ehs_save_restore_area(void *to_area, void
    *from_area);
}
```

Class

Intrinsic System Function

Description

The `_ehs_save_restore_area` function is a combination of `_ehs_save_to_area` (`to_area`) followed by `_ehs_restore_from_area` (`from_area`). This combination is faster than calling the two options separately, and is useful for reducing the context switch time of applications using C++ exception handling.

Notes

The `_ehs_save_restore_area` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_save_to_area`
`_ehs_restore_from_area`

`_ehs_save_size` — Returns Memory Size Required to Save Exception Handling State

Syntax

```
#include <mriehs.h>
extern "C" {
    unsigned int _ehs_save_size(void);
}
```

Class

Intrinsic System Function

Description

The `_ehs_save_size` function returns the size of the buffer required to save the exception handling state. This size remains constant for the entire program execution and across all threads. The returned value should be used to allocate buffers for saving the exception handling state of each thread.

Notes

The `_ehs_save_size` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_init_area`
`_ehs_restore_from_area`
`_ehs_save_to_area`

`_ehs_save_to_area` — Saves the Exception Handling State

Syntax

```
#include <mriehs.h>
extern "C" {
    void _ehs_save_to_area(void *area_ptr);
}
```

Class

Intrinsic System Function

Description

The `_ehs_save_to_area` function saves the exception handling state in the area pointed to by *area_ptr*. The memory area must be of the required size. The required size can be obtained by calling the `_ehs_save_size` function.

Notes

The `_ehs_save_to_area` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_ehs_init_area`
`_ehs_restore_from_area`
`_ehs_save_size`

eprintf — Provides Formatted Print to Standard Error

C Syntax

```
#include <mriext.h>
int eprintf (const char *format, ...);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
int eprintf (const char *format, ...);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `eprintf` function provides formatted output to the C standard error file. The format and results are the same as for **`printf`**. When `eprintf` is successful, it returns the number of characters transmitted.

If an output error is encountered, `eprintf` returns **EOF (End-Of-File)**.

Notes

The function `eprintf` modifies the static array **`_iob`**; no other static variables are modified by `eprintf`. However, the system function **`write`**, which can be called via `eprintf`, can modify **`errno`** and possibly others. For more information on system functions, see Chapter 13, *Embedded Environments*.

`_exit` — Terminates a Program

C Syntax

```
void _exit (int status);
```

C++ Syntax

```
extern "C" {  
    void _exit (int status);  
}
```

Class

System Function

Description

The `_exit` function causes normal program termination to occur with a status code specified by *status*.

The `_exit` function is always called implicitly at the end of an **`exit()`** function call.

Notes

A call to `_exit` never returns.

For more information on system functions, see Chapter 13, *Embedded Environments*.

fileno — Gets File Descriptor

C/C++ Syntax

```
#include <mriext.h>
char fileno (FILE * stream);
```

Class

Microtec C/C++ Compiler Extension (implemented as a macro)

Description

The `fileno` macro returns the file descriptor associated with *stream*.

Notes

The macro yields undefined results if *stream* does not point to an open file.

ftoa — Converts Floating-Point Number to ASCII String

C Syntax

```
#include <mriext.h>
int ftoa (double value, char *str, int format, int prec);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int ftoa (double value, char *str, int format, int prec);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `ftoa` function converts the floating-point number *value* to a **NULL**-terminated ASCII string with a specified format and precision. The output is placed in the area pointed to by *str*. No leading blanks or zeros are produced.

The conversion format character *format* must be **f**, **e**, **E**, **g**, or **G**. The conversion format character and the precision *prec* are the same as those defined for the **printf** function. (The parameter *prec* is the same as the *digits* parameter in **printf**.) In the C compiler, even though *format* is declared as an `int`, it can be specified as a character.

The **printf** function calls `ftoa` to convert floating-point numbers. The `ftoa` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

If an invalid floating-point number is passed to the `ftoa` function, the behavior is unpredictable.

getcwd — Returns Current Working Directory

Syntax

```
#include <direct.h>
char *getcwd(char *buffer, int len)
```

Class

System (SysHost) Function

Description

Copies up to *len* characters of current working directory pathname into returned buffer.

Returns buffer when successful or 0 if an error is detected and sets **errno** to indicate the error.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

getl — Reads a Long Integer From a Stream

C Syntax

```
#include <mriext.h>
long getl (FILE * stream);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    long getl (FILE * stream);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The function `getl` reads a long integer from *stream* and returns it. `getl` returns **EOF** at the end of *stream*. Any long integer read from *stream* must have been written with a **putl** function.

Notes

`getl` yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array **_iob**. Static variables are also modified by `getl`, however, the system function **read**, which can be called via `getl`, can modify **errno** and possibly others. For more information on system functions, see Chapter 13, *Embedded Environments*.

getw — Reads a Short Integer From a Stream

C Syntax

```
#include <mriext.h>
int getw (FILE *stream);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int getw (FILE *stream);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The function `getw` reads a **short int** from *stream* and returns it as an `int`.

In the C compiler, `getw` returns **EOF** if the **End-Of-File** is read. In the C++ compiler, `getw` returns **EOF** at the end of *stream*.

Any short integer read from *stream* must have been written with a **putw** function.

Notes

The function `getw` yields unpredictable results if *stream* does not point to an open file.

The function `getw` can modify the static array `_iob`. Static variables are also modified by `getw`, however, the system function **read**, which can get called via `getw`, can modify **errno** and possibly others. For more information on system functions, see Chapter 13, *Embedded Environments*.

isascii — Tests for an ASCII Character

C Syntax

```
#include <mriext.h>
int isascii (int c);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int isascii (int c);
}
```

Class

Microtec C/C++ Compiler Extension (implemented as a function and as a macro)

Description

The `isascii` function or macro returns a nonzero number if the character in `c` is an ASCII character. It returns 0 otherwise. ASCII characters range in value from 0 to 127.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses as follows:

```
(isascii)(c);
```

- Undefine the `isascii` macro with **#undef** before using it, as follows:

```
#undef isascii
```

itoa — Converts Integer to ASCII String

C Syntax

```
#include <mriext.h>
int itoa (int i, char *cp);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int itoa (int i, char *cp);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `itoa` function converts the signed integer *i* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The `itoa` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

itostr — Converts Unsigned Integer to ASCII String

C Syntax

```
#include <mriext.h>
int itostr (unsigned u, char *cp, int base);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int itostr (unsigned u, char *cp, int base);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `itostr` function converts the unsigned integer *u* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The `itostr` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

lseek — Sets the Current Location in a File

C Syntax

```
long lseek (int fd, long offset, int whence);
```

C++ Syntax

```
extern "C" {  
    long lseek (int fd, long offset, int whence);  
}
```

Class

System Function

Description

Updates the current position in the file to the value given in *offset*. The *whence* argument indicates how to interpret the offset:

<i>whence</i> = 0	<i>offset</i> is relative to the beginning of the file.
<i>whence</i> = 1	<i>offset</i> is relative to the current position.
<i>whence</i> = 2	<i>offset</i> is relative to the end of the file.

The `lseek` function returns the resulting file position, measured in bytes from the beginning of the file. If a failure occurs, -1 is returned.

Notes

The function `lseek` can modify the static array `_iob`; no other static data can be modified.

The function `lseek` has been implemented to be used with XRAY's SysHost feature. If `lseek` is unsuccessful in this environment, a -1 is returned and `errno` is set to indicate the error.

If used outside this environment, `lseek` writes the message:

```
WARNING - lseek (stub routine) called
```

to `stderr` and returns 0. You may need to provide your own `lseek` function for your environment. For more information on system functions, see Chapter 13, *Embedded Environments*.

ltoa — Converts Long Integer to ASCII String

C Syntax

```
#include <mriext.h>
int ltoa (long value, char *cp);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int ltoa (long value, char *cp);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `ltoa` function converts the long signed integer *value* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The `ltoa` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

ltostr — Converts Unsigned Long Integer to ASCII String

C Syntax

```
#include <mriext.h>
int ltostr (unsigned long ul, char *cp, int base);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int ltostr (unsigned long ul, char *cp, int base);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `ltostr` function converts the unsigned long integer *ul* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The `ltostr` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

_main — Calls Static Constructors to Create Static Objects

Syntax

```
extern "C" {  
    void _main();  
}
```

Class

Intrinsic System Function

Description

The `_main` function constructs static objects before the first statement of the **main** function is executed. A static constructor is a compiler-generated constructor to initialize static objects on a per-module basis.

The **main** function calls the `_main` function.

Notes

The `_main` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`_cxxfini`

max — Returns the Greater of Two Values

C/C++ Syntax

```
#include <mriext.h>
max (a, b);
```

Class

Microtec C/C++ Compiler Extension (implemented as a macro)

Description

The macro `max` returns the larger of *a* and *b*. *a* and *b* can be any valid C expression yielding an integral, floating-point, or pointer type. `max` performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

memcpy — Copies Characters in Memory

C Syntax

```
#include <mriext.h>
char *memcpy (char *s1, const char *s2, int c, size_t i);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    char *memcpy (char *s1, const char *s2, int c, size_t i);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `memcpy` function copies characters from memory area `s2` into `s1`, stopping after the first occurrence of character `c` has been copied or after `i` characters have been copied, whichever comes first. This function returns a pointer to the character after the copy of `c` in `s1`.

The `memcpy` function returns a null pointer if `c` was not found in the first `i` characters of `s2`.

Notes

If ANSI features are disabled, the parameter `i` is type **unsigned int** instead of `size_t`. See Chapter 3, *Using Command Line Options*, for information on the compiler command line option that disables ANSI features.

memset — Clears Memory Bytes

C Syntax

```
#include <memset.h>
char *memset (char *cp, size_t i);
```

C++ Syntax

```
#include <memset.h>
extern "C" {
    char *memset (char *cp, size_t i);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The `memset` function sets the first *i* bytes in the memory area *cp* to zero. The `memset` function returns *cp*.

Notes

If ANSI features are disabled, the parameter *i* is type **unsigned int** instead of `size_t`. See the **-A** option in Chapter 3, *Using Command Line Options*, for information on the compiler command line option that disables ANSI features.

min — Returns the Lesser of Two Values

C Syntax

```
#include <mriext.h>
min (a, b);
```

C++ Syntax

```
#include <mriext.h>
min (a, b);
```

Class

Microtec C/C++ Compiler Extension (implemented as a macro)

Description

The macro `min` returns the lesser of *a* and *b*. *a* and *b* can be any valid C expression. *a* and *b* can be any integral, floating-point, or pointer type. `min` performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

open — Opens a Specified File

C Syntax

```
int open (char *filename, int mode);
```

C++ Syntax

```
extern "C" {
    int open (char *filename, int mode);
}
```

Class

System Function

Description

The `open` function opens an existing file for reading, writing, or updating. The **NULL**-terminated string *filename* must correspond to a valid filename on the target operating system. `open` returns a non-negative file descriptor when successful, and a -1 when unsuccessful.

The function opens the file named by *filename* in the mode indicated by *mode*. The values for *mode* and the corresponding symbols are shown in Table 5-9.

Table 5-9. Mode Values for the open Function

Mode Value	Symbol	Description
0x0000	O_RDONLY	Opens for read only.
0x0001	O_WRONLY	Opens for write only.
0x0002	O_RDWR	Opens for read and write.
0x0008	O_APPEND	Performs writes at EOF .
0x0200	O_CREAT	Creates a file.
0x0400	O_TRUNC	Truncates the file.
0x4000	O_FORM	Is a text file.
0x8000	O_BINARY	Is a binary file.

Notes

The function `open` has been implemented to be used with XRAY's SysHost feature. If `open` is unsuccessful in this environment, a -1 is returned and **errno** is set to indicate the error.

If used outside this environment, **open** writes the message:

```
WARNING - open (stub routine) called
```

to **stderr** and returns 0. You may need to provide your own `open` function for your environment. For more information see Chapter 13, *Embedded Environments*.

_pclose — Closes a Pipe from a Process

Syntax

```
#include <stdio.h>
int _pclose(FILE *pipe)

    or

int _pclose(FILE *pipe)
```

Class

System (SysHost) Function

Description

If the *pipe* is write enabled, the buffer is flushed. The pipe is closed and the file structure is reset. If a system buffer was allocated from the heap, it is freed.

Returns 0 when successful or **EOF** if the pipe is not opened with **_popen()**.

Notes

Only **_pclose** can be used with the **-nx** option.

This function has been implemented to work with the XRAY SysHost feature.

For more information, see Chapter 13, *Embedded Environments*.

See Also

_popen

_popen — Opens a Pipe to a Process

Syntax

```
#include <stdio.h>
FILE *_popen(const char *cmd, const char *mode)

    or

FILE *popen(const char *cmd, const char *mode)
```

Class

System (SysHost) Function

Description

Returns a pointer to the stream structure when successful, otherwise a **NULL** pointer is returned.

cmd is a pointer to a command to be executed on the host system. *mode* indicates whether the pipe is to be written to, or read from. Only **r** and **w** are allowed. The SysHost implementation of `popen` creates a pipe between the application and the host system. If the opentype is **w**, then standard input to the command is provided by writing to the returned stream, if the opentype is **r**, then standard output from the command is fetched by reading from the returned stream.

Notes

Only `_popen` can be used with the **-nx** option.

This function has been implemented to work with the XRAY SysHost feature.

For more information, see Chapter 13, *Embedded Environments*.

See Also

`_pclose`

__pure_virtual_function_called — Handles Abnormal Conditions With Pure Virtual Functions

Syntax

```
extern "C" {
    void abort();
    void __pure_virtual_function_called();
}
```

Class

Intrinsic System Function

Description

The `__pure_virtual_function_called` function calls the **abort** function to terminate execution. It is invoked by the C++ run-time support of virtual functions if the program calls a pure virtual function in an abstract class.

Notes

A pure virtual function can be called in a constructor.

`__pure_virtual_function_called` follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

Example

```
class Abstract {
public:
    virtual void pure_vf() = 0;
    void f1(int n) {
        if (n>1)
            pure_vf();
    }
    Abstract(int n) {
        f1(n);
    }
};

class B: public Abstract {
public:
    virtual void f2() { }
    B(int n): Abstract(n) { }
    // B(n) is okay as long as n<=1
    // B(2) calls A::pure_vf() indirectly
};
```

If the function `Abstract::pure_vf` is called, the run-time C++ execution environment calls `__pure_virtual_function_called` to abort the execution. If the program is not aborted when a pure virtual function is called, a custom function `__pure_virtual_function_called` or **abort** can be supplied. For example, a custom function might print a warning string and the program location indicating that a pure virtual function was called by the run-time C++ execution environment.

putl — Writes a Long Integer to a Stream

C Syntax

```
#include <mriext.h>
long putl (long l, FILE *stream);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    long putl (long l, FILE *stream);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The function `putl` writes a long integer to *stream*. `putl` returns *l*.

Notes

`putl` yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array `_iob`; no other static data is modified.

putw — Writes a Short Integer to a Stream

C Syntax

```
#include <mriext.h>
int putw (int w, FILE *stream);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int putw (int w, FILE *stream);
}
```

Class

Microtec C/C++ Compiler Extension

Description

The function `putw` writes a short integer to *stream*. `putw` returns *w*.

Notes

`putw` yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array `_iob`; no other static data is modified.

read — Reads Bytes From a Specified File

C Syntax

```
int read (int fd, char *buffer, int nbyte);
```

C++ Syntax

```
extern "C" {  
    int read (int fd, char *buffer, int nbyte);  
}
```

Class

System Function

Description

The `read` function reads *nbyte* bytes from the file associated with *fd* into the buffer pointed to by *buffer*. The file descriptor *fd* is returned by **open** or **creat**.

The `read` function returns the number of bytes actually read or 0 when **End-Of-File** is reached. In the case of an error, the `read` function returns -1.

Notes

For more information on system functions, see Chapter 13, *Embedded Environments*.

sbrk — Allocates Memory Space

C Syntax

```
void *sbrk (int size);
```

C++ Syntax

```
extern "C" {  
    void *sbrk (int size);  
}
```

Class

System Function

Description

The `sbrk` function allocates additional memory from the system. A new block of memory, at least *size* bytes, is allocated, and a pointer to the start of this memory block is returned.

The `sbrk` function returns a pointer to the starting address of the memory block or -1 if the requested amount of memory cannot be allocated.

Notes

For more information on system functions, see Chapter 13, *Embedded Environments*.

set_new_handler — Sets the Memory Exception Handler for operator new

Syntax

```
#include <new.h>
void * set_new_handler(void (* handler)());
void handler(void);
```

Class

Intrinsic System Function

Description

set_new_handler sets the memory exception handler, **_new_handler**, to a custom function, *handler*. It returns a pointer to the previously set **_new_handler** function.

See Also

operator new
operator delete
__EHS__vec_new
__EHS__vec_delete

set_terminate — Sets the Termination Function to Terminate Execution

Syntax

```
#include <mrhiehs.h>
typedef void (*__EHS_PFV)();
extern "C" {
    __EHS_PFV set_terminate(__EHS_PFV user_terminate);
}
```

Class

Standard C++ Function of Exception Handling

Description

The `set_terminate` function directs the default system **terminate** function to call your custom termination function, *user_terminate*. The `set_terminate` function returns the function previously set using the `set_terminate` call.

Your custom termination function must not take arguments or return anything. All termination functions, including the system default termination function, are of type `__EHS_PFV`.

The exception handling run-time library does not expect the termination function to return. If your termination function returns, it leads to undefined behavior and, in some instances, to a very tight infinite loop. If you do not wish to terminate execution when the termination function is called, you should implement a stack-based recovery scheme to rethrow the exceptions that cause program termination. This scheme requires a routine that rethrows the current exception to a catch-all handler in the call chain. You need to ensure this routine is called by the `set_terminate` function for unexpected exceptions.

Notes

The `set_terminate` function obeys the C linkage name rules; specifically, the linkage name does not include function prototype information.

See Also

set_unexpected
terminate
unexpected

set_unexpected — Sets the Function to Handle Unexpected Exceptions

Syntax

```
#include <mriehs.h>
typedef void (*__EHS_PFV)();
extern "C" {
    __EHS_PFV set_unexpected(__EHS_PFV user_unexpected);
}
```

Class

Standard C++ Function of Exception Handling

Description

The `set_unexpected` function directs the system default function **unexpected** to call the custom function *user_unexpected*. The `set_unexpected` function returns a pointer to the previously set unexpected function.

The *user_unexpected* function can take no arguments and must return `void`. All unexpected exception functions, including the system default function, are of type `__EHS_PFV`.

Notes

The `set_unexpected` function uses the `extern "C"` linkage name, which implies that the linkage name is not encoded with function prototype information.

See Also

set_terminate
terminate
unexpected

socket — Creates a Communication Endpoint on the Host System

Syntax

```
#include <socket.h>
int socket(long domain, long type, long protocol)
```

Class

System (SysHost) Function

Description

Pass the arguments to the host `socket()` call.

Returns non-negative descriptor when successful or -1 if an error is detected and sets **errno**.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

stat — Gets Information about File

Syntax

```
#include <stat.h>
int stat(const char *path, struct stat *info);
```

Class

System (SysHost) Function

Description

Returns information about the file specified by *path*. The information is fetched via the host's `stat()` or `_stat()` system call, and moved into the associated fields of **info*.

Returns 0 when successful, or -1 if an error is detected and sets `errno`.

Notes

This feature has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

swab — Swaps Odd and Even Bytes in Memory

C Syntax

```
#include <mriext.h>
void swab (char *source, char *dest, int count);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    void swab (char *source, char *dest, int count);
}
```

Class

Microtec C++ Compiler Extension

Description

The `swab` function moves data from *source* to *dest*, swapping odd and even bytes in the process. The parameter *count* should be even. If *count* is odd, the last byte of *source* is not moved.

Notes

If *source* and *dest* memory spaces overlap, the behavior is undefined.

sys_in — Reads Characters from Standard Input Window

Syntax

```
#include <sysio.h>
int sys_in(char *buffer, int count, SYS_INMODE mode);
```

Class

System (SysHost) Function

Description

Gets characters from the normal standard input and stores them into a buffer. *mode* indicates whether to wait for a line, multiple lines, or just return what has already been input. Depending on *mode*, *count* can indicate the maximum number of characters to read, or the maximum number of lines.

This function is not affected by other operations on the **stdin** file descriptor (for example, it is unaffected by **close(0)**).

Returns the number of characters read or -1 if an error is detected and sets **errno**.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

sys_out — Sends Characters to Standard Output Window

Syntax

```
#include <sysio.h>
int sys_out(const char *buffer, int maxbytes);
```

Class

System (SysHost) Function

Description

Sends characters to standard output. This function is not affected by other operations on the **stdout** file descriptor (1) (for example, **close(1)**).

Returns non-zero if successful, or 0 if an error is detected and sets **errno**.

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

sys_stat — Checks for Input Characters Waiting

Syntax

```
#include <sysio.h>
int sys_stat(void)
```

Class

System (SysHost) Function

Description

Checks if characters are available from standard input. Returns non-zero if characters are waiting, or zero otherwise. This function is not affected by other operations on file descriptor 0 (for example, **close(0)**).

Notes

This function has been implemented to work with the XRAY SysHost feature.

For more information on system and SysHost functions, see Chapter 13, *Embedded Environments*.

terminate — Calls abort() or the Last Function Set

Syntax

```
#include <mrhiehs.h>
extern "C" {
    void terminate();
}
```

Class

Standard C++ Function of Exception Handling

Description

The `terminate` function calls the function specified by the last **set_terminate** function or the **abort** function to terminate execution. The `terminate` function is called when the exception handling run-time system encounters errors.

The default termination function can be changed using the **set_terminate** function.

Notes

The `terminate` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

set_terminate
set_unexpected
unexpected

toascii — Converts a Byte to ASCII Format

C Syntax

```
#include <mriext.h>
int toascii (int c);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int toascii (int c);
}
```

Class

Microtec C/C++ Compiler Extension (implemented as a function and as a macro)

Description

The `toascii` function converts the value of its argument *c* to an ASCII character by truncating all but the lower order 7 bits. The argument *c* is not modified. `toascii` returns a value in the range 0 to 127.

The macro is invoked by default. To call the function, you must do one of the following:

- Enclose the name in parentheses as follows:

```
(toascii)(c);
```

- Undefine the `toascii` macro with **#undef** before using it, as follows:

```
#undef toascii
```


`_tolower` — Converts Characters to Lower-case Without Argument Checking

C Syntax

```
#include <mriext.h>
int _tolower (int c);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int _tolower (int c);
}
```

Class

Microtec C/C++ Compiler Extension (implemented as a function and as a macro)

Description

The `_tolower` function converts the value of its argument `c` to a lower-case letter. The conversion is performed whether or not `c` is an upper-case letter. The argument `c` is not modified.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses as follows:

```
(_tolower)(c);
```

- Undefine the `_tolower` macro with **#undef** before using it, as follows:

```
#undef _tolower
```

Notes

If `c` is not an upper-case letter, the result is undefined.

`_toupper` — Converts Characters to Upper-case Without Argument Checking

C Syntax

```
#include <mriext.h>
int _toupper (int c);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    int _toupper (int c);
}
```

Class

Microtec C/C++ Compiler Extension (implemented as a function and as a macro)

Description

The `_toupper` function converts the value of its argument *c* to an upper-case letter. The conversion is performed whether or not *c* is a lower-case letter. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must do one of the following:

- Enclose the name in parentheses as follows:

```
(_toupper)(c);
```

- Undefine the `_toupper` macro with **#undef** before using it, as follows:

```
#undef _toupper
```

Notes

If *c* is not a lower-case letter, the result is undefined.

unexpected — Calls `terminate()` or the Last Function Set

Syntax

```
#include <mriehs.h>
extern "C" {
    void unexpected();
}
```

Class

Standard C++ Function of Exception Handling

Description

The `unexpected` function calls the function specified by the last **`set_unexpected`** function or the **`terminate`** function. The `unexpected` function is called to handle unexpected exceptions.

Example:

```
void foo() throw (int, char) {
    bar();
}
```

In this example, the `unexpected` function would be called if `bar` raised an exception of any type other than **`int`** or **`char`**.

You can direct the compiler run-time environment to call a custom function to handle the unexpected exceptions using the **`set_unexpected`** function. In that case, the `unexpected` function calls the last function set by the **`set_unexpected`** function.

Notes

The `unexpected` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

`set_unexpected`
`set_terminate`
`terminate`

unlink — Unlinks a Filename

C Syntax

```
int unlink (char *filename);
```

C++ Syntax

```
extern "C" {  
    int unlink (char *filename);  
}
```

Class

System Function

Description

The `unlink` function removes the ability to access the file named by *filename*. Generally, this is accomplished by removing the directory entry for *filename*. If no other links or directory entries for the file exist, then the file is deleted.

Notes

The function `unlink` is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **`_WARNING_unlink_stub_used`** symbol is unresolved. If your program calls the `unlink` stub, it writes the message:

```
WARNING - unlink (stub routine) called
```

to **`stderr`**. You must provide your own `unlink` function for your environment. For more information see Chapter 13, *Embedded Environments*.

The `unlink` function should return -1 if the operation fails.

write — Writes Bytes to a Specified File

C Syntax

```
int write (int fd, char *buffer, int nbyte);
```

C++ Syntax

```
extern "C" {  
    int write (int fd, char *buffer, int nbyte);  
}
```

Class

System Function

Description

The `write` function writes *nbyte* bytes from a buffer to the file associated with the file descriptor *fd*. The starting buffer address is specified by *buffer*. The file descriptor *fd* is returned by **open** or **creat**. Up to 64KB may be written, starting at the current file position.

The `write` function returns the number of bytes actually written or a -1 if an error occurs.

Notes

For more information on system functions, see Chapter 13, *Embedded Environments*.

zalloc — Allocates Data Space Dynamically

C Syntax

```
#include <mriext.h>
void *zalloc (size_t size);
```

C++ Syntax

```
#include <mriext.h>
extern "C" {
    void *zalloc (size_t size);
}
```

Class

Microtec C++ Compiler Extension

Description

The `zalloc` function allocates a new area of memory for program use and returns a pointer to that area. The size of the area allocated is equal to *size* bytes. The `zalloc` function sets the area to zeros. The space allocated is guaranteed to start on a boundary that is suitable for aligning any type.

If the requested amount of space cannot be allocated, `zalloc` returns a null pointer. Allocations are provided from a data structure called the heap, which is located in the stack section.

Notes

The function `zalloc` can modify the static structure **`_membase`** or the static variables **`_avail`** or **`_badlist`**; no other static data is modified.

If ANSI features are disabled, *size* is type **`unsigned int`** instead of `size_t`. See the **`-A`** option in Chapter 3, *Using Command Line Options*, for information on the compiler command line option that disables ANSI features.

C Library Customizer 6

The C Library Customizer lets you alter a Mentor Graphics general distribution library to build your own custom libraries. The C Library Customizer is actually the Microtec C compiler plus certain scripts (for this chapter, the term “script” refers to UNIX shell scripts and Windows batch files).

To customize a library, copy one of the configuration files, modify it to fit your needs, and then execute a simple script. You can validate such a library by running a test suite provided with the compiler.

When to Create a Custom Library

Some common reasons to build a custom library:

- Eliminate the linking of the floating-point package when **printf** is used (enable the definition of the preprocessor symbol **EXCLUDE_FORMAT_IO_FLOAT_FMT**)
- Debug the library by using the **-g** option
- Use the dot character instead of the underscore to prefix C names by using the **-upd** option
- Compile to take full advantage of a specific target
- Save space by excluding selected routines
- Build libraries with single-precision floating-point by using the **-Kq** option
- Rename sections to segregate library variables from program variables by using the **-N** option

Directories in the C Library Customizer

On UNIX systems, the following directories containing the C Library Customizer are installed with the Microtec C compiler:

- **cmd** — Scripts
- **src** — Source Files
- **include** — System Headers
- **test** — Test Files

- **config** — Configuration Files
- **lib** — Library Files and Linker Command Files

These directories are write protected. You must copy the directories into your own work area before you can start building libraries. Please contact your system administrator to obtain the installation location of the C Library Customizer.

On Windows systems, the C Library Customizer is available in the directory **C:\MCC68K\RTL**. However, you should also make a working copy of the C Library Customizer in the directory in which you will build the custom libraries.

All scripts are designed to run with the **cmd** directory as the default directory and they assume the given directory structure. The **cmd** directory is where you will probably do most of your work.

The **config** directory contains all the Mentor Graphics general distribution configuration files, and is where you will place your customized configuration file. On Windows, the **config** directory contains additional sub-directories containing configuration files, which correspond to the libraries and library sub-directories provided on the distribution.

Windows System Requirements

The C Library Customizer build and test scripts make frequent use of environment variables, so you should specify an environment size of at least 2048 bytes. You can specify this environment size by placing a command in your **config.sys** file:

```
shell=c:\command.com /e:2048 /p
```

The actual path to your **command.com** file may vary for your system. Please refer to your Windows documentation for more information about the **shell** command.

Using the C Library Customizer

To build a customized library using the C Library Customizer, following these steps:

1. Create a custom configuration file
2. Build the library using the **bld_lib** script
3. Test the custom library using the **test_lib** and **test_one** scripts

Note

Library testing assumes the availability of the simulator debugger. If you have a different debugger (monitor or emulator versions), you will need to modify the **test_lib** or **test_1.bat** scripts.

Step 1: Creating a Custom Configuration File

Before you can generate a library, you must generate a custom configuration file that defines how the library will be built. A configuration file is a text file that you can edit with a text editor. It acts as a header file that is read by the compiler, and contains compiler options in the form of **#pragma option** directives. The argument to the **#pragma option** directives are actually compiler command line options.

To generate a configuration file, follow these steps:

1. In the **config** directory, select the general distribution configuration file which most closely matches the library you are going to build. If you want to start with a configuration file which only defines defaults, select **a_type.h**.
2. Copy the existing configuration file to a new file. The base name of the configuration file must be identical to the base name of the library you are building. So if the library is to be named *library.lib*, then the configuration file must be named *library.h*.
3. Edit the new configuration file to reflect your choice of compiler options.

Note

If you are using Windows, do not use a name with more than 8 characters to label your new library.

There are several preprocessor symbols that may be defined to reconfigure the libraries. These symbols are also defined using the **#pragma option** directive. Currently, there are options in the configuration files that will reconfigure the formatted **printf** and **scanf** I/O routines using compiler preprocessor symbols. Enabling these symbols allows you to remove the features of these routines which you are not currently using.

Table 6-1 shows the preprocessor symbols used to customize I/O routines. These preprocessor symbols will disable options, flags, and formats for the **printf** and **scanf** library functions.

Table 6-1. C Library Customizer Preprocessor Symbols

Preprocessor Symbol	Function
Options	
EXCLUDE_FORMAT_IO_h_OPT	Disables the h option (indicates I/O object is a signed or unsigned short int)
EXCLUDE_FORMAT_IO_l_OPT	Disables the l option (indicates I/O object is signed or unsigned long int)
EXCLUDE_FORMAT_IO_L_OPT	Disables the L option (indicates signed or unsigned long double)
EXCLUDE_FORMAT_IO_ASSGN_SUPP	Disables the * option in input (scanf) functions. The * option allows variable precision and field width specifications
EXCLUDE_FORMAT_IO_STAR_OPT	Disables the * option in output (printf) functions. The * option allows variable precision and field width specifications.
Flags	
EXCLUDE_FORMAT_IO_MINUS_FLAG	Disables the - flag (left justify)
EXCLUDE_FORMAT_IO_PLUS_FLAG	Disables the + flag (right justify)
EXCLUDE_FORMAT_IO_SPACE_FLAG	Disables the white space flag (causes a space to be prefixed if no + or - is indicated)
EXCLUDE_FORMAT_IO_SHARP_FLAG	Disables the # flag (output is to be printed in an alternate form)
EXCLUDE_FORMAT_IO_ZERO_FLAG	Disables the 0 flag (pads field width with zeros)
Formats	
EXCLUDE_FORMAT_IO_BRAKT_FMT	Disables the [format (convert character sequences)
EXCLUDE_FORMAT_IO_CHAR_FMT	Disables the c format (convert individual characters)

(cont.)

Table 6-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Function
EXCLUDE_FORMAT_IO_DEC_FMT	Disables the d format (convert signed integer)
EXCLUDE_FORMAT_IO_INT_FMT	Disables the i format (convert signed integer)
EXCLUDE_FORMAT_IO_FLOAT_FMT ^a	Disables the f format (convert floating-point numbers)
EXCLUDE_FORMAT_IO_NUMB_FMT	Disables the n format (convert number of characters read or written)
EXCLUDE_FORMAT_IO_OCT_FMT	Disables the o format (convert octal number)
EXCLUDE_FORMAT_IO_PNT_FMT	Disables the p format (convert pointers)
EXCLUDE_FORMAT_IO_STR_FMT	Disables the s format (interpret argument as a string)
EXCLUDE_FORMAT_IO_UNF_FMT	Disables the u format (convert unsigned number)
EXCLUDE_FORMAT_IO_HEX_FMT	Disables the x format (convert hexadecimal number)
Line Buffered I/O	
EXCLUDE_LINE_BUFFER_DEFAULT	Disable line buffering for stdin and stdout
EXCLUDE_TERMINAL_SIMULATION	Disable character echoing during input
LINE_BUFFER_SIZE= <i>n</i>	Specifies the buffer size for I/O
Non-reentrant Functions	
EXCLUDE_ATEXIT	Disables support for the atexit() function
EXCLUDE_ERRNO	Disables support for errno and functions that use errno
EXCLUDE_INITDATA	Disables the call to initcopy() , part of the initdata feature
EXCLUDE_IOB	Disables support for C I/O routines
EXCLUDE_MALLOC	Disables support for all memory allocation functions and removes global data used to support them

(cont.)

Table 6-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Function
EXCLUDE_RAND	Disables support for rand() and srand() and removes global data used to support them
EXCLUDE_SIGNAL_RAISE	Disables support for the signal() and raise() functions
EXCLUDE_STRTok	Disables support for strtok() and removes global data used to support it
EXCLUDE_TIME	Disables support for all time functions and removes global data used to support them
INCLUDE_BUILD_ARGV	Includes build_argv() for command line processing
Math Functions	
INCLUDE_FAST_POW	Incorporates a faster, but less accurate pow() function

- a. If you are not using floating-point components in your program, you may enable the preprocessor symbol **EXCLUDE_FORMAT_IO_FLOAT_FMT**, which will automatically disable **%f** processing in both the **scanf** and **printf** functions. When you disable **%f** processing, the floating-point library will not be linked in and will result in a much smaller code size.

Setting the configuration file's preprocessor symbols efficiently trims the formatted I/O routines down to minimal size while retaining the functionality you need.

After creating your configuration file, save it and move to the **cmd** directory.

Step 2: Building a Custom Library

To build a custom library, you must first make the **cmd** directory your default directory. You then execute the **bld_lib** script to create the library. The **bld_lib** script creates the library based on the contents of the configuration file.

Syntax

The following command syntax invokes the library build script for UNIX hosts:

```
bld_lib configuration_file
      [ -bindir bin_dir ]
      [ -output output_file ]
```

The following command syntax invokes the library build script for Windows hosts:

```
bld_lib subdir/configuration_file
      [ -bindir bin_dir ]
      [ -output output_file ]
```

Description

<i>bld_lib</i>	Specifies the name of the library build script.
<i>subdir</i>	Specifies the subdirectory for the library in the configuration directory (Windows only).
<i>configuration_file</i>	Specifies the name of the configuration file. The configuration file may be stated either without a filename extension or with either a .h or .lib extension.
<i>-bindir bin_dir</i>	Accesses Microtec C compiler, assembler, librarian, and linker from the directory <i>bin_dir</i> (UNIX only).
<i>-output output_file</i>	Writes the results of the build operation to <i>output_file</i> (Windows only).

The script **bld_lib** builds a custom library and places it in the **lib** directory. If the name of the configuration file was *name.h*, then the name of the library will be *name.lib*.

Once the **bld_lib** operation is complete, you are ready for testing.

Step 3: Testing a Custom Library

Once a library has been built, it must be tested. Two scripts are provided to help you with this process, **test_lib** and **test_one**. The **test_lib** script runs all available test routines. The **test_one** script runs an individual test.

Note

To prevent compilation errors, do not run the test scripts while you are running the build script.

Syntax

```
test_lib configuration_file
    [ -bindir bin_dir ]
    [ -cmdfile cmd_file ]
    [ -debugger command ]
    [ -output output_file ]
    [ -testsrcfile src_file ... ]
```

```
test_one configuration_file test_file
    [ -bindir bin_dir ]
    [ -cmdfile cmd_file ]
    [ -debugger command ]
    [ -output output_file ]
    [ -testsrcfile src_file ... ]
```

Description

<code>test_lib</code>	Specifies the name of the test script which runs a complete set of tests.
<code>test_one</code>	Specifies the name of the test script which runs an individual test.
<code>configuration_file</code>	Specifies the filename of the configuration file. The configuration file may be declared either without a filename extension or with either a .h or .lib extension.
<code>test_file</code>	Specifies the name of an individual test file. This file must be located in the test directory.
<code>-bindir <i>bin_dir</i></code>	Accesses the Microtec C compiler, assembler, librarian, and linker from the directory <i>bin_dir</i> (UNIX only).
<code>-cmdfile <i>cmd_file</i></code>	Assigns <i>cmd_file</i> as the linker command file.
<code>-debugger <i>command</i></code>	Assigns <i>command</i> as the XRAY Debugger name (default: XHStar for UNIX and Windows where <i>tar</i> is an abbreviation for 68K or CF).
<code>-output <i>output_file</i></code>	Writes the results of the <code>test_lib</code> and <code>test_one</code> build operation to <i>output_file</i> (Windows only).
<code>-testsrcfile <i>src_file</i></code>	Compiles and links <i>src_file</i> before the customized library. The file <i>src_file</i> will always be compiled with the full

debugging option (**-g**) enabled. The file *src_file* must be located in the **src** directory.

Upon completion, executable files generated by the compiler are placed in the **test** directory along with the output files generated by the test routines. These output files are named *test_file.out*.

Assessing Test Results

The output of the test scripts has the following format:

- An individual test begins with the line:

```
TESTING: test_name LIBRARY: library_name
```

where *test_name* is the name of the test file being executed and *library_name* is the name of the library which is being tested. Compiler warnings and errors will appear after this line.

- The results of the tests appear between two lines of asterisks (*). If the test was successful, the following message is issued:

```
This test has completed with no errors.
```

If any other message appears or if there is any other output along with this message, the test has failed.

Note

If there is no output between the lines of asterisks, the test failed.

- On UNIX systems, **test_lib** will print a summary of the test results. It will either indicate that the test suite was successful, or it will indicate that the test suite has failed and list those tests which failed. On Windows systems, **test_lib** will not generate a summary; the user must examine the results of each individual test.

Once the **test_lib** script has completed with no errors, the custom library is ready for use.

Debugging Custom Libraries

If **test_lib** indicates that there is a problem with the custom libraries, you can debug these libraries. If the custom libraries were built with debugging information enabled, bring up XRAY Debugger and step through the particular test that failed.

However, if the custom library was built without debugging information enabled, stepping through the library routines will be difficult. If this is the case, do one of the following:

- Rebuild the library with debugging information enabled. This method is the simplest, but it takes time to rebuild the library.
- Use the **test_one** script to recompile the test routine and those library routines where you think there might be a problem. Specify those library routines as arguments to the **-testsrcfile** option. The routines will be recompiled with debugging information enabled and they will be linked in ahead of your custom library. Once this is done, you can easily step through the library routines with the XRAY Debugger.

Once the problem has been located and corrected, you should rerun the individual test using the **test_one** script. If the results of the individual test indicate that the problem has been fixed, rerun the entire test suite using the **test_lib** script. When the **test_lib** script indicates that your custom library is passing all tests, the library is ready for use in your application.

Using Custom Libraries

To use your new custom library, link your custom library as you would with any other objects (or another library). However, your custom libraries must precede any general distribution libraries during the link.

If a compilation driver is available, you can specify the custom library in the command line, as in the following example:

```
mcc68k -o prog mod1.c mod2.c mod3.s mod4.o iBUILTIt.lib
```

In this example, the custom library will be passed to the linker before the general distribution libraries supplied by the driver.

If your custom library supersedes the general distribution library, you may want to use the driver to manually include the custom library name in the linker command file. You can specify your custom library by using the **-elinker_commandfile** driver command line option, as in the following example:

```
mcc68k mod1.c -eMyCmds
```

In this mode, the driver does not supply the Mentor Graphics general distribution libraries, which can result in a faster linking. You may want to look at the sample linker command file provided by Mentor Graphics.

Optimizations 7

Optimizations are techniques that compilers use to improve the object code they produce. The improvements include reduced program code size and increased execution speed. The extent of improvement depends on the content of an individual program, its coding style, and the compiler's ability to recognize and optimize certain constructs.

The Microtec C/C++ compilers perform many common optimizations, which are described in this chapter. Some of these optimizations are applied before, some during, and some after generating code for a C or C++ expression. Most optimizations are independent of the target processor, but some are specific to the microprocessor and the calling conventions.

In general, for a given C or C++ expression, the compiler tries to generate optimal code that is slightly in favor of reducing code size. Sometimes the compiler finds an alternate code sequence that results in significant improvement in code size but also in a substantial reduction in execution speed or vice versa. Under such circumstances, you can choose between optimizing in favor of speed or code size with the **-Ot** and **-Os** command line options. Refer to Chapter 3, *Using Command Line Options*, for details on these options.

General Optimizations

The Microtec C/C++ compilers perform several optimizations before generating code. These optimizations usually involve constant expressions or expressions with known results that can be evaluated at compile time. Programmers seldom write explicit expressions with only constants, but it is not uncommon for them to write an expression with defined constants.

Algebraic Simplification

These optimizations replace an expression with an equivalent but simplified expression.

Examples:

```

a * 1  —>  a
b - 0  —>  b
c == c —>  1
x -= x —>  x = 0

```

Redundant Code Elimination

Code sequences that produce no real side effect are considered redundant and can be eliminated. They usually evolve as the result of other optimizations. If the variable is declared as **volatile**, this optimization is not performed.

Examples:

```

a = a + 0;  —>  No code generated
b + c;      —>  No code generated
x &= x;     —>  No code generated

```

Strength Reduction

These optimizations replace operations with equivalent but less expensive (reduced strength) operations.

Examples:

```

a * 128      —>  a << 7
(unsigned) b / 16 —>  b >> 4
(unsigned) c % 32 —>  c & 31

```

Global Optimizations

The Microtec C/C++ compilers can also make programs smaller and faster by applying global data flow and control flow analysis on the entire function.

Dead Code Elimination

This optimization eliminates unreachable code (“dead code”). Unreachable code occurs through user error and conditional compilation techniques that exploit the preprocessor. Conditional expressions that can be reduced to constants also produce unreachable code. Unreachable code sequences are usually detected after performing jump optimizations. Unlabeled instructions immediately following an unconditional jump can be removed.

Examples:

```

#define TRUE 1
#define FALSE 0
#define DEBUG 0

while (a)
{
    test1();
    if (b)
        continue;
    else
        break;
    test2();
}
if(DEBUG)
    printf("debug1 \n");

while (TRUE)
{
    test1();
    test2();
}
return (b);

```

→ Dead code
 → Dead code
 → Dead code
 → Dead code
 → Dead code

```

goto 11;
i1 = 3;
i2 = 4;
i3 = 5;
11:
i4 = 3;

```

→ No code generated

```

a = ( 3 > 0 ) ? xx (6) : yy (7);
if ( a != a ) b = 5;

```

→ a=xx(6);
 → No code generated

The optimization in this example has the potential to cause confusion. Note that if the user sets a breakpoint on the line containing the statement `i2=4`, the actual breakpoint is set on the line containing the statement `i4=3`. Then, whenever a jump is made to 11, the relocated breakpoint will stop the program.

A more typical example of dead code removal applies when the compiler determines that the condition of an **if** statement equates to a constant, which occurs if every operand is a constant, or if a variable has been assigned to a constant value.

Example:

```

#define control 0
int auxcontrol;
. . .
auxcontrol = 0;
if (control | auxcontrol) >>> These lines
{                               >>> are removed
    i1 = 3;                     >>>
    i2 = 4;                     >>>
}                               >>>
else                             >>>
    i3 = 5;

```

The preprocessor symbol `control` is replaced by the text `0` and is interpreted as the constant zero. The compiler determines that the integer variable `auxcontrol` still retains the value of zero at the beginning of the `if` statement and uses zero for `auxcontrol`. The compiler evaluates the `if` expression `(control | auxcontrol)` as `(0 | 0)` or zero and causes the first part of the `if` statement to become dead code which is removed.

Factorization

Factorization optimization is particularly useful for storing a frequently used static address in a register. It references the static value or address through register indirect addressing and stores it at the beginning of the function. Factorization optimization is applied to the entire function.

Example:

```

test(1);movea.l#_test,a2
test(2);pea 1
test(3);jsr(a2)
    pea    2
    jsr    (a2)
    pea    3
    jsr    (a2)

```

Global Constant Propagation

When a constant is assigned to a variable, the constant may be carried forward to the subsequent uses of the value of that variable until a new value is assigned to that variable. This optimization is performed across basic blocks.

Example:

```

; var1 = 2;
    moveq        #2,d0
    move.l       d0,_var1    /* var1 is global variable. */
; if (var1)                /* Will not do test, var1 */
                                /* is known to be 2. */
;     i = var1;            /* 2 will be used instead */
                                /* of var1. */
    moveq        #2,d1
; f(var1,i);                /* 2 will be used instead */
                                /* of var1. */
    move.l       d1,-(sp)
    pea          2
    jsr          _f
    addq.l       #8,sp

```

Global Copy Propagation

When a variable that is “cheap” to access (for example, a register variable) is assigned to a variable that is “expensive” to access (for example, a global variable), the cheap variable may be carried forward to the subsequent uses of the expensive variable until a new value is assigned to the expensive variable. This optimization is performed across basic blocks.

Example:

```

; var1 = i;
    move.l       d2,_var1    /* i is in d2. */
                                /* var1 is global variable. */
; if (var1)                /* Will test i instead */
                                /* of var1. */
    tst.l       d2
    beq.s       L2
;     f(var1+2);            /* Function will pass i+2 */
                                /* instead of var1+2. */
    move.l       d2,d0
    addq.l       #2,d0
    move.l       d0,-(sp)
    jsr          _f
    addq.l       #4,sp
L2:

```

Global Value Propagation

If global optimization (**-Og**) is selected, the values of variables as well as constants are remembered. Once the value of a global variable is placed into a register, future references use the register value instead of re-accessing the global variable.

Example:

```

globl = 2;

i = globl;      /* The register value will be used */
                /* instead of globl */

f(globl,i);     /* if -Og, the register will be used
                /* instead of globl */

```

Register Allocation

Optimizing register usage throughout an entire routine is also referred to as register coloring. It is especially valuable for keeping most of the local variables in the registers at all times. Using data flow analysis, the compiler finds the lifetime of each variable. The register coloring algorithm can then increase the number of variables that are stored in registers by using the same register for several variables in the same routine.

Unused Definition Elimination

An unused definition is an assignment in which the left-hand side variable is either never used or used after it is reassigned. The compiler eliminates unused definitions. The C++ compiler will not eliminate local definitions that are declared as **volatile**.

Example:

```

. . .
. . .
i = j + 1;      /* if the value of the variable i is never used
. . .           again, this statement will be eliminated */
. . .

```

Example:

```

i = j + 1;      /* this assignment will be eliminated since i
                will not be used before it is reassigned */

i = k + 1;

```

Loop Optimizations

This section describes optimizations that can be applied to loop constructs.

Array Operator Synthesis (C compiler only)

Because the C language does not have an array assignment operator, **for** loops are often used to assign a value to all elements of an array or to assign one array to another.

Example:

```
char arr[HI_BND], arr2[HI_BND];
for (i=0; i<HI_BND; i++)
    arr1[i] = 'a';
for (i=0; i<HI_BND; i++)
    arr2[i] = arr1[i];
```

Without this optimization, constructs like those shown in this example would result in very inefficient code because of the number of loop iterations required.

The Microtec C compiler recognizes and optimizes constructs of the following form:

```
for (i=const1; i<const2; i++)
    arr[i]=const3;
for (i=const1; i<const2; i++)
    arr[i]=arr2[i];
```

Depending on the number of elements to be assigned, the compiler may “unroll” the **for** loop or may minimize the number of iterations required. For example, to initialize a character array, the compiler will use **move.l** rather than **move.b**, which requires only twenty-five percent as many iterations.

Stepping through a **for** loop does not work as expected with this type of optimization. The **for** loop is considered to be a single statement, so one XRAY Debugger **STEP** command is sufficient to step over the entire **for** loop.

Because this optimization may result in larger (but faster) code, it can be disabled by a compiler option. Refer to Chapter 3, *Using Command Line Options*, for information about the appropriate option.

Loop Invariant Code Optimization

Loop invariant analysis speeds up loops by eliminating computations of invariant expressions and addresses. These computations are moved out of the loop and the value is stored in a register. This optimization is valuable for removing array subscripting from a loop when the subscript is a variable or expression that is not modified in the loop.

The loop control code generated by an **if** or other similar construct will be removed at compile time if the compiler determines that the controlling loop variables will

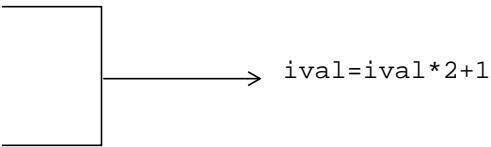
cause a loop to be executed only once. The entire loop is removed like dead code if the compiler determines that the loop control variables keep the loop from ever being executed.

Example:

```

for (i1=1; i1<=1; i1++)
{
    ival=ival*2+1;
}

```



The loop invariant code optimization searches loops for expressions that yield the same result regardless of the number of times the loop is executed. The optimization places those expressions before the loop.

Example:

```

while (i<10)
{
    test(j+k+1);
    i++;
}

```

```

L1:
    move.l    d3,d2
    add.l     d3,d2
    add.l     d3,d2
    bra.s     L2

```

```

L2:
    move.l    d2,-(sp)
    jsr       _test
    addq.l     #4,sp
    addq.l     #1,d3

```

```

    moveq     #10,d0
    cmp.l     d3,d0
    bgt.s     L1

```

Loop Rotation

In both the **for** and **while** loops, the controlling expression is located syntactically at the top of each loop. When translating these loops, the controlling expression is rotated to the bottom. This rotation reduces the number of instructions within the body of the loop.

The **while** loop on the left is translated as if it were written as the sequence of statements on the right:

<pre>while (<i>expression</i>) { <i>statements</i>; }</pre>	<pre>goto test;¹ loop: { <i>statements</i>; } test: if (<i>expression</i>) goto loop;</pre>
---	--

The **for** loop on the left is translated as if it were written as the sequence of statements on the right.

<pre>for (<i>expr1</i>; <i>expr2</i>; <i>expr3</i>) { <i>statements</i>; }</pre>	<pre><i>expr1</i>; goto test;² loop: { <i>statements</i>; } <i>expr3</i>; test: if (<i>expr2</i>) goto loop;</pre>
--	---

Strength Reduction and Index Simplification

Strength reduction and index simplification is also referred to as induction variable elimination. The values of induction variables increment or decrement by a constant value for each iteration of the loop. These variables are often used to count or index an array. Multiples of these variables can also be computed. It is often possible to replace all but one through the process of induction variable elimination.

Example:

```
for (i=1; i<10; i++)moveq#1,d1
    ary[i] = i; lea.l -76(a6),a0
    L1:
    move.l d1,(a0)+
    addq.l #1,d1
    moveq #10,d0
    cmp.l d1,d0
    bgt.s L1
```

-
1. If the compiler determines that *expression* is initially true, then this goto statement is suppressed.
 2. If the compiler determines that *expr2* is initially true, then this goto statement is suppressed.

Local Optimizations

Local optimizations are generally applied to small sections of the generated code. Some of the jump optimizations described in the *Jump Optimizations* section in this chapter can also be considered local optimizations.

Common Subexpression Elimination

Common subexpression optimizations remove previously evaluated expressions.

Example:

```
if (a+b > 3)move.ld2,d0
    test(a+b);add.ld3,d0
    move.ld0,d2
    moveq #3,d1
    cmp.l d0,d1
    bge.s L1
    move.ld2,-(sp)
    jsr _test
```

Constant Folding

Arithmetic and logical operations involving only constants are evaluated at compile time.

Examples:

<code>i = 1 + 2 * 3 + 4</code>	\longrightarrow	<code>i = 11</code>
<code>(a >> 3) >> 1</code>	\longrightarrow	<code>a >> 4</code>
<code>i + 5 + j - 6 + k - 7</code>	\longrightarrow	<code>i + j + k - 8</code>
<code>(4 & 1) (5 > 7)</code>	\longrightarrow	<code>0</code>

Generating Code for switch Statements

The Microtec C/C++ compilers apply a special optimization to generate efficient code for the **switch** statement. Depending on the number of cases and the range of case values, the compiler uses one of the following methods to generate a code sequence for the **switch** statement:

- If there are three cases or less or if the jump table is sparse, a series of compare and jump instructions is generated. These cases are compared in the same order that they appear in the program. You can improve the execution speed by placing the most frequently occurring case first.

- If the table is fairly dense (not too many holes), the compiler uses a jump table indexed by the switch value. The compiler generates a short sequence of instructions to scale the switch value and to compare to the size of the jump table before indexing. This method has the fastest execution speed of the three methods.
- If the jump table in the previous method is sparse, two parallel tables are constructed instead: one for the case values and one for the jump labels. The compiler generates a short sequence of instructions to linearly search the case value table for the switch value. If it finds the switch value, the table entry offset is used to index into the jump label table. The order of case values is the same as they appear in the program, depending on their range.

The compiler estimates the code and table size required for each method described and then selects the one with the minimum code size. When optimizing for time, the compiler will favor the second method (dense table).

Redundant Load and Store Elimination

Loading a variable after storing it is redundant and can be eliminated. This optimization can also be applied to pushing the same variable onto the stack.

Example:

```
i = a * 5;  
j = i + 5; /* i is not loaded again, since it is still */  
          /* in a register. */
```

Jump Optimizations

The Microtec C/C++ compilers perform jump optimizations after generating code for each function. The compilers examine every jump instruction and its target location to recognize specific patterns for optimization. Since most jump optimizations create opportunities for further improvements, the optimizations are applied iteratively until no further changes result.

Branch Tail Merging

The compilers combine common instructions preceding two jump instructions that jump to the same location. This optimization is enabled with the **-Og** option.

Example:

```

jsr _func1 jsr _func1
addq.l #1,d0L12: addq.l #1,d0
add.l d0,8(a6) → add.l d0,8(a6)
bra.s L1     bra.s L1
...
jsr _func2 jsr _func2
addq.l #1,d0bra.s L12
add.l d0,8(a6)
bra.s L1
...

```

In this example, the three instructions following the jump instruction on the left were combined under the L12 label.

Example:

```

1    if(i)
2    {
3        i2 = 3;
4        i1 = 7;
5    }
6    else
7    {
8        i2 = 21;
9        i1 = 7;
10   }

```

In this example, the compiler does not generate code for two `i1=7` statements. Instead, it generates code for the `i1=7` statement on line 9 and only a jump statement for the `i1=7` statement on line 4.

Code Hoisting

Similar to branch tail merging, code hoisting optimization moves common code in a program to a common point.

Example:

cmp.l 12(ab),d0		cmp.l 12(ab),d0
bge.s L3	→	lea.l (a1,d1.1),a0
lea.l (a1,d1.1),a0		bge.s L3
...		...
L3: lea.l (a1,d1.1),a0		L3: ...
...		...

This optimization may be turned off with the appropriate compiler command line option. Refer to Chapter 3, *Using Command Line Options*, for more information.

Example:

```

k = i+prime;
while (k<10)
{
    flags[k] = 0;
    k += prime;
}

```

The instruction `k += prime` at the bottom of the loop is combined with the instruction `k = i+prime` before the loop entry point. When stepping through this loop in the debugger, the program counter will step out of the loop and highlight the `k = i+prime` statement. If you examine the resulting assembly code, you can see that only part of this statement is being executed (`k=temporary_value`).

Cross-Jump Optimization

The compiler replaces a jump and some prior instructions by a jump to an earlier location that has the same instructions.

Example:

move.l d0,-(sp)		
jsr _func		
bra.s L1	→	bra.s L12
...		...
move.l d0,-(sp)		L12: move.l d0,-(sp)
jsr _func		jsr _func
L1: ...		L1: ...

This optimization can change breakpoints and perceived program flow and can affect the values of variables. This optimization may be turned off with the appropriate compiler command line option. Refer to Chapter 3, *Using Command Line Options*, for more information.

Multiple Jump Optimization

Generated code for complex control flow statements contain jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These multiple jumps can be rerouted to jump directly to the target locations.

Example:

<pre> bra.s L1 ... bne.s L1 ... L1: bra.s L2 ... </pre>	<p>→</p>	<pre> bra.s L2 ... bne.s L2 ... L1: bra.s L2 ... </pre>
---	----------	---

This optimization changes the perceived program flow.

Redundant Jump Elimination

A jump instruction that jumps to a location that immediately follows it can be eliminated. Redundant jumps usually appear after applying other jump optimizations.

Example:

<pre> bra.s L3 L3: </pre>	<p>This instruction will be eliminated.</p>
---------------------------	---

Example:

<pre> if (i3==3) goto L1; i1 = 3; i2 = 3; L1: goto L2; k = 5; L2: </pre>	<pre> >>> if(i3==3) goto L2; </pre>
--	--

In this example, the `if` statement will be changed as shown. A breakpoint at label `L1` will not be encountered when `i3==3`.

The next example shows a more typical example of redundant jumps.

Example:

```

1   while (x)
2   {
3       array(x) = 0;
4       x = x+1;
5       while (i)
6       {
7           istat(i) = 0;
8           i = i-1;
9       }
10  }
```

Short/Long Displacement Optimization

During code generation, the compiler does not know the exact displacement between a jump instruction and its target location. Many of these jump displacements are changed as a result of applying jump optimizations. As a last step in jump optimizations, the compiler determines the exact displacements and replaces them with the shortest instructions possible for conditional and unconditional jumps.

Example:

	cmpi.l #20,d0		cmpi.l #20,d0
	bne.l L1		beq.l L2
	bra.l L2	→	L1: ...
L1: ...			bra.s L3
bra.s L3			

Function In-Lining

In-line expansion of a user function saves the overhead usually associated with function calls, parameter passing, register saving, stack adjustment, and value return. Sometimes, further optimizations become possible because the actual parameters used in a call become visible, and the side effects of function calls also become exposed to the caller.

Example:

```

func(char op, int a, int b)
{
    if (op == 1)
        return(a+b);
    else
        return(a-b);
}

main()
{
    int x, y, z;
    ...

    ; Parameter passing of 1, x, y removed.
    ; Register saving in func removed.
    ; if (op == 1), comparison removed.
    ; Register restoring in func removed.
    ; Return value removed.
    ; Stack adjustment after call removed.
    ; The code generated is:

    z = func(1, x, y);      /*      move.l      d3,d4 */
                           /*      add.l      d2,d4 */
}

```

A function will not be in-lined under the following circumstances:

- The inline optimization (**-Oi** option) is not enabled.
- The function declares a local volatile variable.
- The function uses the **asm** pseudo function.
- The function is recursive.
- The invocation point is not favorable. For example, not enough registers are available to make in-lining beneficial at a given point in the program.
- The function is not defined in the same module as the call site.
- The types of function call arguments do not match the types of formal parameters.
- Too many functions have already been in-lined in the calling function.

inline Keyword

Any function may be qualified as in-line to tell the compiler which functions are the best candidates for in-lining when invoked within the same module. The Microtec Compiler extensions (**-x** option) must be enabled to accept the **inline** keyword.

Syntactically, the keyword **inline** is part of a declaration specifier and can be placed exactly as the keyword **interrupt**.

Instruction Scheduling

Improved performance can be achieved by rearranging the instruction sequence to avoid stalling the instruction unit pipeline. This includes overlapping CPU instructions with FPU instructions and inserting useful instructions between those instructions that could cause pipeline stall because of data dependence.

Machine-Dependent Optimizations

The Microtec C/C++ compilers also perform optimizations that are specific to the calling convention or architecture of the target machine. For information on compiler command line options for optimization, refer to Chapter 3, *Using Command Line Options*.

Generating Code for Function Prologue and Epilogue

The standard code sequences for function prologues and epilogues are described in Chapter 10, *Interlanguage Calling*, in this manual. The Microtec C/C++ compilers generate the full prologue or epilogue only when necessary or when the **-Kf** option is used. In particular, a prologue will not be generated if a stack frame is not needed, if there are no local variables, or if the function does not return a structure.

If only one register needs to be saved, the compilers will use the **move.l** instruction instead of the **movem.l** instruction. This saves 6 cycles.

In-Line Library Function Expansion

Certain library functions can optionally be replaced with in-line code. In addition to removing the overhead of a function call, this can also enable leaf and scheduler optimizations.

Example:

```
char stg[100]
f3( )
{
    strcpy(stg, "constant string");
}
```

Grouping Stack Adjust Instructions

The calling convention requires the calling routine (the caller) to remove the function arguments from the stack after the call. The stack pointer is adjusted by adding a constant that equals the total size of the function arguments pushed. The compiler tries to combine two or more stack adjust instructions by accumulating the adjustment to the last instruction.

Example:

```
/* code with -Oc enabled */
jsr    _function1
jsr    _function2
lea.l  12(sp),sp

/* code with -nOc enabled */
jsr    _function1
addq.l  #8,sp
jsr    _function2
addq.l  #4,sp
```

This optimization cannot be applied if the first call is one of the function arguments of the second call or if any jump or label occurs between the two function calls. The compiler accumulates a maximum of 50 bytes to reduce the stack growth. This maximum is particularly important for recursive function calls.

You can disable this optimization by using the **-nOc** option.

Indexing Arrays

When indexing into arrays that are less than or equal to 64K bytes, the index expression will be computed in **short** rather than **int**. This will result in faster but not smaller code.

This optimization will not be applied to arrays of unknown size (which can be declared as []), arrays of size 1 (sometimes used to denote an unknown size), or arrays of greater than 64K bytes.

Char Operations Where Possible

When comparing masked 16-bit expressions with 8-bit values or when assigning 16-bit expressions to 8-bit results, 8-bit arithmetic is used when the result is not affected.

Example:

```
char c;
int i, j;
/* 16-bit operations required (int result) */
j=i+2;

/* 8-bit operations allowed (char result) */
c=i+2;

/* 8-bit shortcut for masked int */
if ((i & 255) == 27)
```


Template Instantiation 8

This chapter describes the fundamentals of the Microtec C++ Compiler template implementation. A template is a feature in C++ that replaces most uses of macros. It also has some unique properties that cannot be simulated with macros.

Templates Versus Macros

Templates can be considered as “smart macros.” Both templates and macros can be used to define generic functions or classes. The abstraction of these generic definitions is important in the design of reusable and maintainable software.

The following three examples show the ways in which macros and templates are commonly used to implement a generic “swap” function, which can be used with any type **T**.

In-Line Macro

In the following example, the in-line macro generates code where it is invoked to swap the values.

Example:

```
#define SWAP(T,X,Y) \
do { T tmp_var; tmp_var=X; X=Y; Y=tmp_var; } \
while (0) /* expand macro when used */

SWAP(int, int_var_X, int_var_Y);
SWAP(String, String_var_X, String_var_Y);
SWAP(int, int_var_Z, int_var_Y);
SWAP(String, String_var_Z, String_var_Y);
```

Macro Function

The following example uses a macro called `DEF_SWAP_FUN` to declare a function that does the swap. This function is actually called by invoking the `SWAP_FUN` macro with two sets of parentheses. The type is specified within the first set, and the values to be swapped are specified within the second set.

Example:

```
/* declare the function */
#define SWAP_FUN(T) swap_function_##T
#define DEF_SWAP_FUN(T) \
void SWAP_FUN(T)(T &X, T &Y) \
{ \
    T tmp; \
    tmp=X; X=Y; Y=tmp; \
}

/* create instances */
DEF_SWAP_FUN(int)
DEF_SWAP_FUN(String)

/* use instances in the function */
SWAP_FUN(int)(int_var_X, int_var_Y);
SWAP_FUN(String)(String_var_X, String_var_Y);
SWAP_FUN(int)(int_var_Z, int_var_Y);
SWAP_FUN(String)(String_var_Z, String_var_Y);
```

Function Template

The following example uses a function template:

Example:

```
template <class T> void swap(T &X, T &Y)
{
    T tmp; tmp=X; X=Y; Y=tmp;
}

/* use automatically generated instances */
swap(int_var_X, int_var_Y);
swap(String_var_X, String_var_Y);
swap(int_var_Z, int_var_Y);
swap(String_var_Z, String_var_Y);
```

Templates are preferred over macros for the following reasons:

- Template names are recognized by the C++ compiler, so better semantic checking can be performed. Macro names are lost after the C++ preprocessor.
- A template unit has its own scope and can avoid naming problems in the definition of macros. For example, in the **SWAP** in-line macro example, the name **tmp_var** cannot be used as the argument for **X** or **Y**.
- The creation of template instances is automatic. While macros can be used to define generic functions or classes, their instances must be manually created before they can be used.
- Template functions used in multiple compilation modules are not duplicated in the final linked program.

In **DEF_SWAP_FUN**, if **SWAP_FUN(T)** is defined as static, it is duplicated in every module and explicit instances have to be created in each module. If it is a global function, each instance of **SWAP_FUN** has to be defined in exactly one module.

- Arguments to templates are typed and checked for each instance. This catches errors in the arguments of template instances and also requires users to give each template a well-defined functionality.

Macros do not have type checking. For example, **SWAP(int, a_int_var, a_char_var)** is usually accepted by a C/C++ compiler with automatic conversion between **int** and **char**, which may be incorrect in some situations.

Compile-Time Template Instantiation

The Microtec C++ compiler implements a compile time template instantiation strategy. This strategy gives users early error messages, full control over the scope of template instances, easy creation of class libraries, and, in most cases, the shortest edit-compile-run cycle. The following is a typical organization of source files.

Note that both the **stack.h** file and **stack.def** file have inclusion guards to avoid multiple inclusion. Source files such as **user1.cc** include **stack.h**, which includes **stack.def**, in order to obtain the declaration and definition of the stack template.

Note

The nesting level of template class instantiation has a limit of 17.

Declare the Class Template Stack in stack.h

Example:

```

/* to avoid repeated inclusion */
#ifndef STACK_H
#define STACK_H

/* include required header files for Stack declaration */
#include <stdlib.h>

/* the declaration of Stack template */
template <class X> class Stack {
    public: static int some_data;
    void push(X);
    ...
};
#ifndef STACK_DEF
#include "stack.def"
#endif
#endif /* STACK_H */

```

Define the Class Template in stack.def

Example:

```

#ifndef STACK_DEF
#define STACK_DEF
#ifndef STACK_H
#include "stack.h"
#endif

/* include required header files for Stack implementation */
#include <malloc.h>

template <class X> void Stack<X>::push(X data) {
    /* generic code for push */
}

void Stack<double>::push(double data) {
    /* special code for Stack<double> */
}

template <class X> int Stack<X>::some_data = 1234;
/* default initial value for an instance */

int Stack<double>::some_data = 2468;
/* special initial value for instance Stack<double> */

#endif /*STACK_DEF */

```


Use Template Class in Source File user1.cc

Example:

```
#include "stack.h"
...
Stack<int> myStack;
int n;
myStack.push(n);
```

Automatic Instantiation

By default, the Microtec C++ compiler automatically generates all template instances used in each module of an object file. Duplicated instances are skipped at link time, so there is no duplication in the final program.

To avoid duplication in object files, the **+tm** option can be used to disable the creation of template instances in a compilation. Template instances that are used but not generated should be generated in some other modules and linked into the final program.

Note that template instances are checked only when they are created. Some syntax errors in a template definition are caught only if the template is instantiated. This is a requirement by the C++ language standard. Thus, using the **+tm** option also disables the syntax and semantic checks of template instances.

Scope of Template Instances

A template declaration or definition usually contains names undefined in its local scope. The binding of names in a template depends on the scope of the template instance.

The Microtec C++ compiler implements the binding rules described in *The C++ Annotated Reference Manual*. All names used in a template definition are bound according to the following two rules:

- If a name does not depend on the template arguments, it is bound in the context of the template definition.
- If a name might depend on the template arguments, it is bound by the file scope immediately before the first place where the template instance is used.

The programmer is responsible for including a template definition at the right place in a file. To avoid inconsistent scoping of template instances, include template definitions in the header file that declares the template. All modules that use a specific

template should include the header file for that template to obtain the correct declaration of the template, as well as the correct scope of the template definition.

Manual Template Instantiation

To have the shortest compile-and-link cycle or to save disk space, manual control of template instantiation is necessary. However, you should use these manual controls only as an optimization after a working version is built.

The **+tu** option is selected by default to generate all used template instances in a module's object file. To avoid duplication, **+tm** can be used on some compilation modules to turn off the creation of template instances. Note that you can still use all template instances when compiling with **+tm**, but you need to create those instances in some other module and link them to the final program.

To further control the creation of template instances, two instantiation pragmas are provided:

```
#pragma instantiate a_template
#pragma do_not_instantiate a_template
```

For example, if you want to design a C++ library that uses class **Stack<int>**, you might want to put only one instance of **Stack<int>** in your library and turn off its instantiation everywhere else. You can put the following statement in your library header file:

```
#ifndef BUILDING_MY_LIBRARY
#pragma do_not_instantiate Stack<int>
#else
#pragma instantiate Stack<int>
#endif
```

Hence, all users of your library do not need to spend time creating the **Stack<int>** instance, because the instance is explicitly created in the library.

In the instantiation pragmas, *a_template* can be one of the following:

Template class name:	<code>Stack<int></code>
Member function name:	<code>Stack<int>::push</code>
Static data member name:	<code>Stack<int>::some_data</code>
Member function declaration:	<code>void A<int>::f(int, char)</code>
Template function declaration:	<code>char* foo(int, float)</code>

When a template class name is used in an instantiation pragma, all member functions and static data members of that class are affected.

A member function name can be used only when it is not overloaded. If there are overloaded member functions, a full declaration of a member function should be used.

Specialization

Special definitions of template instances must be seen before they are used. Otherwise, the compiler generates instances from the generic template definition.

To make it easier for users to define special instances, the Microtec C++ compiler does not consider multiple special instances an error at link time. For example, in **stack.h**:

```
template <class T> class Stack {
public: static int some_data;
...
};

template <class T> int Stack<T>::some_data = 1;
// initial value for the generic definition of some_data

int Stack<int>::some_data = 2;
// special definition of Stack<int>::some_data
```

in **a.cc**:

```
#include "stack.h"
...
Stack<char>::some_data; // get value 1
Stack<int>::some_data;  // get value 2
```

in **b.cc**:

```
#include "stack.h"
...
Stack<char>::some_data; // get value 1
Stack<int>::some_data;  // get value 2
```

When you compile and link **a.cc** and **b.cc**, instances of `Stack<char>::some_data` and `Stack<int>::some_data` are created in both **a.o** and **b.o** files. The Microtec linker removes the duplicated instances without any error or warning message.

On the other hand, if the special definition of `Stack<int>::some_data` is moved from **stack.h** to **a.cc**, **a.o** gets this special definition, but **b.o** gets the generic definition. Depending on your link options, you might not get any warning of this inconsistency. In the final program, whether `Stack<int>::some_data` has an initial value of 1 or 2 is unknown.

Precompiled Header Files 9

Precompiled header files can be used to shorten compilation time for source files that include several of the same headers. Precompiled header files are produced by saving a “snapshot” of the current state of compiling source code to a file so that when you compile the same source again or compile different source files with the same set of headers, the compiler can recognize the “snapshot point” and read in the previously-compiled state.

For the 68000 and ColdFire families, only C++ source files have precompiled header files.

Precompiled Header File Processing

Using the **-jH** option on the command line enables automatic precompiled header processing. The compiler automatically looks for a usable precompiled header file to read in, or creates one for use on subsequent compilations.

A precompiled header file contains a snapshot of all code preceding the header stop point. The header stop point generally occurs at the first token in the primary source file that does not belong to a preprocessing directive. It can also be forced by using **#pragma hdrstop**.

Examples:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

In the preceding code, the header stop point occurs at the `int` declaration; the precompiled header file for this source contains a snapshot showing the inclusion of the files `xxx.h` and `yyy.h`.

```
#include "xxx.h"
#ifndef YYH
    #define YYH 1
    #include "yyy.h"
#endif
#if TEST
    int i;
#endif
```

In this example, the first code that does not belong to a preprocessing directive is the `int` declaration again. However, the `int` declaration occurs as part of a `#if`

block. In this case, the header stop point occurs just before the `#if` that begins the block containing the `int` declaration.

Precompiled header files are produced only if the header stop point and the code preceding it meet the following requirements:

- The header stop point must occur at file scope. It cannot be with an unclosed scope established by a header file. For example, no precompiled header file is produced for the following files:

```
// xxx.h
class A{

// xxx.C
#include "xxx.h"
int i;
};
```

- The header stop point cannot be inside a declaration started within a header file, nor can it be part of a declaration list of a linkage specification. For example, no precompiled header file is generated for the following files, since the **`int`** declaration is a continuation of the declaration begun in **`yyy.h`**:

```
// yyy.h
static

// yyy.C
#include "yyy.h"
int i;
```

- The header stop point cannot be inside a **`#if`** block or a **`#define`** statement started in a header file.
- The processing before the header stop point must be error-free.
- No references to the predefined macros **`__DATE__`** or **`__TIME__`** can occur.
- The **`#line`** preprocessor directive cannot be used.
- The code cannot contain **`#pragma no_pch`**.

Precompiled header files contain information that can be checked to determine when they can be reused. This information includes the following:

- Compiler version, including the date and time the compiler was built
- Current working directory
- Main source file directory

- Command line options
- Initial sequence of preprocessing directives from primary source file
- Date and time of header files specified in **#include** directives

This information is called the precompiled header file prefix. If the prefix for a precompiled header file matches the data for a file being compiled with the **-jH** option, the remainder of the precompiled header file is read in.

If a source file can use more than one precompiled header file, the one representing the most preprocessing directives from the primary source file is used. For example, if a primary source file begins with the following:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

and there are precompiled header files for "xxx.h" and for both "xxx.h" and "yyy.h," the latter is used if both files match the current compilation. Further, after the precompiled header for the first two header files has been read in and the third one compiled, a new precompiled header file encompassing all three headers is created.

Using Options With Precompiled Header Files

Several options interact directly with precompiled headers. These include the **-jH**, **-jHc**, **-jHd**, and **-jHu** options. These options are described in Chapter 3, *Using Command Line Options*.

Some options cannot be used with precompiled header options. If options specifying precompiled headers are used alone with these options, the precompiled header options are ignored and the compiler generates a warning message. Options that cannot be used with precompiled headers include: **-C**, **-E**, **-EP**, **-Ep**, **-Es**, **-Fli**, **-Flp**, **-Flt**, **-I**, **-P**, and **-Ps**.

Another set of options is independent of precompiled header file operations. Their settings are not saved in precompiled header files. These options include the following: **-Fc**, **-Fe**, **-jH**, **-jHc**, **-jHm**, **-jHu**, **-L**, **-m**, **-N**, **-NB**, **-ND**, **-NG**, **-NI**, **-NL**, **-NM**, **-NS**, **-NT**, **-NZ**, **-Nd**, **-Nm**, **-Nx**, **-o**, **-QA**, **-Qe**, **-Qfn**, **-Qfs**, **-Qi**, **-Qme**, **-Qmi**, **-Qms**, **-Qmw**, **-Qo**, **-Qs**, **-Qw**, **-Q**, **-q**, **-S**, **-s**, **-T**, **-v**, **+w**, **-Vb**, **-Vw**, **-V**, **-X**, and **-y**.

Any option not listed above has its state saved in precompiled header files. If the option is used from the command line, rather than with the **#pragma option** directive, it must match exactly to be able to reuse a precompiled header file.

Using the Preprocessor With Precompiled Headers

There are several preprocessor macros and pragmas that interact with precompiled header processing. Some of these define properties used by precompiled header files.

The following pragmas can affect precompiled header processing:

- **#pragma asm**

When **#pragma asm** or **#pragma endasm** appears in the main source file, it behaves as **#pragma hdrstop** for precompiled header processing. This prevents repeated execution of assembly code denoted by these directives.

- **#pragma hdrstop**

This pragma forces a header stop point.

- **#pragma no_pch**

This pragma disables the use and generation of precompiled header files.

- **#pragma option**

If a precompiled header file uses **#pragma option**, the same options are set upon reuse of the precompiled header file. Command line options must always match for precompiled header files to be used, regardless of the state of options set using **#pragma option**.

In addition, pragmas that have side effects that cannot be reproduced (such as **#pragma info**, **#pragma error**, **#pragma macro**, and **#pragma warn**) will not be executed.

Precompiled Header File Usage Guide

There are two ways to use precompiled header file options. One is to use the **-jH** option, and the other is to use the **-jHc** and **-jHu** options.

Using the **-jH** option is the simplest approach. The compiler searches for all precompiled header files in the current directory or the directory specified by the **-jHd** option, looking for a best-match precompiled header file to use for the current compilation. If no suitable precompiled header files are found, a new precompiled header file is created with the same base name as the source file and has the extension **.pch**. This file is created in the directory specified by the **-jHd** option or in the current directory if none is specified.

By creating multiple precompiled header files, the **-jH** option provides the greatest possibility of reusing precompiled header files. However, it might create more precompiled header files than necessary if most of the source files use different sets of header files. This situation can consume disk space quickly, since each precompiled header file can take up half a megabyte or more.

The **-jHc** and **-jHu** options are designed to minimize the space used by precompiled header files. One way to avoid the creation of excess precompiled header files, and to still reuse precompiled header files as much as possible, is to pick one key source file to generate a single precompiled header file to be used by all the other files. For example, your key source file might contain the following lines:

```
#include <iostream.h>
#include "mycommon.h"
#pragma hdrstop
#include "extra.h"
```

When you compile this file with the **-jHc mypch.pch** option, the file **mypch.pch** is created with the compiled information of **iostream.h** and **mycommon.h**. Then you can compile the remaining files using the **-jHu mypch.pch** option. The precompiled header file will be reused for all files that have an inclusion sequence starting with **iostream.h** and **mycommon.h**.

A precompiled header file is reused only when all included header files have unchanged time stamps and when many of the command line options are the same. This is usually not a problem for a project that compiles multiple files with a single set of options for all the files. When you compile the same file the second time with the same option, the precompiled header file created in the previous compilation is reused unless some header files were changed.

Precompiled header files should be considered as intermediate files when performing parallel compilations. There is no concurrency control regarding the access of precompiled header files. Concurrent compilation with the **-jH** or **-jHc** option is not recommended, in order to avoid concurrent write to the same precompiled header file. However, it is generally safe to use the **-jHu** option with concurrent compilation. If the same precompiled header file is accessed by two processes at the same time, concurrent write may result in corrupted precompiled header files, and concurrent write and read can cause the read operation to fail. If a precompiled header file is found corrupted, it is deleted. When a precompiled header file read operation fails, normal compilation (without precompiled header files) executes and generates the correct code.

Interlanguage Calling 10

This chapter describes C, C++, and assembly language calling conventions for the Microtec C/C++ compilers. This chapter provides enough information for you to write assembly language routines to interface with C or C++ language functions. The Microtec C/C++ compilers also make use of several intrinsic functions that are tuned for performance and do not follow any calling convention.

Parameter Passing

Assembly language routines must follow the C/C++ parameter passing conventions to pass values to or receive values from C/C++ functions. C/C++ functions pass parameters by value on the stack. Parameters are always evaluated from right to left, with the first parameter evaluated last. The next section describes the conventions for procedure calls.

Pushing Parameters

All parameters are pushed on the stack in the parameter area. The first parameter in the source program is placed at the lowest address as shown in Figure 10-1.

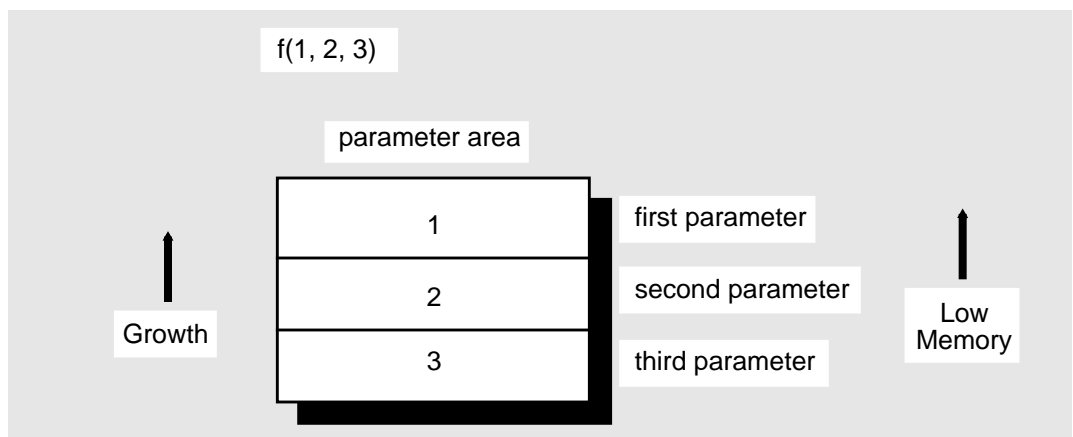


Figure 10-1. Parameter Area of the Stack

Setting Short Integers

The **-Zp2** and **-Zp4** options affect parameter promotion; see Chapter 3, *Using Command Line Options*, for more information.

Without function prototyping, **short** integers (byte or word) are pushed on the stack as long words. **short** integers are usually extended by padding or sign extension before they are pushed (see Figure 10-2).

If function prototyping is used with the **-Zp2** option, only a short word is pushed onto the stack for a **short** integer argument. If the **-Zp4** option is specified and a **short** argument is passed to a function with a **short** prototype parameter, it is not extended. It is pushed as a long word, but the upper two bytes are meaningless.

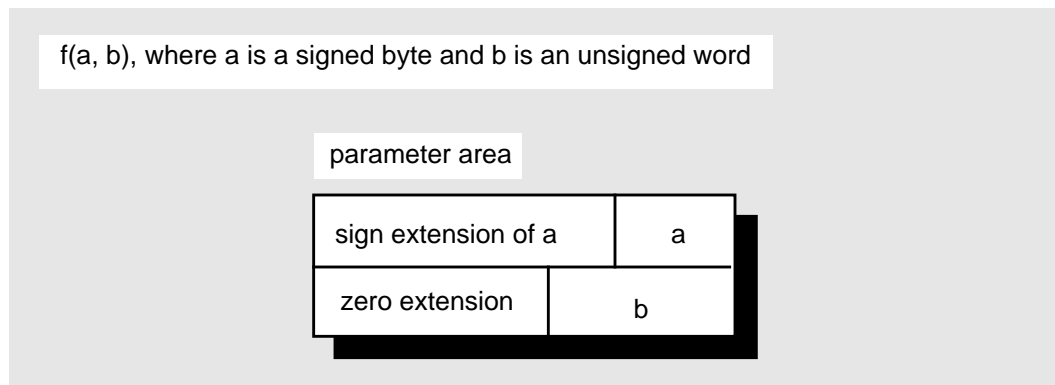


Figure 10-2. Parameter Area and Short Integers

The **-Zp2** option generates incorrect code if one of the following occurs:

- A function with **char/short** parameters is defined with a prototype in one module and is called from another where it is not declared with a prototype.

Example:

```
module1:
    f(char c, short s, int i)
    {
        ...
    }

module2:
    extern int f();
    main()
    {
        int i;
        char c;
        short s;

        f(c,s,i);           /* Bad code generated */
    }
```

- A function call is made by dereferencing a pointer to a function which contains the address of a function which has been declared with a prototype containing a **char/short** parameter.

Example:

```

f(char c, short s, int i)
{
    ...
}

int (*fp)();
main()
{
    int i;
    char c;
    short s;

    fp = f;
    (*fp)(c,s,i);    /* Bad code generated */
}

```

Implicit Parameter for Structure/Union Return Value

When the return value type is a structure or union, the calling function pushes the pointer to the area in which the return value is set by the called function. This return value area should be allocated by the calling function as shown in Figure 10-3.

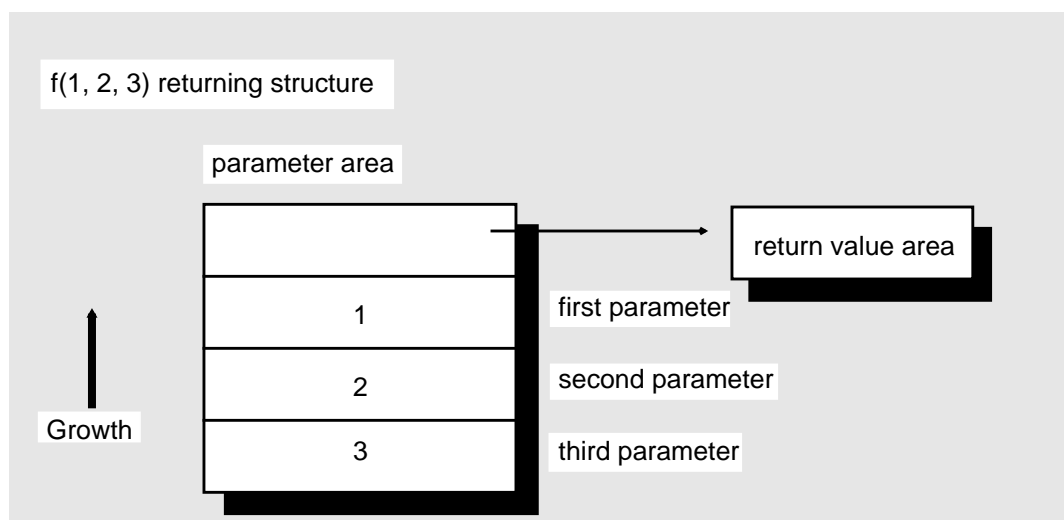


Figure 10-3. Parameter and Return Value Areas

Alignment of Structure/Union in Parameter Area

If the parameter is a structure or union, and its size is not a multiple of 2, the size of the parameter area is rounded up to be a multiple of 2 and the parameter data is set at the beginning of this area. This technique is illustrated in Figure 10-4.

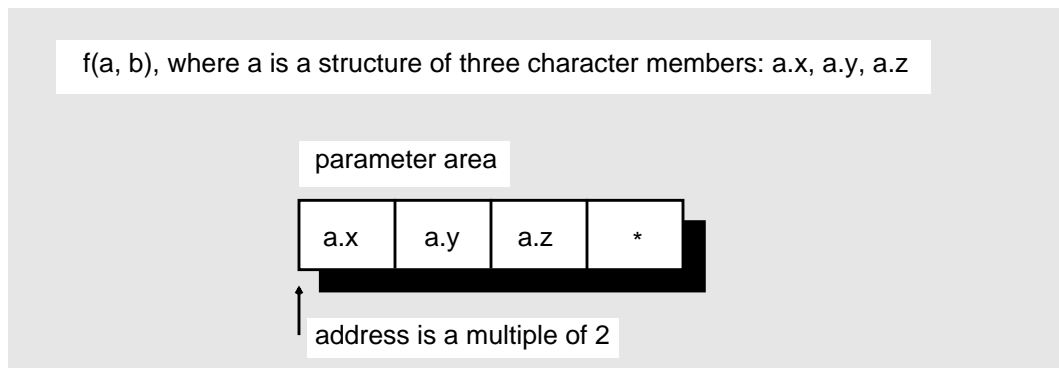


Figure 10-4. Parameter Area and Structure/Union

Function Return Values

This section describes the type of return values that are possible.

An integer or pointer type return value is set in **D0**. An integer return value with a size of less than 4 bytes is not extended to a long word before returning.

A floating-point return value with FPU is set in **FP0**. A floating-point return value without FPU is returned in **D0/D1**, with **D0** containing the high order value.

A structure or union return value is returned through the implicit first parameter which points to the return value area. When a structure return value is assigned to another structure or pushed onto the stack as a parameter, the implicit parameter that indicates the return value area can be assigned as in the following example.

Example:

```
struct s {
int  r, i;
} a, f();    <implicit parameter is the address of a>
:
a=f ( );          PEA    _a
:      JSR      _f
:      ADDQ.L   #4, SP
```

```

<implicit parameter is the address of
  parameter area>
g(f(),1);
PEA    1
PEA    -8(A6)
JSR    _f
ADDQ.L#4,SP
LEA.L  (A6),A0
MOVE.L-(A0),-(SP)
MOVE.L-(A0),-(SP)
JSR    _g

```

Popping Parameters

The parameter area is popped by the calling function.

Register and Stack Usage With Functions

The Microtec C/C++ compilers use register **A7** as the stack pointer and register **A6** as the frame pointer. All other registers store temporary variables and intermediate results.

The compiler can also be forced to reserve a register (**A2** through **A6**) by using the **-Kh** option. When a register is reserved, the compiler does not generate any code that uses the reserved register. If the **A6** register is reserved, **A5** will be used as the frame pointer.

Functions that are not declared as interrupt handlers are assumed to destroy the registers **D0**, **D1**, **A0**, and **A1** (and **FP0** and **FP1** if a floating-point coprocessor is present). At the return of a function, the condition codes are undefined. All other registers that are used will be saved and restored automatically by a called C/C++ function. Assembly language routines that are called from C/C++ must also save and restore all registers that are used, except for registers **D0**, **D1**, **A0**, and **A1** (and also **FP0** and **FP1** if the floating-point coprocessor is present).

Stack Frames

Local frames for functions are automatically generated by the Microtec C/C++ compilers if a function requires local stack storage. Frames are generated by a **LINK** instruction on entry and an **UNLK** instruction on exit. When a frame is generated, local stack storage and variables are accessed on the stack with the **A6**-relative addressing mode.

Note

Modifying the stack pointer can cause unpredictable run-time errors.

If a function does not require local stack storage, local frames are not generated by default. When a frame is not generated, variables and parameters are accessed on the stack with the **A7**-relative addressing mode. You can force the use of local frames by using the **-Kf** option.

When a function returns, any arguments pushed on the stack are removed. If several function calls follow each other in a basic block, the stack continues to grow until the end of the block. You can force the compiler to pop the stack after each function call by using the **-nOc** option. Use the instructions in Table 10-1, depending on how many bytes need to be removed.

Table 10-1. Instructions for Removing Bytes from Stack

Number of Bytes to Remove	Instruction to be Used
1 to 8	ADDQ.L XX,SP
8 to 32768	LEA X(SP),SP
More than 32768	ADD.L X,SP

Function frames larger than 32K bytes are much less efficient than smaller frames. The 68000 addressing modes are designed for 16-bit offsets. When offsets are larger than 16 bits, every machine instruction with a large offset has one to three instructions added to it in order to bring the offset within range.

The Prologue

The prologue is the sequence of code at the beginning of a function. It sets up a local stack frame and a frame pointer so that function parameters and automatic variables can be accessed through the pointer. The prologue can also include code that saves certain registers on the stack.

You can write your own assembly routines to interface to C/C++ programs. However, if the routines change the contents of any of the registers **D2** through **D7**, **A2** through **A5**, or **FP2** through **FP7**, you must save the old values.

The following example shows the recommended instruction sequence for entry to an assembly language routine.

Example:

```
entry:
    link    a6, #-N
    movem.l reglist, -(sp)
```

In this example, N is the number of bytes allocated for local variables and `reglist` is the list of registers that the routine can modify. The last argument pushed by the function call (which is also the first argument given in the call's argument list) is at address (**A6+8**). Subsequent arguments begin at (**A6 + 8 + the size of preceding arguments**).

Local Variables in the Prologue

Local (automatic) variables are located on the stack with negative displacements from the frame pointer, and there can be a maximum of $2^{31}-1$ bytes addressable by the frame pointer with negative displacements.

If you write your own applications and the assembly language routine does not have function arguments and local variables, it is not necessary to set up a local stack frame. Your routine should save the contents of the registers **D2** through **D7**, **A2** through **A5**, and **FP2** through **FP7**, if they are used.

The Epilogue

The epilogue is the sequence of code at the end of a function. The epilogue restores the saved registers and resets the stack frame to the condition at entry. It then passes control back to the calling routine with an **RTS** instruction.

Registers that are saved in the prologue should be restored first. Reset the local stack frame by loading the stack pointer with the frame pointer. The calling routine's frame pointer is then restored by popping the saved value from the stack.

The following is the recommended instruction sequence for exiting a routine:

```
movem.l    -N(a6), reglist
unlk      a6
```

In this example, N is based on the number of bytes allocated for local variables and the number of registers in `reglist`.

Assembly Language Interfacing

Create a minimal local stack frame when you interface to an assembly language routine, in order to keep local data on the stack. This also allows the Debugger to display assembly language routine names in the procedure traceback viewport.

To write an assembly language routine that can be called from C/C++, perform these steps:

1. Reference C/C++ global and external variables.

Prefix all global (that is, public) and external names with an underscore.

2. Create a local stack frame.

Create a minimal stack frame using the **LINK** instruction. **LINK** saves the old frame pointer (**A6**) of the calling routine on the stack and directs the new frame pointer (**A6**) to the old frame pointer.

The **LINK** instruction can also create a local data storage area in the stack frame if needed. By creating local stack storage in this way, you do not have to pop off the exact number of arguments that were pushed onto the stack when returning from the assembly language routine.

Example:

```
link a6,#-8
```

3. Save registers.

Save any of the registers **D2** through **D7**, **A2** through **A5**, and **FP2** through **FP7** that were used in the assembly language routine. If you did not create a local stack frame, you must also save the registers **A6** and **A7** if they were destroyed by the assembly language routine. All registers should be saved in numerical order immediately after the stack frame is built. Save data registers first, followed by address registers, then floating-point registers.

Example:

```
movem.l   d2/d3/a2/a3,-(sp)
fmovem.x  fp2/fp3/fp4,-(sp)
```

4. Reference arguments (optional).

Reference arguments passed to the routine with the frame pointer (**A6**) or the stack pointer (**A7**). If you use **A7**, the offsets to the arguments must allow for the size of the local stack frame plus any saved registers. The size

of the stack frame is the size of the **A6** register (4 bytes), plus the amount of local storage allocated. If you use **A6**, offsets to the arguments must allow for the size of the saved **A6** value only.

5. Return values.

Return values should be correctly placed before returning. For more information, see the section *Function Return Values* in this chapter.

6. Restore registers.

At the end of the assembly language routine, the saved registers and the old stack frame must be restored immediately before the return instruction.

Example:

```
fmovem.x  -60(a6),fp2/fp3/fp4
movem.l   -24(a6),d2/d3/a2/a3
unlk      a6
rts
```

Assembly Language Routine Example

A prototype of an assembly language routine (named `_foo`) that can be called from a C function is shown as follows.

Example:

```
_foo: link a6,#0; Creates stack frame.
      ; Allocates 0 bytes of space
      ; for local data. Saves
      ; original A6. A6 now points
      ; to saved A6 value on stack.
movem.l d2/d3/a2,-(sp); Pushes 3 registers on stack.
      ; This instruction is used
      ; only if registers D2-D7,
      ; A2-A5 are used in the
      ; assembly language routine
      ; body.
```

```

; Stack Frame:
; 2nd argument      A6+12  SP+24
; 1st argument      A6+8   SP+20
; A6 -->Original A6  A6+0   SP+12
;      Original A2  A6-4   SP+8
;      Original D3  A6-8   SP+4
; SP -->Original D2  A6-12  SP+0
.
.
; Body of procedure.
.
.
movem.l -12(A6), d2/d3/a2; Gets saved registers from the
; stack. Values are not popped.
; The stack pointer is not
; used because other values may
; have been pushed onto the stack
; in the procedure body. This
; uses the A6 register, and the
; constant offset (12) shown
; above. You must compute it by
; adding the value given to the
; LINK instruction (0) and 4 *
; the number of address and data
; registers saved (3);
; 0+(4*3)=12. It is a negative
; offset because it is below the
; A6 register. The UNLK below
; will set the stack pointer back
; to its original value. The
; instruction is used only if
; registers D2-D7 or A2-A5 are
; used in the procedure body.

unlk a6; Undoes the stack frame.
; Restores A6 to original value.
rts    ; Returns to caller.

```

Variable References from Assembly Routines

The compiler automatically prepends an underscore to all variable names. The options described in Chapter 3, *Using Command Line Options*, can be used to change this behavior. However, they are not recommended because they prevent access to the standard C/C++ library functions, since all library functions have a leading underscore.

Interrupt Handlers

When an interrupt occurs, interrupt handlers preserve the values in various registers and storage locations, and transfer control to routines to service the interrupt.

If the **-KFi** option is used, the interrupt handler saves the contents of **FP0** and **FP1**, the status and control registers (**FPSR** and **FPCR**), and the internal state of the FPU (**FSAVE**). By default, the C/C++ compilers assume that the values in registers **A0**, **A1**, **D0**, and **D1** may be destroyed by each called function. However, if a function is declared as an interrupt handler, it forces the values of these registers to be saved on the stack on entry to the function, and to be restored on exit.

Use the **interrupt** keyword to declare a function as an interrupt handler. Interrupt functions have no parameters and return the **void** type. The following example shows how to set an interrupt function for a 68020 address error when the interrupt vector table begins at memory location 0x1000.

Example:

```
interrupt void foo_int(void)
{ ...
}

void set_vector(void)
{
    /* setting interrupt function for address
       error Vector(3) */
    *(int *)0x100c = (int)foo_int;
}

main()
{
    /* set Vector Base Register */
    asm("      move.l #1000,d0");
    asm("      movec  d0,vbr");

    set_vector();
    ...
}
```

By default, a function declared as an interrupt handler will return to the calling function with the **Return To Exception (RTE)** instruction rather than the standard **RTS** instruction.

If the **-Kr** option is used, the interrupt handler will return with an **RTS** instruction.

For more information about interrupt vector tables, refer to the Motorola reference manual appropriate to your chip.

Defining a Function

C++ is a strongly typed language that does extensive cross-checking to uncover programming errors. Every C++ function definition specifies the types of its arguments, and the compiler checks that every call to a function contains the appropriate number and type of arguments. In order to implement this feature, each C++ function is uniquely identified by the compiler. This enables the C++ programmer to use duplicate function names, which is not permitted in the C language. Using the same function name for two different functions is referred to as “overloading” the function name. The following is an example of function overloading.

Example:

```
func_1 (char);      /* unique identifier: _func_1__Fc */
func_1 (int);       /* unique identifier: _func_1__Fi */
```

The C++ function declaration is called a prototype declaration. A function prototype declaration includes the function name, return type of the function, argument types in parentheses, and a closing semicolon. A function prototype declaration must be specified before calling the function. The C++ compiler uses this prototype information (the function name and the argument types) to generate the unique identifier, known as a C++ function signature or a C++ mangled function name.

Calling C Functions From C++

The compiler must prevent the mangling of C function names, since C code does not recognize mangled names. To prevent the C++ compiler from creating the unique C++ function signature, you must inform the C++ compiler when you call a C function from your C++ program. The meaning of the C keyword **extern** has been extended for this purpose.

As shown in the following example, declare all the C functions called from C++ programs as `extern "C"` (a capital C is required). Functions declared in this manner are treated as C functions and are subject to standard C rules.

Example:

```
extern "C" {
    func_1 (char);
    func_2 (char *, int);
}
```

After a file is modified to use the `extern "C"` declaration, that file can no longer be compiled by a C compiler because the declaration is illegal in the C language. To compile the same header file with C and C++, use `#ifndef __cplusplus`, as follows:

Example:

```

#ifdef __cplusplus
    extern "C" {
        func_1(char);
        func_2(char *, int);
    }
#else
#ifdef __STDC__
    extern func_1(char);
    extern func_2(char *, int);
#else
    extern func_1(), func_2();
#endif
#endif

```

Passing Arguments

In most cases, traditional C function declarations work without complications. Most ANSI C compilers, however, “promote” arguments of type **float** (4 bytes) to type **double** (8 bytes) if a nonprototyped C function declaration is used. This behavior can confuse programmers who do not explicitly define argument types in their **extern "C"** declarations.

Example:

In a C source file, define two functions, **f()** and **g()**:

```

f (old_fd)          /* traditional C function definition */
float old_fd;       /* old_fd treated as double */
{
    ...
}

g (float ansi_fd) /* ANSI C function prototype definition */
{
    /* ansi_fd treated as float */
    ...
}

```

To call the two C functions from a C++ source file, add the following code to the C++ source file:

```

extern "C" {          /* function definitions in C++ file */
    f(double);       /* variable treated as double */
    g(float);        /* variable treated as float */
}

```

Function **f()** is defined according to traditional C, with the argument name in parentheses. Traditional C compilers promote the variable **old_fd** to **double**, even though it is explicitly defined as **float**. All subsequent uses of this variable are

treated as **double**, which might affect the results of a program. No such problem exists in ANSI C, since ANSI C compilers enforce the variable type **float** specifically defined in the function prototype definition.

The C++ compiler uses types explicitly defined in function prototype definitions for ANSI C and C++ code. For traditional C code, the C++ compiler correctly interprets the function declaration of `f()` in the `extern "C"` declaration in a C++ file as `f(double)` unless specified as `f(float)`. Although a declaration of `f(float)` seems logical, it is incorrect (since the C compiler always treats it as a **double**) and may have unwanted side effects.

To avoid confusion, explicitly define `float` arguments for traditional C functions as `double` in `extern "C"` declarations, as shown in the example, and add a comment. This explicit declaration instructs the C++ compiler to interpret the argument like the traditional C compiler did. It also clearly indicates the argument type to anyone reading the code. You should always add an explanatory comment when declaring a traditional C **float** argument as `f()` or `f(double)`. An inexperienced C++ programmer could change the traditional C function definition to `f(float)` for consistency, possibly introducing side effects into working code.

A similar approach is to redefine traditional C function definitions to change **float** variables to **double**. This method eliminates the need for promotion and avoids misinterpretation by inexperienced C++ programmers, but it requires altering the original function definitions on C source code, which might not be desirable.

Calling C++ Functions From C

This section outlines factors to consider when calling C++ functions from C.

Overloaded Functions

You cannot call C++ overloaded functions from C without explicitly invoking the proper function with the corresponding C++ function signatures or mangled names.

Member Functions

Avoid calling member functions directly. It is more systematic and less error-prone to call a C++ nonmember function or friend function layer, which then invokes the proper member functions and performs the class object operations/manipulations. This C++ nonmember function or friend function layer should be declared in C++ as the **extern "C"** linkage so that it is compatible with the C linkage name rules.

The following example shows the use of a nonmember function layer in C++ to create the interlanguage relationship needed between the C and C++ files. Figure 10-5 illustrates the relationship.

Example:

```
#include <stdio.h>
class Matrix {
    int row;
    int col;
public:
    Matrix (int row1=0, int col1=0) {
        row = row1;
        col=col1;
    }
    int get_row() { return row; }
    int get_col() { return col; }
    void self() {
        printf("Your location is: row (%d) col (%d)\n",
            get_row(), get_col() );
    }
};

extern "C" {
    void print_matrix(Matrix *);
}

void print_matrix(Matrix *mobj) {
    mobj->self();
}

extern "C" {
    int C_Matrix_get_row (Matrix *mobj);
}

int C_Matrix_get_row (Matrix *mobj) {
    return mobj->get_row();
}
/* for Matrix::get_row() */

extern "C" {
    int C_Matrix_get_col (Matrix *mobj);
}

int C_Matrix_get_col (Matrix *mobj) {
    return mobj->get_col();
}
/* for Matrix::get_col() */
```

C Code	C++ Code
<pre> extern struct Matrix; typedef struct Matrix Matrix; extern void print_matrix(struct Matrix *); extern int C_Matrix_get_col(struct Matrix *); extern int C_Matrix_get_row(struct Matrix *); void f() { struct Matrix *p; int x; . . . print_matrix(p); x=C_Matrix_get_col(p); x=C_Matrix_get_row(p); . . . } </pre>	<pre> class Matrix { . . . public: void self() { . . . } }; extern "C" { void print_matrix(Matrix *); } void print_matrix(Matrix *mobj) { mobj->self(); } extern "C" { int C_Matrix_get_col (Matrix *, mobj); int C_Matrix_get_row (Matrix *, mobj); } int C_Matrix_get_col (Matrix *, mobj) { return (mobj->get_col()); } int C_Matrix_get_row (Matrix *, mobj) { return (mobj->get_row()); } . . . </pre>

Figure 10-5. C/C++ Interlanguage Calling

To call the C++ `self()` function from your C++ code, refer to it as **Matrix::self()**. However, if you want to call the C++ `self()` function from your C code, you need another way to reference this function, because a term like **Matrix::self()** is not recognized in the C language.

To solve this problem, create a C++ nonmember function layer, also known as a C-callable C++ wrapper, in your C++ code. This wrapper is created by defining a new function `print_matrix` in your C++ code. This new function is passed a pointer to the class (`Matrix`) containing the function that the C program wants to call.

`Matrix` is defined as `struct` in the C code and `class` in the C++ code if these structures are equivalent across the languages. You must ensure that the structures are equivalent within both structures in terms of structure size, structure layout, and the

offsets of the data members. To ensure that the `Matrix` types are usable across C and C++, use an external declaration in your C code, as follows:

```
extern struct Matrix;
```

Your C++ code declares `Matrix` as follows:

```
class Matrix {...};
```

The `print_matrix` function is declared in your C code as follows:

```
extern void print_matrix (Matrix *);
```

The `print_matrix` function is declared in your C++ code as follows:

```
extern "C" {  
    void print_matrix (Matrix *);  
}
```

You must declare the function `print_matrix` as `extern "C"` in your C++ code to prevent the function name from being mangled with a function signature (the C code would not be able to recognize the function name if it were encoded). If you declare `p` as an external variable, its implementation names are the same in Microtec Compiler implementations of C and C++, which facilitates passing of global objects between C and C++.

Using the code shown in Figure 10-5, your C code call safely invokes the `print_matrix` function, which points to the C++ `self()` function contained in class `Matrix`. (The `print_matrix` function is a legal C function because of the `extern "C"` declaration.) This approach allows you to call C++ functions from C code.

You could also develop a friend function for each visible (public) member function of the class that you would call with C functions. The following example shows how to define a C-callable C++ wrapper through friend functions.

Example:

```

#include <stdio.h>
extern "C" {
    int C_Matrix_get_row (Matrix *);
    int C_Matrix_get_col (Matrix *);
    void print_matrix (Matrix *);
}
class Matrix {
    int row;
    int col;
public:
    Matrix (int row1=0, int col1=0) {
        ...
    }
    int get_row () {
        ...;
    }
    int get_col () {
        ...;
    }
    void self () {
        ...;
    }
    friend int C_Matrix_get_row (Matrix *);
    friend int C_Matrix_get_col (Matrix *);
    friend void print_matrix (Matrix *);
};

```

In this example, each C-callable C++ nonmember function of class `Matrix` is declared as a friend function to class `Matrix`.

Exception Handling Between C++ and C

The only requirement for C programs is a stack frame pointer for each function. Use `-Ze` or `-Kf` to compile the C module.

Exception Handling Between C++ and Assembly

The only requirement for assembly programs is a stack frame pointer for each function.

Consider the following system with a C++ function `foo` that calls an assembly function `goo`. The assembly function `goo` calls a C++ function `goo2`, which throws an exception. Figure 10-6 illustrates the system.

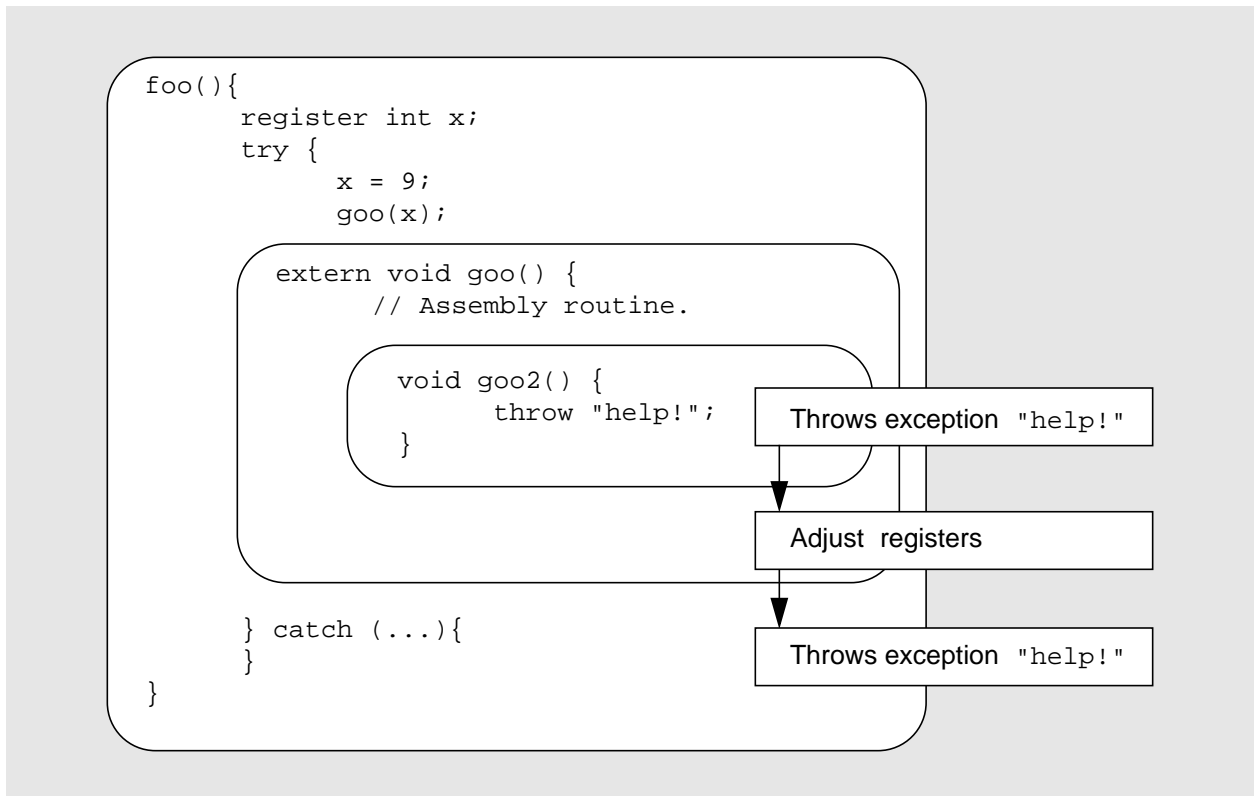


Figure 10-6. Example of an Assembly Function in an Exception Handling Call Chain

NOTE

Microtec C++ Compiler implementation of EHS since version 3.0 has been changed to FAST EHS, which is incompatible with the previous implementation.

Example:

```

extern "C" {
    extern void goo(int);
}

void goo2() { // called by the Assembly routine goo.
    throw "help!";
}

foo(){
    register int x;
    try {
        x = 9;
        goo(x);
    } catch (...) {
    }
}

```

In the preceding example, the `extern "C"` feature is used to control the linkage name of the assembly function `goo` and the C++ function `goo2`.

To modify a function for exception handling, make sure that the function has a frame. If it does not, add a **link a6,#0** instruction at the beginning of the function and an **unlk a6** instruction before the **rts** instruction in the epilogue.

Example:

```

SECTION code,,R
XREF _goo2__Fv
XDEF _goo
_goo:
    link a6,#0                ; frame set up
    ...
    jsr _goo2__Fv             ; call goo2(void)
    ...
    unlk a6                   ; pop the frame
    rts                       ; return from function

```

Internal Data Representation 11

This chapter describes internal data formats of the C and C++ languages and the run-time organization of C/C++ programs for supported microprocessor families.

Storage Layout

Memory is accessed according to the type of processor. There are two basic types of processors:

- Big-endian: addresses objects in memory from the “big” or most significant byte end. The preprocessor symbol **_BIG_ENDIAN** is defined for these processors.
- Little-endian: addresses objects in memory from the “little” or least significant byte end. The preprocessor symbol **_LITTLE_ENDIAN** is defined for these processors.

When addressing the memory shown in Figure 11-1, the instruction to load address N is handled in two different ways. A big-endian processor considers N the more significant byte, with $N+1$ as the less significant byte. A little-endian processor considers $N+1$ the more significant byte, with N as the less significant byte (see Figure 11-2).

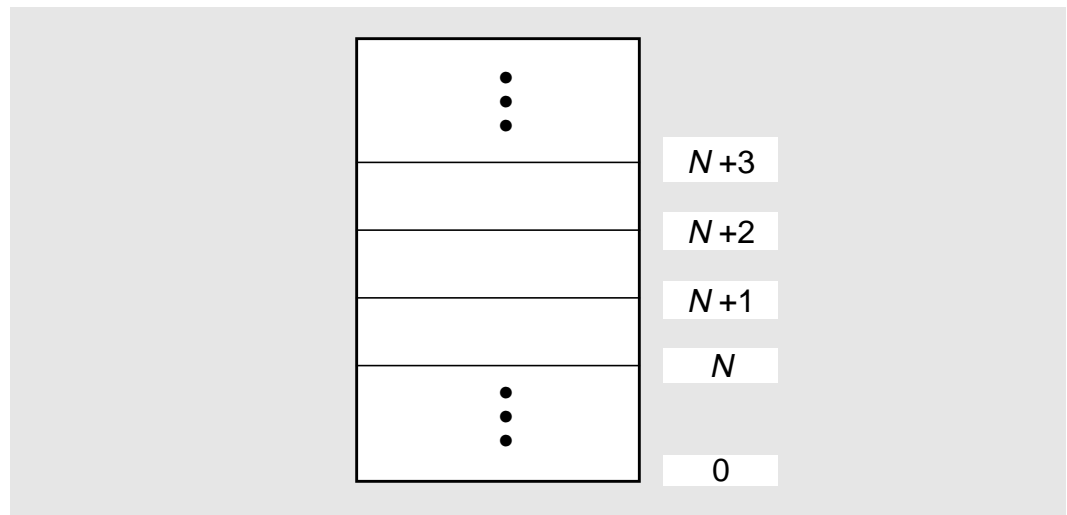


Figure 11-1. Memory Layout

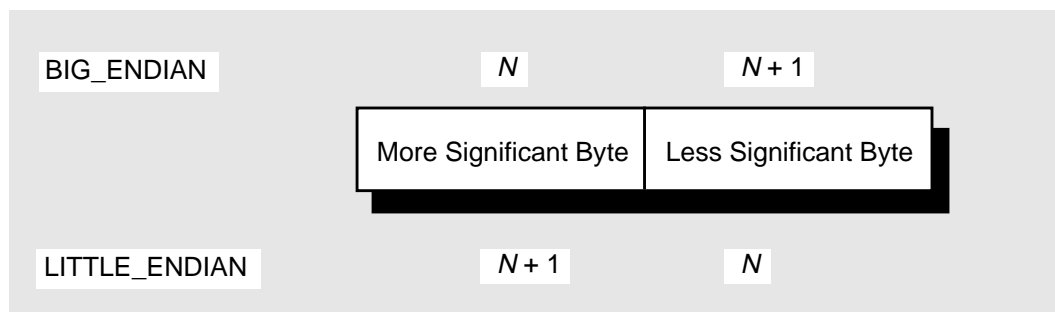


Figure 11-2. Loading Dependent on Processor Type

For a 4-byte quantity with address N in a big-endian processor, N represents the address of the most significant byte of the high-order word; the low-order word is located at address $N+2$, leaving the least significant byte at address $N+3$. The little-endian processor treats $N+3$ as the most significant byte, with N as the least significant byte. Figure 11-3 shows a 4-byte quantity.

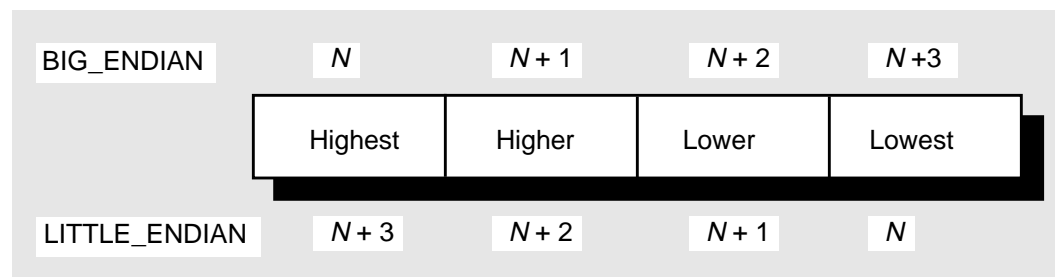


Figure 11-3. Byte Ordering in Words

Note

The term “word” refers to 2 bytes; “long word” refers to 4 bytes.

Data Type Summary

Data types for the supported microprocessor families include both scalar and complex data types.

Table 11-1 shows the size and value range of the scalar data types. The range of values are decimal representations, and the alignment is in bytes.

Pointers

Pointers occupy two or four bytes, depending on the processor, and are aligned as shown in Table 11-1.

Table 11-1. C/C++ Scalar Data Types

Data Type	Size	Range	Alignment (Processors With 16-Bit Data Bus)	Alignment (Processors With 32-Bit Data Bus)
signed char	8 bits = 1 byte	-128 to 127	1	1
unsigned char	8 bits = 1 byte	0 to 255	1	1
short int	16 bits = 2 bytes	-32768 to 32767	2	2
unsigned short int	16 bits = 2 bytes	0 to 65535	2	2
enum	32 bits = 4 bytes		2	4
int ^a	32-bit processors: 32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned int ^a	32-bit processors: 32 bits = 4 bytes	0 to 4294967295	2	4
pointer ^a	32-bit processors: 32 bits = 4 bytes		2	4
long int	32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned long int	32 bits = 4 bytes	0 to 4294967295	2	4
float	32 bits = 4 bytes	$1.18 * 10^{-38}$ to $3.4 * 10^{38}$	2	4
double	64 bits = 8 bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	2	4
long double	64 bits = 8 bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	2	4

a: The C compiler only supports the 32-bit processors.

Note

If the keyword **unsigned** does not appear, the **char** data type is considered to be signed unless the **-Ku** option is used, in which case it is considered to be unsigned.

Table 11-2 shows the size and value range of the complex (aggregate) data types.

Table 11-2. Complex (Aggregate) Types

Complex Types	Size
array	Combined size of elements
struct	Combined size of members (plus possible padding)
union	Size of largest member (must be a multiple of the -Zn setting)

Type Conversion

The compiler performs data type conversions under the following circumstances:

- When arguments are passed to a function that does not have a prototype. In this case, the function's arguments are integrally promoted as described in the section *Mixed Operands* in this chapter. [This applies to the C compiler only]
- When the data type of an operand is forced to be converted by type casting.
- When a binary operator is used (commonly referred to as “usual arithmetic conversions”).
- When two or more operands of different types appear in an assignment expression. The right operand is converted to the type of the left operand. The conversion of mixed operands is described in the section *Mixed Operands* in this chapter.

Mixed Operands

Mixed operands of binary operators are converted according to an order of precedence, as shown in Table 11-3.

Table 11-3. Mixed Operand Conversion

Operand Type	Compiler Action
Either is long double	Converts other to long double .
Either is double	Converts other to double .
Either is float	Converts other to float .
Either is unsigned long	Converts other to unsigned long .
One is long and other is unsigned	Converts both to unsigned long .
Either is long	Converts other to long .
Either is unsigned	Converts other to unsigned .

If the operands do not fit into the preceding type categories, the compiler performs the integral promotions shown in Table 11-4 to ensure that both operands have type **int**.

Table 11-4. Integral Promotion

Operand Type	Compiler Action
char	Sign-extends operand to int unless specified otherwise when the compiler is invoked. ^a
signed char	Sign-extends operand to int .
unsigned char	Zero-extends operand to int .
short int	Sign-extends operand to int .
long int	Operand treated same as int .
unsigned short int	Zero-extends operand to int .

a. See the section *Producing Minor Code Generation Variations* in Chapter 3, *Using Command Line Options*, for more information.

Type Casting

Type casting forces the conversion of an expression to the specified data type.

- Casting an integral type to a larger type causes sign-extension or zero-padding according to the type of the value being cast.
- Casting an integral type to a smaller integral type causes the low-order bits to be retained.
- Casting an integral type to a floating-point type (or vice versa) causes the bits to change completely.
- Casting a floating-point type to another floating-point type causes the high-order bits of the fractional part to be retained.

Examples:

```
(char *) 5           /* Treat 5 as a pointer to a char */
(unsigned) -1        /* Value: 4294967295 */
(long) -128          /* Value: -128, 4-byte */
                    /* representation */
(unsigned long) -128 /* Value: 4294967168 */
(char) 4294967167    /* Value: 127 */
(int) 65535           /* Value: 65535 */
p2 = (type2 *) p1;    /* Assign p1's value to p2, */
                    /* even though p2 points to a */
                    /* different type */
```

Function Operations

The following sections discuss Microtec Compiler extensions to function declarations.

Passing Prototyped Parameters Smaller Than int

When data is passed to a function, the default is to pass the exact number of bytes required to contain the value, which can result in suboptimal code. With the **-Zp2** and **-Zp4** options, values smaller than an **int** are passed as 2-byte or 4-byte values. For further information, refer to the description of the **-Zp2** and **-Zp4** options in Chapter 3, *Using Command Line Options*.

Function Declaration With interrupt Keyword

A function can be declared as an interrupt handler by using the **interrupt** keyword. For more information, see Chapter 13, *Embedded Environments*.

Example:

```
interrupt void handler (void);
```

Structure Operations

Unlike the pre-ANSI definition of C, Microtec Compiler C allows structures to be assigned, passed as parameters to functions, and returned from functions. Structure values are placed on the stack. The amount of storage needed by a structure is the sum of the space used by its members plus any padding.

The following sections discuss Microtec Compiler extensions to structure operations.

Structure Alignment

Structure members are aligned according to their type (see Table 11-1 for alignment of scalar types; see the section *Aggregates* for alignment of aggregates). The compiler might add padding between structure members and at the end of a structure so that the size is a multiple of the maximum alignment of the members of the structure. However, if the largest member of the structure is a **char**, the structure is aligned on an even boundary.

- For processors with a default alignment of 2, a structure of the type shown in the following example is allocated $4 + 1 + (1 \text{ byte padding}) + 4 = 10$ bytes.
- For processors with a default alignment of 4, the same structure is allocated $4 + 1 + (3 \text{ byte padding}) + 4 = 12$ bytes.

These rules can be modified with the **-Za2** and **-Za4** options.

Example:

```
struct date                /* structure tag */
{
    int day;
    unsigned char month;
    int year;
} holiday;                 /* structure variable name */

main()
{
    holiday.day = 25;
    holiday.month = 12;
    holiday.year = 1988;
}
```

In this example, the structure variable is `holiday` and the members of the structure are `day`, `month`, and `year`. The `date` is an identifier called a structure tag. This struc-

ture tag can be used for later definitions and declarations without repeating the structure members. For example:

```
struct date workday;
```

Determining Structure Size

You can make the compiler reveal the size of a structure and the offsets of its members. Knowing the size of a type is useful for double-checking type defaults and for establishing the size of complex structures and unions.

Example:

```
struct x {
    char a;
    long b;
    char c;
};
packed struct y {
    char a;
    long b;
};

struct_size () {
    struct_size (struct x, struct y);
}
```

In this example, nonexecutable code is used to provoke a warning from the compiler that will list structure sizes. The structure types `struct x` and `struct y` are specified as parameters to the function `struct_size`. The warning message lists the structure sizes and states that the type was replaced by the size of the structure:

```
(W) type used as argument; replaced with its sizeof: 8
```

By using the C/C++ Compilers and the XRAY Debugger, you can create a program that will generate the size of the structure members.

Example:

```
#include <stddef.h>      /* include offsetof macro */

#define print_offset(tag, member) \
    printf("\noffset of %s.%s\t%d", #tag, #member, \
        offsetof (struct tag,member))
```

```

struct x {
    char a;
    long b;
};
struct y {
    char a;
    struct x b;
    long c;
    char d;
};
main () {
    print_offset (x,a);
    print_offset (x,b);
    print_offset (y,a);
    print_offset (y,b);
    print_offset (y,c);
    print_offset (y,d);
    puts("");
}

```

In this example, each structure member will be passed to the defined macro `print_offset`. The `print_offset` macro prints offsets determined by the `offsetof` macro for all structure members.

Bit Fields

A member of a structure can be defined as a bit field. A bit field defines the number of bits of storage that the member requires. A bit field is specified by:

basetype member_name:number_of_bits;

In addition to the base types **int**, **signed int**, or **unsigned int**, the Microtec compilers provide support for bit fields of any integral type, including **char**, **signed char**, **unsigned char**, **short**, **signed short**, **unsigned short**, **long**, **signed long**, **unsigned long**, **packed** enumerated types, or **unpacked** enumerated types. Storage is allocated according to the base type. The minimum size of any bit field or group of bit fields of the same base type is the *number_of_bits* in the *basetype*. The maximum size of a single bit field is also the number of bits in its base type.

Example:

```

struct status {
    unsigned control:3;
    int data:13
} pvar;

```

Bit fields in unpacked structs are stored in a base type, starting at higher order bits. If a successive field element does not fit into the remaining space of the current base type, a new byte, word, or long word is allocated (according to the base type). The field member is placed into the new base type, starting again from the most significant bit.

Bit fields in packed structs are stored in a base type in the same way, but if there is not enough room for a bit field in the current base type, it is partially allocated to the current base type and partially to a new base type. However, if a bit field allocated in this way overlaps four or more byte boundaries, padding is added to fill out the current base type and the bit field is allocated to a new base type.

In both packed and unpacked structs, field members of zero length are not allocated any space. To fulfill the requirements of the base type of the zero-length bit field, the field members cause the current allocation unit to be completed and the next byte to be allocated so that they will be properly aligned.

When the size of the base type of the next bit field differs from the size of the current base type, padding is added so that the next bit field is aligned to its base type boundary. Padding is also added if the next element is not a bit field, regardless of the type of element. Changing type sign in a sequence of bit field declarations does not cause padding to be added (Figure 11-8).

Warning

A **signed** bit field that is one-bit wide can only have the values 0 and -1 since the high-order bit is considered to be the sign bit.

Example:

```
struct {  
    char bf1 : 5;  
    short bf2 : 3; /* size changed, pad to next word */  
    int bf3 : 2; /* size changed, pad to next long word */  
    signed int bf4 : 7;  
} sl;
```

Figure 11-4 shows the bit field allocation for the preceding example.

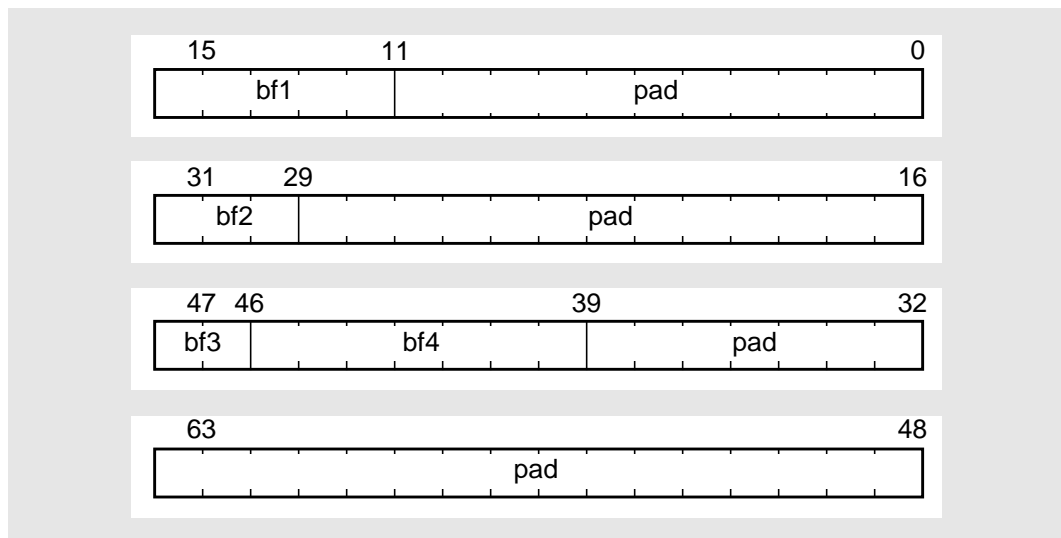


Figure 11-4. Sample Bit Field Allocation (Over 8 Bytes)

Example:

```

struct {
    char bf1 : 3;
    unsigned char bf2 : 3; /* changed sign */
    char bf3 : 3; /* does not fit in current byte */
} s2;

```

Figure 11-5 shows the bit field allocation for the preceding example.

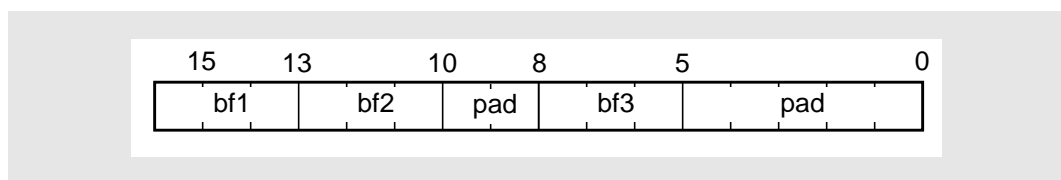


Figure 11-5. Sample Bit Field Allocation (Over 2 Bytes)

Example:

```

packed struct {
    char bf1 : 3;
    unsigned char bf2 : 3;
    char bf3 : 3;
} s3;

```

Figure 11-6 shows the bit field allocation for this example.

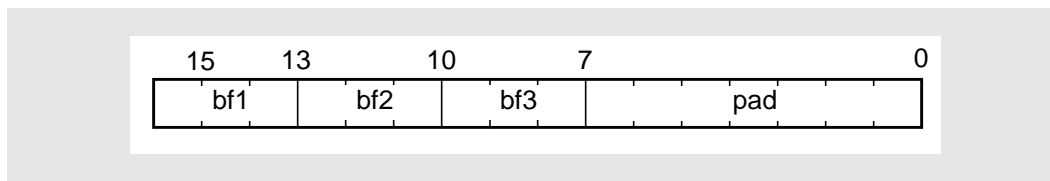


Figure 11-6. Sample Packed Struct Bit Field Allocation (Over 2 Bytes)

Alignment and Packing

This section describes the alignment and packing of complex data types.

The **-v** option diagnoses potential alignment problems and reports them as warnings. Refer to Chapter 3, *Using Command Line Options*, for more information.

Alignment of Bit Fields

When dealing with data, the most important feature of an architecture is the data bus width. Fetching the size of the data bus width is the most efficient method of accessing data, assuming information is properly aligned.

A long word that contains bit fields is aligned to a 2-byte boundary if the **-Za2** option is selected (the default for processors with a 16-bit data bus) and to a 4-byte boundary if the **-Za4** option is selected (the default for processors with a 32-bit data bus).

Figure 11-7 illustrates bit field packing.

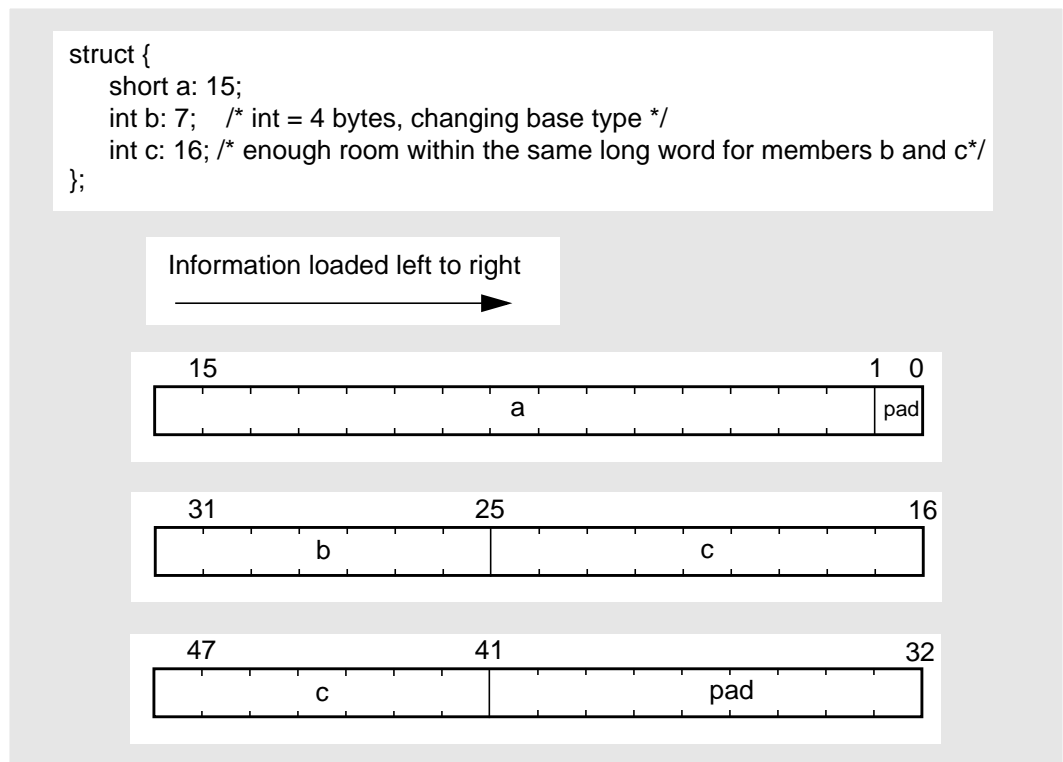


Figure 11-7. Packing Bit Fields

Figure 11-8 illustrates signed bit field packing.

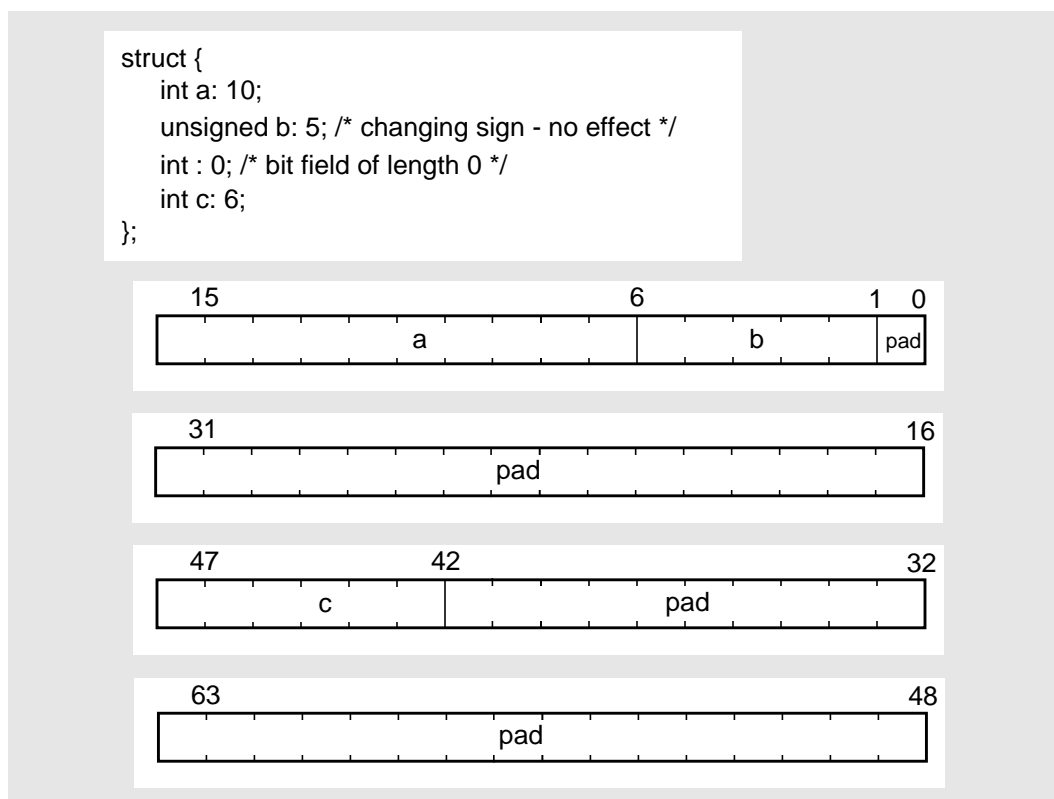


Figure 11-8. Signed Bit Fields

Aggregates

In aggregates such as arrays, structures, and unions, each member is aligned to the boundary of its natural alignment (see Table 11-5). The aggregate itself usually takes the alignment of the member with the maximum alignment size. However, when a struct or union is a member of another aggregate, its alignment cannot be less than the value of **-Zn**.

Examples:

	Processors With a 16-Bit Data Bus	Processors With a 32-Bit Data Bus
array of double	2-byte alignment	4-byte alignment
struct with long word and byte member	2-byte alignment	4-byte alignment
union with word and byte member	2-byte alignment	2-byte alignment
array of char	1-byte alignment	1-byte alignment
struct containing only chars	2-byte alignment	1-byte alignment

Size of Aggregates

The size of an aggregate should be divisible by its alignment size. Unused space after the last member is allowed, but it should not be occupied by any effective data (see Figure 11-9 and Figure 11-10).

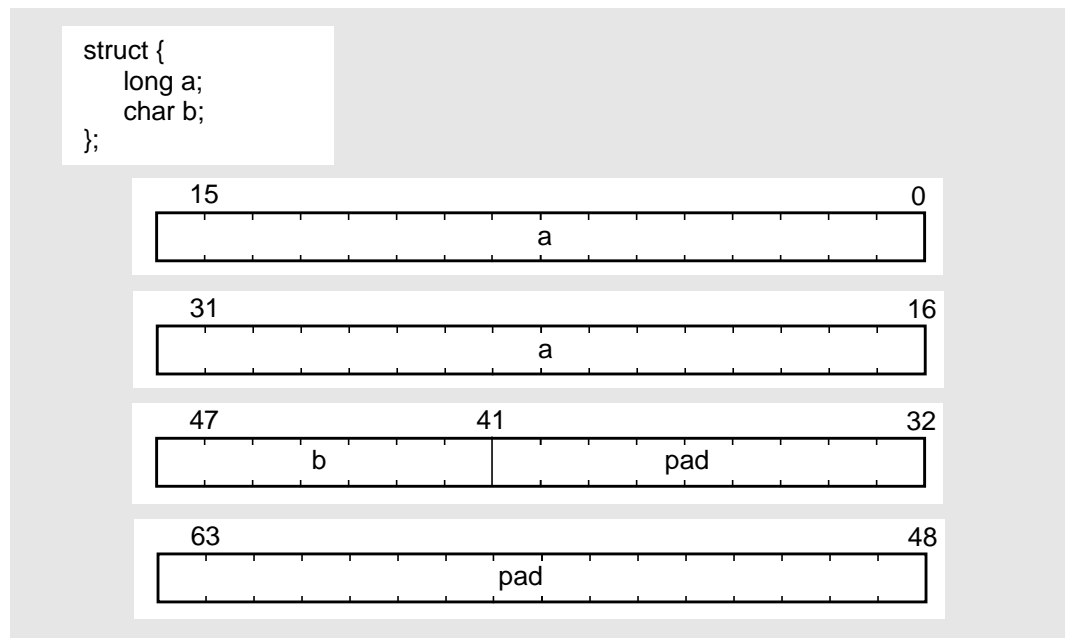


Figure 11-9. Packing of Aggregates (Big-Endian)

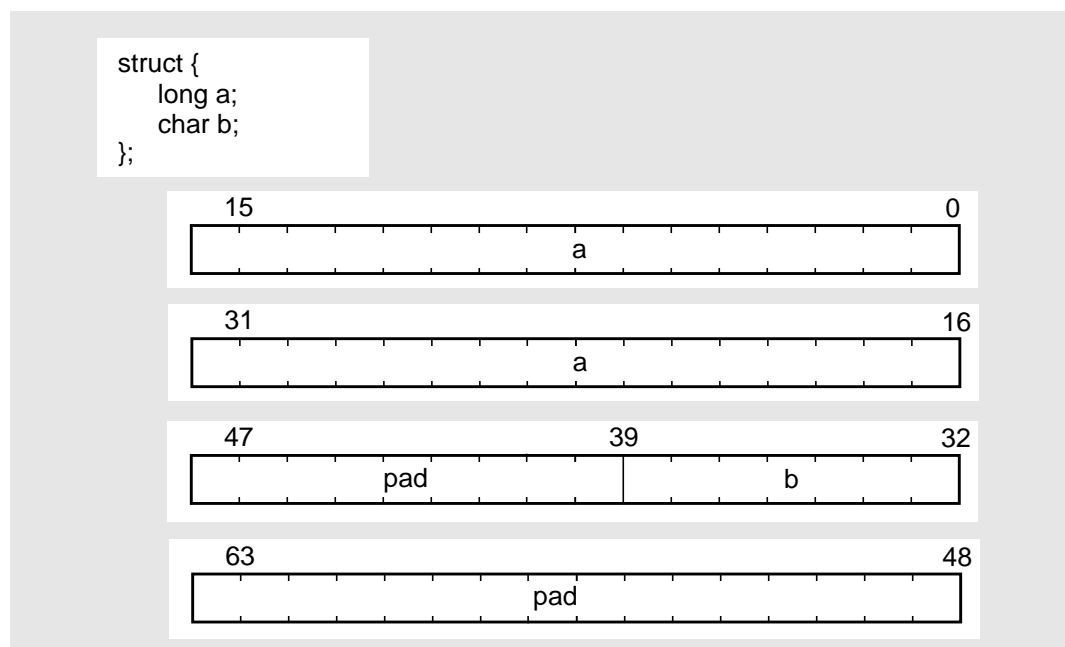


Figure 11-10. Packing of Aggregates (Little-Endian)

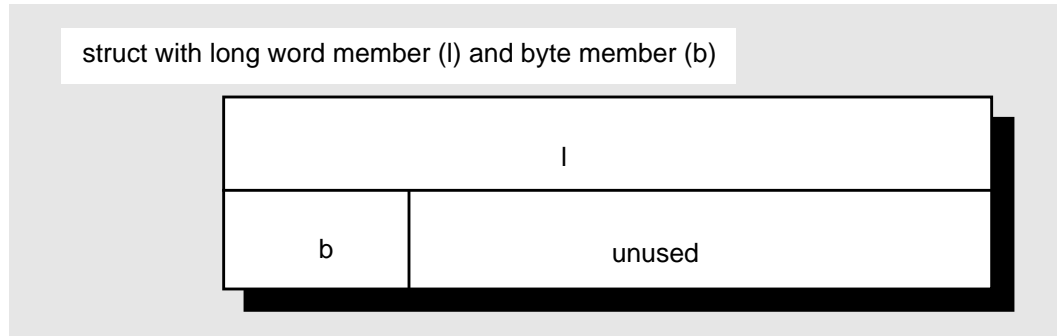


Figure 11-11. Packing of Aggregates (Big-Endian)

The **sizeof** operator applied to an aggregate always returns a number divisible by its alignment size.

Examples:

```
sizeof (struct {           /* size = 8, 4-byte alignment */
    int a;
    char c;
})
sizeof (struct {           /* size = 6, 2-byte alignment */
    short a,b;
    char c;
})
sizeof (char [3][7])      /* size = 21, 1-byte alignment */
```

Packed Structures

By default, structure members will be aligned as described in the section *Aggregates* in this chapter. The **packed** keyword can be used to force no padding between structure members.

The following example shows the allocation for a packed and unpacked structure member.

Example:

```
/* PKD will be defined to be packed or unpacked */
#define PKD packed
typedef PKD struct {
    char c1;
    int i;
    char c2;
    char c3;
} str;
```

Figure 11-12 shows the unpacked and packed structures.

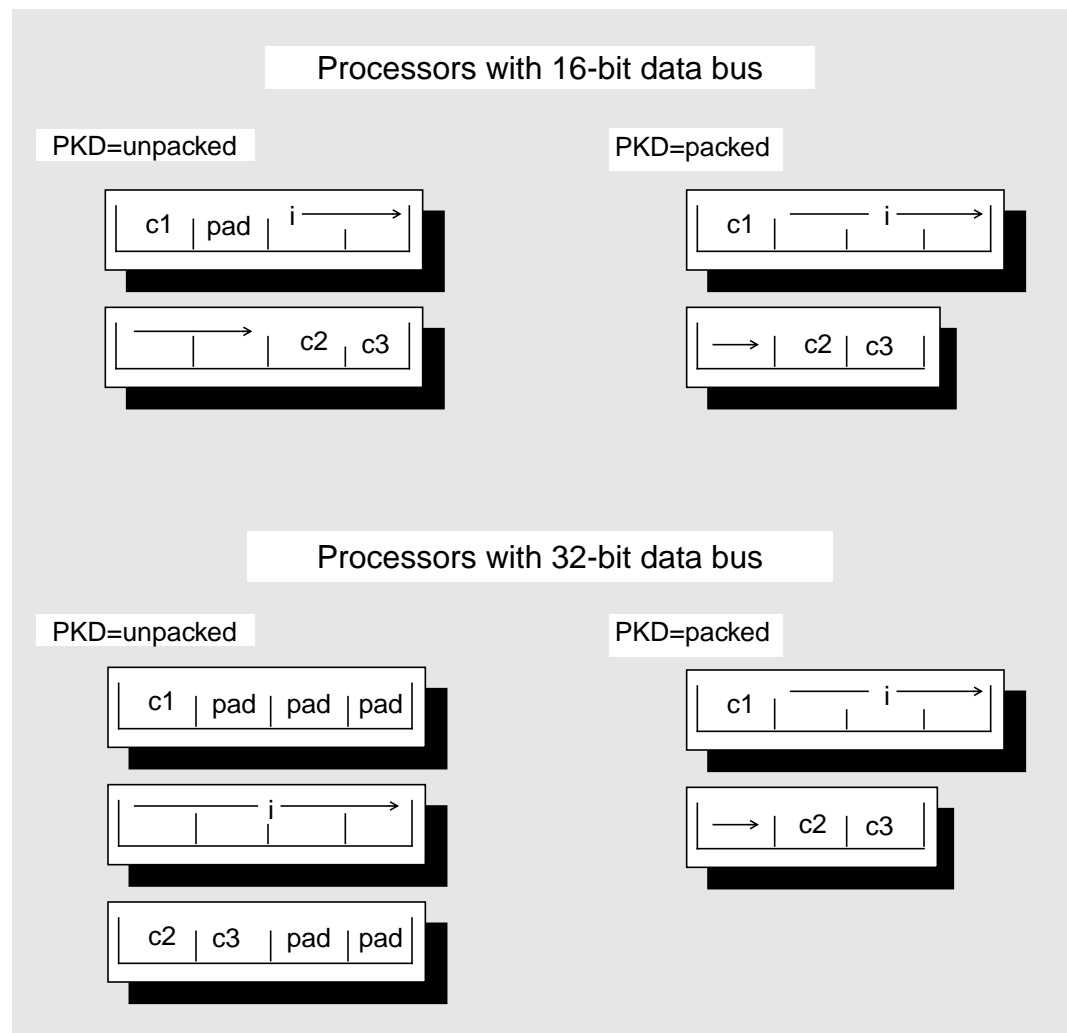


Figure 11-12. Unpacked and Packed Structures (Big-Endian)

Structure Padding

Structure layout is determined by the following factors:

- Bus width

Processors with a 16-bit data bus cannot access multibyte data on odd boundaries. Multibyte quantities can only be accessed directly if they reside at an address that is a multiple of 2. For example, an attempt to fetch a **short** (word size) at an odd address would cause an address error. In gen-

eral, this error does not occur because variables and fields of unpacked structures are aligned on correct boundaries.

In packed structures, fields are not aligned, and nonaligned fields can cause word or long word fields to start at odd addresses. In these situations, the compiler “bursts” the access to these fields by loading or storing them byte-by-byte or word-by-word.

Processors with a 32-bit data bus can usually access multibyte data on odd boundaries, which permits access to any field on any boundary (meaning that an address error never occurs). However, if the fields are not aligned to their correct boundary, the hardware uses extra cycles to access the information, which reduces performance. To prevent this, words should be aligned to addresses that are multiples of 2, and long words should be aligned to addresses that are multiples of 4.

Unpacked structures lay out their fields on the optimal boundary. Packed structures, however, pack elements as tightly as possible, ignoring boundaries. No special code is needed to access fields that do not start on their natural boundary because the hardware automatically takes care of the alignment (at the expense of performance).

- Field alignment requirements

Any field other than a bit field is aligned to its natural (required) or most efficient boundary. Bit fields are aligned according to the natural boundary of the integral type specified in their definition. Table 11-5 shows the alignment requirements of C/C++ data types.

Table 11-5. Natural Boundary Alignment

Data Type	Byte Alignment (Processors With 16-Bit Data Bus)	Byte Alignment (Processors With 32-Bit Data Bus)	Notes
char	1	1	
short	2	2	
int	2	4	
long	2	4	
pointer	2	4	
float	2	4	
double	2	4	
char : <i>n</i>	1	1	Same as char
short : <i>n</i>	2	2	Same as short
int : <i>n</i>	2	4	Same as int
long : <i>n</i>	2	4	Same as long
enum	2	4	Same as int
packed enum (1 byte)	1	1	Same as char
packed enum (2 bytes)	2	2	Same as short
packed enum (4 bytes)	2	4	Same as int
array			Same as component
union			Same as the field with the most demanding alignment requirement
struct			Same as the field with the most demanding alignment requirement
packed struct	1	1	See the -Zm compiler option

- Structure alignment requirements

Structure alignment is based on the alignment of the field with the most demanding alignment requirement. A structure whose largest alignment requirement belongs to a **long** element has the same alignment requirements as a **long**. When a struct or union is a member of another struct, its alignment cannot be less than the value of the **-Zn** option.

- Padding and trailer bytes

The compiler can add padding bytes between two fields to force the alignment of the next field to match its natural boundary. Trailer bytes can be added at the end of a structure to make its size a multiple of its alignment requirement. To avoid padding or trailing bytes, use packed structures.

- Size

The size of an unpacked structure is the sum of the size of all its fields, plus the size of all padding and trailer bytes.

Example:

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

For processors with a 16-bit data bus, `struct s` has an alignment requirement of 2 (because the largest field, `int`, has an alignment requirement of 2) and a size of 10 (1 + 1 pad + 2 + 4 + 1 + 1 trailer byte). Note that if `struct s` were packed, its size would only be 8 (1 + 2 + 4 + 1). Figure 11-13 shows the unpacked structure size.

For processors with a 32-bit data bus, `struct s` has an alignment requirement of 4 (because `int` has an alignment requirement of 4 for those processors) and a size of 12 (1 + 1 pad + 2 + 4 + 1 + 3 trailer bytes). Figure 11-13 shows the structure size.

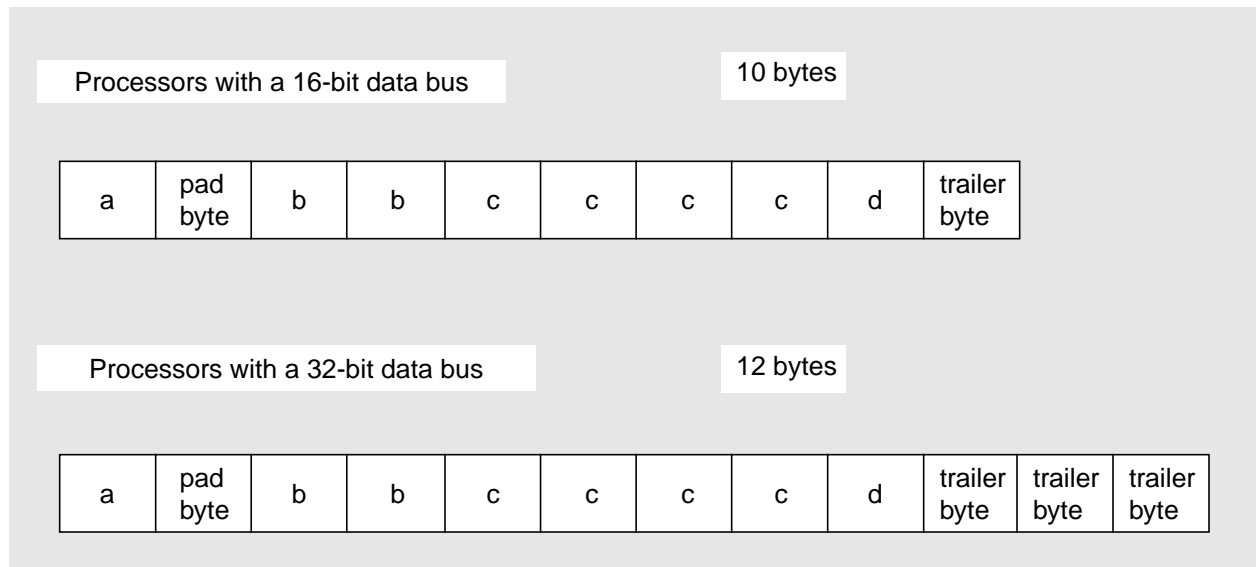


Figure 11-13. Unpacked Structure: struct s

Structure Alignment When Default Alignments Differ

When porting an application, it is important to guarantee that structure layouts are identical. Packed structures use the same alignment for all processor types. For unpacked structures, follow these guidelines:

- Use **-Za2** or **-Za4** consistently. Using **-Za2** aligns all fields of 4 bytes or greater to a multiple of 2, and can negatively affect performance on some systems. Using **-Za4** aligns all fields of 4 bytes or greater to a multiple of 4. Though this maximizes efficiency on some processors, it does not affect efficiency on others and may waste space.
- Specify the options directly inside every source file using **#pragma options**:

```
#pragma options -Za4
```

Example:

```
struct {
    struct inner {
        char x;
    } a;
    long b;
} port;
```

The size of struct port differs according to the options specified, as shown in Table 11-6 and Figure 11-14.

Table 11-6. Effect of Compiler Options on Structure Layout

UNIX/Windows Options	Size of struct port for Processors With 16-Bit Data Bus	Size of struct port for Processors With 32-Bit Data Bus
Defaults: -Za2 or -Za4	1 + 1 padding byte + 4 Total: 6 bytes	1 + 3 padding bytes + 4 Total: 8 bytes
-Za4	1 + 3 padding bytes + 4 Total: 8 bytes	1 + 3 padding bytes + 4 Total: 8 bytes

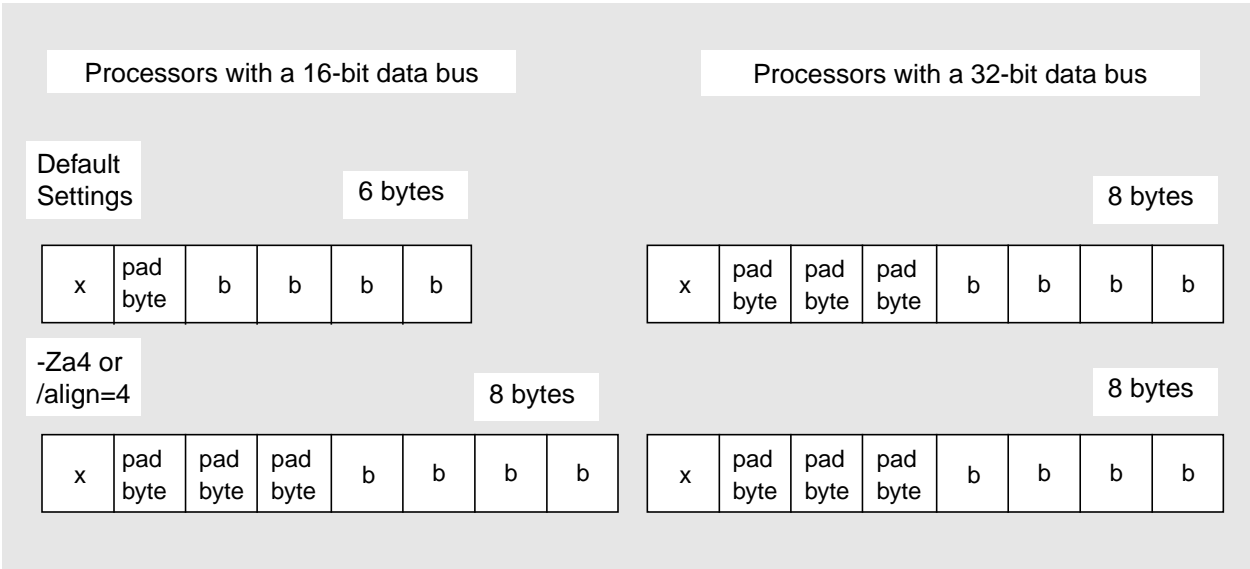


Figure 11-14. Structure Alignment Example

Warning

If all modules are not compiled with the same set of options, run-time errors can occur because the “same” structure might have different layouts in different modules.

Packed Enumeration Types

A standard unpacked enumerated data type has **signed int** as an underlying type. A packed enumerated type is instead given an underlying integral type that is chosen to minimize the amount of storage necessary to efficiently hold its values.

Examples:

	Underlying Type
<code>enum color {red, yellow, green};</code>	signed int
<code>unpacked enum color {red, yellow, green};</code>	signed int
<code>packed enum color {red, yellow, green};</code>	signed char
<code>packed enum color {red=1, yellow, green=100};</code>	signed char
<code>packed enum color {red=127, yellow, green};</code>	signed short
<code>packed enum color {red=1, yellow, green=127+1};</code>	signed short
<code>packed enum color {red=ou, yellow, purple=255};</code>	signed short
<code>packed enum color {green=MAX_UNSIGNED_INT};</code>	unsigned int
<code>packed enum color {x=32768};</code>	signed int
<code>packed enum color {x=100000U};</code>	unsigned int

Tips About Packing

Bit fields are declared by specifying an underlying type and a bit length. For example, the following declaration would describe a structure made up of 2 bits, 3 bits, 3 bits, and a **short**:

```
#pragma option -Zml
packed struct
{
    unsigned char a:2;
    unsigned char b:3;
    unsigned char c:3;
    short s;
} three_tight_bytes;
```

If this structure is not packed, the byte total is: 1 (char) + 1 byte padding (for even alignment) + 2 (short) = 4 bytes. If the structure is packed, the byte total is 1 (a+b+c=8 bits) + 2 (short) = 3 bytes.

Figure 11-15 shows the packed and unpacked structure size.

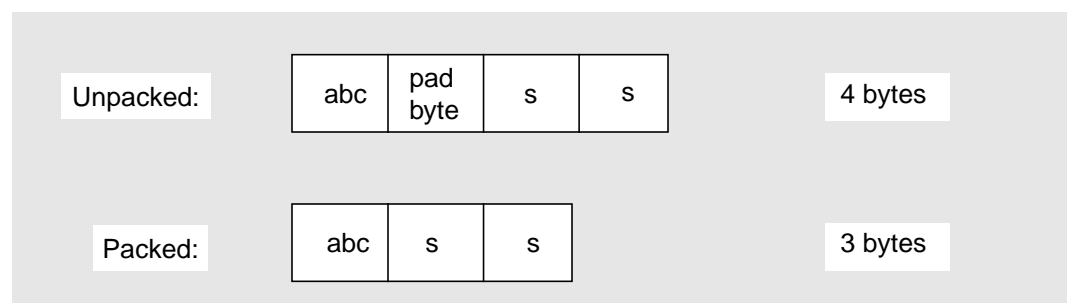


Figure 11-15. Structure three_tight_bytes

When packing a structure, remember that whenever the size of the underlying type changes, the remaining portion of the previous type is padded.

Example:

```
struct
{
    char a:2;
    short b:3;
    char c:3;
} four_pretty_loose_bytes;

packed struct
{
    char a:2;
    short b:3;
    char c:3;
} hope_one_tight_byte;
```

The unpacked structure contains 6 bytes (padding will be added to the end of the structure for word alignment). The packed structure also contains 4 bytes. Since the size of a `char` (1 byte) is not the same as a `short` (2 bytes), the remaining 6 bits of `char a` are padded.

By definition, an enumerated type is **signed int** type.

Example:

```
typedef enum {FALSE=0, TRUE=1} boolean;

packed struct
{
    unsigned char a:2;
    boolean flag1:1;
    unsigned char b:2;
    boolean flag2:1;
} str;
```

Looking at this structure, you might think it will fit into one byte. However, since an enumerated type is **signed int** and therefore has a different size than `char`, the bit fields are not combined. Thus, the structure size is 10 bytes:

1 (char) + 4 (enum size is **int**) + 1 (char) + 4 (enum).

The `packed` declaration applies only to the `struct` and does not affect the `boolean` enumeration. To pack this `struct` into a single byte, declare `boolean` as **packed**, as follows:

```
typedef packed enum {FALSE=0, TRUE=1} boolean;
```

This results in `boolean` being allocated as a `char`.

Applications such as operating systems or data transmission may have data structures that have a fixed or predefined layout. The compiler provides a number of extensions to allow processing of such structures.

For example, you can describe the processor status word data structure shown in Figure 11-16 using packed structures.

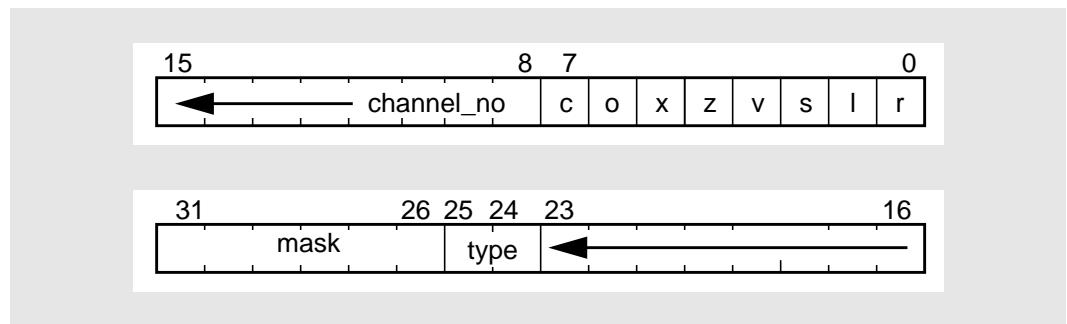


Figure 11-16. Sample Data Structure

Extensions to the Microtec compiler let you define the structure using a combination of **packed struct**, **packed enum**, and bit fields:

Example:

```
packed struct {
    packed struct {
        signed char mask:6;
        signed char type:2;
    } interrupts;
    short int channel_no;
    packed enum {c=128,o=64,x=32,z=16,v=8,s=4,l=2,r=1U} flag;
} psw = { {0x1f,0}, 1, v|x|s };

assert(sizeof(psw)==4);
```

Allocation of Variables

Memory variables are allocated based on their size and frequency of use. Scalar and pointer variables generally qualify for allocation to a register unless the variable address is taken. Floating-point variables qualify for allocation to a register if a floating-point coprocessor is available.

Register, local, and static variables are discussed in the following sections.

Register

The register storage class is used for local variables or parameters only. For each function, C/C++ allocates as many variables as possible to the hardware registers.

Local Variables

The compiler can put local (automatic) variables into registers even if a **register** declaration is not used. Eight, sixteen, and thirty-two bit variables can be allocated into registers. The lifetime of each local variable is examined by the compiler; several variables can share the same register in one routine, if possible. A variable allocated to a register always resides in that register. However, since other variables can share the register, the value of the register might not always contain the value of the variable in question.

Variables that are declared in **register** declarations are allocated to registers before nonregister variables. This allocation allows variables to be specified in their order of importance.

Static Variables

All static variables are initialized to zero automatically unless they are explicitly initialized to a value other than zero. Memory might not be allocated to static variables that are never used.

By default, uninitialized static variables are allocated in the **zerovars** section. This section is cleared at program start-up time. For options that can alter this behavior, see Chapter 3, *Using Command Line Options*.

Memory Allocation

The 68000 and ColdFire processors are both exclusively big-endian. For more information about big-endian and little-endian processors, refer to the section *Storage Layout* earlier in this chapter.

For processors with a 16-bit data bus, the stack, data items that are larger than one byte, and all complex data types are even-aligned. For processors with a 32-bit data bus, the stack, data items that are a multiple of four bytes, and all complex data types are quad-aligned. Data items that are a multiple of two bytes are even-aligned.

Packed bit fields are allocated starting at the most significant bit, and each bit field must be fully contained in no more than four bytes.

Programs that rely on byte ordering (for example, programs that declare a variable **int** in one module and **char** in another) might not be portable across CPUs.

Run-Time Organization 12

This chapter describes the memory organization of C/C++ programs for 68000 and ColdFire families microprocessor-based systems.

Code Organization

The Microtec C/C++ Compilers place each variable and function into a specific section. Table 12-1 shows the sections that are generated by the compiler.

Each section has a name and a type. The following types are defined:

CODE	For code or data that is read-only and therefore can be safely placed in read-only memory (ROM).
DATA	For all data that could be modified. Typically, this data is placed in Random-Access Memory (RAM).

There can be several differently-named sections of the same type. For example, the **code**, **strings**, **literals**, and **const** sections are all of type **CODE**, since they are all typically read-only sections.

If a variable is not initialized in any module, it is treated as a “C common.” The linker allocates “C common” and uninitialized static variables in the **zerovars** section. The **-X** option can be used to modify this behavior. For more information on this option and specifying alternate section names, refer to Chapter 3, *Using Command Line Options*

Table 12-1. Sections Generated by the Compiler

Name	Type	Contents
code	CODE	Program code
strings	CODE	String literals
literals	CODE	Compiler-generated data
const	CODE	Explicitly initialized const variables
initfini ^a	CODE	Keeps C++ instructions to initialize and finalize C++ static variables (generated only if some static variables call to constructor or destructor functions)
pixinit ^a	CODE	Keeps C++ instructions to initialize virtual function pointers for position-independent programming (generated when position-independent code or data options are used)
cxx_edt ^a	CODE	Keeps the C++ exception dispatch table, which is used when a C++ exception is thrown (generated only when C++ exception handling is used)
cxx_rtti ^a	DATA	Keeps C++ run-time type information (RTTI) (generated when C++ polymorphic classes or exception handling are used)
ioports	DATA	Simulated I/O ports used with the Debugger
vars	DATA	Explicitly initialized non- const variables
zerovars	DATA	Uninitialized variables
tags	DATA	Symbol definitions to tag the entry and exit points of each function
stack ^b	DATA	Stack pointed initialized to point to the end of the section
heap ^b	DATA	Heap pointed initialized to point to the start of the section
syshost	DATA	Resides in the RTL. Used by application to access the host resource through XRAY 4.x

a: C++ Compiler only.

b: C Compiler only.

Changing Default Addressing

The **code** section should be located in ROM. Items in the **code** section are referenced according to the address mode chosen with the appropriate **-Mc** code option.

The **zerovars** section should be located in RAM. Items in the **zerovars** section are referenced according to the address mode chosen with the appropriate **-Md** data option.

The **tags** section is located in RAM. Items in the **tags** section are referenced according to the address mode chosen with the appropriate **-Md** data option (absolute, PC-relative, or **An**-relative). Refer to the description of the **-Kt** option in this manual for information on the **tags** section.

The remaining sections (**vars**, **const**, **strings**, and **literals**) can each be located in ROM or RAM at link time. If you are generating position-independent code or data and you will be locating any of these sections in the nondefault area (ROM/RAM), you must tell the compiler how to address these sections.

Refer to the appropriate **-a** option for information on specifying alternate addressing modes for the various sections.

For more information on any of the compiler options mentioned in this section, refer to Chapter 3, *Using Command Line Options*, in this manual.

Compiler-Generated Sections

The following example illustrates how a compiler generates sections. The example uses two modules: **module1.c** and **module2.c**. These modules are compiled and assembled to produce **module1.o** and **module2.o** and are linked together to produce an executable file. Table 12-2 shows the contents of **module1.c** and **module2.c**.

Table 12-2. Example Modules

module1.c	module2.c
<pre> int a = 0; int b; static c; const ten = 10; main () { int d; func ("hello"); /* ... */ } </pre>	<pre> int a; int b; static c; func (char * string) { /* ... */ } </pre>

Table 12-3 shows the contents of each section in **module1.o** and **module2.o**. The Microtec C/C++ compilers prepend an underscore (`_`) to symbol names.

Table 12-3. Sections Contained in module1.o and module2.o

Name	Type	module1.o	module2.o
code	CODE	<code>_main</code>	<code>_func</code>
strings	CODE	<code>"hello"</code>	
literals	CODE		
const	CODE	<code>_ten</code>	
vars	DATA	<code>_a</code>	
zerovars	DATA	<code>_.S0_c</code>	<code>_.S0_c</code>

In **module1.o**, `_b` is an unresolved reference that has not been allocated to any section. The section for `_b` cannot be determined until linking; it can eventually reside in the **const**, **vars**, or **zerovars** sections, or in a different user-defined section. Similarly, in **module2.o**, `_a` and `_b` are unresolved references that have not been allocated to any section. The variables `_d` and `_string` are not allocated to any section. They are accessed through the stack at run time. The two modules are linked to create an executable file. The linker combines all sections of the same name together. Table 12-4 shows the sections contained in the resulting executable file.

Table 12-4. Sections Contained in the Executable File

Name	Type	Contents
code	CODE	<code>_main, _func</code>
strings	CODE	<code>"hello"</code>
literals	CODE	
const	CODE	<code>_ten</code>
vars	DATA	<code>_a</code>
zerovars	DATA	<code>_b, _.S0_c, _.S0_c</code>

The two static variables named `_c` remain separate and are not linked together. Since `_b` was not present in any object file section, the linker placed it into the **zerovars** section.

The following sections describe the compiler-generated sections in more detail.

code Section (Type CODE)

The **code** section contains all program code. All code is assumed to be read-only and, therefore, can be safely placed in ROM.

strings Section (Type CODE)

The **strings** section contains the contents of all string literals. If your application does not attempt to modify string literals, then the **strings** section can be safely placed in ROM.

literals Section (Type CODE)

The **literals** section contains all compiler-generated literal data. You do not have direct access to this data at the C/C++ language level. Compiler-generated literal data can always be safely placed in ROM.

const Section (Type CODE)

The **const** section is used for external or static variables that are explicitly initialized. These variables are declared using the keyword **const**.

In some cases, scalar static **const** variables might not be placed in this section. Instead, the compiler could decide not to allocate any storage for them and choose to substitute a constant value each time they are used.

If your application does not attempt to modify **const** variables, then you can safely place the **const** section in ROM. Direct assignment to **const** variables is prohibited by the compiler but can be achieved by using type casting or by using **const** inconsistently across different modules.

pixinit Section (Type CODE)

The **pixinit** section is generated when position-independent code is generated and there are compiler-generated data pointers to be initialized at run time. At start-up time, the code referred to by this section will initialize all variables that were not set properly by the linker. Refer to the *pixinit Section* in Chapter 13, *Embedded Environments*, for more information.

cxx_edt Section (Type CODE)

The **cxx_edt** section contains a read-only jump table that is used when a C++ exception is thrown.

cxx_rtti Section (Type DATA)

The **cxx_rtti** section contains run-time type information to support C++ features such as typeid, dynamic cast, and exception handling. This section needs to be initialized during startup. It can be placed in ROM only with INITDATA in the linker command file.

ioports Section (Type DATA)

The **ioports** section is generated whenever a program uses the **read()** or **write()** functions. The **ioports** section is used to hold the **_simulated_input** and **_simulated_output** variables that are used to facilitate I/O simulation with the XRAY Debugger. The **ioports** section should be placed in RAM.

vars Section (Type DATA)

The **vars** section is used for all static and external variables that are explicitly initialized. These variables are declared without the **const** keyword. Typically, this data is placed in RAM.

zerovars Section (Type DATA)

The **zerovars** section is used for all uninitialized external or static variables. These variables are initialized to zero when the program is loaded into memory or when execution starts.

If a portion of a variable is explicitly initialized, then the entire variable is considered to be initialized and is placed in the **const** or **vars** section. For example, in the following declaration:

```
char buffer [1024] = {0};
```

the first element of `buffer` is explicitly initialized, so this variable cannot be placed in the **zerovars** section.

Variables in the **zerovars** section are initialized to zero when the program is loaded into memory or when execution starts. Since the **zerovars** section contains only uninitialized variables, it does not occupy any physical space in the executable file. The allocation and initialization of **zerovars** variables depends on the presence of a program loader, as follows:

- If the operating system has a program loader, then the loader must allocate memory space for the **zerovars** data. Some program loaders initialize this memory to 0. If your loader does not provide this service, then your start-up routine should initialize this memory to 0.

- If the operating system does not have a program loader, then your program code must be permanently resident in ROM, and the **zerovars** data must be allocated to an address space within RAM. Your start-up routine should initialize the **zerovars** memory to 0 each time the program is restarted.

tags Section (Type DATA)

The **tags** section is generated when the **-Kt** compiler option is specified. This section contains symbol definitions to tag the entry and exit points of each function. These tags can be used by real-time analysis tools to monitor program execution. The **tags** section should be placed in RAM.

syshost Section (Type DATA)

The **syshost** section is not generated by the compiler, but included in the RTL. This section is used by the application to access the host resource through XRAY 4.x. This section should be placed in RAM. The **syshost** section must be initialized in order to allow data to be copied from ROM to RAM during initialization. To initialize the **syshost** section, add the following command to your linker command file:

```
INITDATA SYSHOST
```

For more information about the linker command file and the **INITDATA** command, refer to the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.

Embedded Environments 13

This chapter includes the following information to help you use the Microtec C/C++ compiler to write embedded applications:

- How to include assembler source code in your C program
- How to generate reentrant code
- How to write programs that use position-independent code and data
- How to modify the system functions and initialization routines included in the distribution
- How to reserve a register for shared data, system data, reentrant programs, or shared programs
- How to create a linker command file
- Tips on data initialization
- Tips on using the linker to initialize program variables in RAM at start-up

Considerations for Embedded Systems

You should consider the following when working with an embedded system:

- The hardware starting address must coincide with the main entry point (or code that ultimately jumps to the main entry point), so the sample initialization routine in the file **entry.c** may need to be modified. If any modifications are made, recompile **entry.c**, and link the resulting object module with your program. Either load the new version ahead of the Microtec C library in the linker command file, or replace the old version in the library. See the section *User-Modified Routines* in this chapter for more information on the initialization routine and how to modify it.
- Embedded environments typically have their own memory organization and I/O system, so you may have to customize several run-time library routines, such as the I/O routines (**read**, **write**), the memory allocation routines (**malloc**, **calloc**), and the system functions (**open**, **close**).
- Initialized variables can be placed in ROM or RAM or initialized by the program at run time. If variables are placed in ROM, they cannot be altered by the program. The *Data Initialization* section in this chapter discusses this topic in detail.

- To use the C or C++ I/O and memory allocation library, you must modify several routines to adapt them to your embedded environment. See the section *User-Modified Routines* in this chapter for more information.
- Interrupt handlers written in assembly language must save and restore all registers they use. These routines may call C or C++ functions as described in Chapter 10, *Interlanguage Calling*.

In-Line Assembly

Embedded applications require close and efficient control of the target architecture. To accommodate this need, the Microtec C/C++ compilers support assembler statement and C code intermixing.

Use the pseudofunction **asm** (or, equivalently, **ASM**) to include any number of assembler statements in C code.

Syntax:

```
asm([type[,]] [string [,string]...]);
```

Description:

<i>type</i>	Tells the compiler what is returned by the asm invocation and, implicitly, where it is returned. By default, the value returned by the asm pseudofunction is of type int .
<i>string</i>	Represents assembler instructions. Generally, each <i>string</i> separated by a comma corresponds to a new assembler line. The number of assembler instructions you can insert with a single asm statement is unlimited, but you must follow these rules: <ul style="list-style-type: none"> • Each assembler statement must be on its own line, so either use a separate string for each assembler statement or use <code>\n</code> to generate a newline character. • Include at least one white space character (tab or space) in front of any assembler mnemonic. However, labels must start in column 1, so there should not be any white space before a label.

The **asm** pseudofunction behaves like a function in that it takes parameters and returns a type, but it does not generate a procedure call. Instead, it inserts the assembler code “in-line.”

Like any C function, **asm** can be invoked with or without taking its return value into consideration. The global optimizer in the compiler disables certain optimizations based on the return value. The **asm** pseudofunction cannot return structures or unions. For other types, the compiler expects the return value to be in the same register(s) as a function of that type.

Follow these guidelines to add **asm** pseudofunctions to your code:

1. Write your C program.
2. Compile the program.
3. Look at the places in the assembler output where you are considering adding **asm** pseudofunctions.
4. Notice that the compiler adds underscores (_) in front of all variables and accesses auto variables by using **SP**.
5. Add assembly code (also known as “inserts”) to **asm()** pseudofunctions within procedures.
6. Use **#pragma asm** and **#pragma endasm** to add assembly code outside procedures.

Examples

The following are some simple examples of how to use **asm** to put assembler lines into a C program.

Example:

```
main()  
{  
    asm(" move.l $1000,SP ; initialize stack pointer");  
    ...  
}
```

In the example above, the compiler inserts the quoted string immediately after the prologue code that it generates for `main()`. A newline is added after the string to avoid concatenating it with the next line.

Since no type was specified in the above **asm** statement, the return type is **int**. The return type is ignored since the return value is not assigned to any variable.

Example:

```
asm(char*, " MOVE.L #0,D0 " ) [100]='*';
```

This example shows how to impose a `char` array view on physical memory.

Example:

```
asm( " NOP", " TRAP #0" );
asm( " NOP\n TRAP #0" );
asm( " NOP ", " T" "RAP #0 " );
/* no ', 'after T - same as " TRAP #0 " */
asm( " NOP "); asm( " TRAP #0 " );
asm(int, " NOP", " TRAP #0");
```

These statements all generate the same code:

```
NOP
TRAP #0
```

The compiler supplies an **End-of-Line** character at the end of every string argument. Thus, in the statement:

```
asm( " NOP\n", " TRAP #0" );
```

the `\n` is unnecessary.

Examples:

```
asm("label:"); /*label is in col 1 */
asm("MOVE.L #1,4(A6,D2.W); /* ERROR - MOVE in col 1 */
asm(" ORG.S $100");
asm("thous DC.L $1000,$2000,$3000 " );
asm(" garbage in ...");
```

These examples show how powerful the **asm** feature can be. Use **asm** carefully; since the compiler passes the string arguments of **asm** directly to the assembler, it does not check the syntax of the assembler statements.

Features of Assembler In-Lining

This section describes features of **asm** and provides examples.

Assigning asm to a Variable

Like any other C function, the return value of **asm** can be assigned to a variable. The following example determines the value of the stack pointer.

Example:

```
sp_reg=asm(" move.l sp,d0 ; get value of sp register");
```

The above **asm** statement returns an integer (the default type). According to the calling conventions, integers (like other scalars) are returned in register `d0`. Since

asm behaves strictly like a function, the compiler generates code to move the returned value to the target.

The compiler produces this sequence in the assembly file:

```
move.l sp,d0      ; get value of sp register
move.l d0,_sp_reg
```

Returning a Typed Value

The type argument to **asm** can be used to tell the compiler what is returned by the **asm** invocation and, implicitly, where it is returned.

Example:

```
int *sp=asm(int *, " move.l SP,D0");
```

Since the compiler knows a pointer is returned in **D0**, after the pseudo-invocation of **asm**, the compiler generates code to move **D0** to the variable **SP**.

Example:

```
double d=asm(double, " fmove FP1,FP0");
```

In this example, since floating-point values are returned in **FP0** (when **-f** or **-p68040** options are used), the value of **FP0** will be moved into **d**.

No real call to **asm** takes place. The passing of the parameters and the call itself are replaced by the string(s) that you supply. Store the values that interest you in **D0**.

Using #define for Readability

asm statements are more readable with the **#define** directive.

Example:

```
#define D4 asm(int, " move.l D4,D0")
#define D5 asm(int, " move.l D5,D0")
if (D4>D5)
...
```

This example shows that it is possible to access each machine device at the C level. The overhead is confined to a move. For **D0** (and **FP0**), this move is not required.

Variable Names Inside asm

Global and local variables can be inserted by name inside an **asm** string. At the assembly level, a global variable is represented by an underscore (**_**) prepended to

its name (for example, `_x` represents global variable `x`). To access a global variable inside an **asm** string, add an underscore.

Local variables are represented at the assembler level by frame offsets or by registers that can change between compiler releases. Their addresses can even change between two compiles if new code or variables have been added to a procedure.

The Microtec C/C++ compilers let you insert variable names in any **asm** string by quoting them with a back quote (```). Since it is unlikely that this character will appear in any assembler statement, it has been chosen as the default.

Example:

```
foo()
{
    int automobile, garage;
    asm(" move.l `automobile`,`garage`");
    . . .
}
```

The actual addresses of the inserted variables will be supplied while conforming to the usual scope rules. According to the memory/register binding in force for the compilation, the above example will generate:

```
move.l 12(a6),d4
```

To use a character other than the back quote, use the **-uichar** option to change the insert character to *char*. Use the **-uichar** option inside a **#pragma option** rather than from the invocation line so that the insert character does not change between compilations.

Inserts are recommended for global and local variables. Variable names inside a string are not affected by compiler options that modify naming conventions, like **-upd**, unless they are enclosed in back quotes. However, inserts are option-sensitive.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l _global,_global2");
```

The **-upd** option cannot affect the global names inside the string.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
. . .
asm(" move.l `global`,`global2`");
```

In this example, the inserts reflect the `-upd` option.

#pragma asm or asm

Since **asm** is a pseudofunction, it can only be used where a normal function can be invoked: inside a procedure. Outside a procedure, any number of unquoted assembler instructions can be inserted between **#pragma asm** and **#pragma endasm**. Use these directives to create your own assembly procedure.

Example:

```
#pragma asm
#if _FPU
    fmove.l    ROUNDING,FPCR ; set rounding mode
#endif
#if _CHAR_SIGNED
    #if (_68020 || _68030 || _68040 || _CPU32)
        extb.l  D1
        move.l  D1,D0
    #else
        ext.w   D1          ; sign extend
        ext.l   D1
        move.l  D1,D0
        #define XXX signed.s
    #endif
#else
    moveq      #0,D0        ; zero extend
    move.b     D1,D0
    #define XXX unsigned.s
#endif
#include XXX
#pragma endasm
```

You can use user-defined macros as well as predefined macros inside **#pragma asm/ #pragma endasm** pairs, since full preprocessor functionality is available. Assembler inserts (**asm** function) cannot be used.

Specify the size of a **#pragma asm** and **#pragma endasm** procedure block with the **#pragma option** preprocessor directive and the **-Zinumber** command line option.

Example:

```
#pragma option -Zi10
#pragma asm
. . .
#pragma endasm
#pragma option -Zi
```

In this example, `#pragma option -Zi10` sets the size of the `#pragma asm` and `#pragma endasm` procedure block to 10 bytes. The line `#pragma option -Zi` resets the size for subsequent **#pragma asm** and **#pragma endasm** blocks and **asm()** pseudofunction invocations to the default (infinite size). For more information on the **-Zinumber** compiler option, refer to Chapter 3, *Using Command Line Options*.

Considerations for Assembler In-Lining

String contents are passed to the assembler as is. No control is placed on the string. Direct consequences of this are:

- You must be ready to read errors issued by the assembler.
- The compiler pessimistically assumes that every jump crossing the inserted instructions might be long. Consider the following example:

```
asm( " DS $1000" )
```

- **asm** is not portable. However, it is likely that programs which use direct machine instructions are not usually meant to be portable.

Example:

```
printf( "%x",asm() );
```

This statement prints the contents of register **D0**. The data printed may be completely different if the program is compiled on another compiler.

- By default, the compiler uses the assembler command line flag **noabspcadd**. However, when PC-relative addressing is specified, the compiler uses **abspcadd**. This means that an absolute expression in conjunction with the mnemonic **PC** (for example, **5(PC)**) refers to an absolute address accessed through PC-relative mode rather than to a relative displacement to the current program counter. **asm** strings that use this addressing mode are assembled differently depending on the compiler options specified.

When you use **asm**, check the effects of optimizations. Even though some optimizations are disabled, the Microtec C/C++ compiler can rearrange, change, and delete code. This behavior can affect your program.

Addressing

A variety of methods exist for accessing addresses.

Direct Memory Addressing

You can examine and modify absolute memory locations by defining a macro that functions exactly like a normal C or C++ variable, except that it is located at a particular memory address given by a number. A macro called **memloc** is defined in the following example. This macro represents the contents of the byte at location 1A (hexadecimal). The macro definition is:

```
#define memloc (*(char *)0x1A)
```

Whenever **memloc** appears in the program, it is replaced with the macro definition text `*(char *)0x1A`, which means “1A hexadecimal, treated as a pointer to a character and dereferenced.” If **memloc** appears in an expression or on the right-hand side of an assignment, it represents the contents of byte location 0x1A. If **memloc** appears on the left-hand side of an assignment, it represents the byte address 0x1A.

The macro **memloc** can be used like an ordinary byte variable:

```
chrl = memloc + 1;  
memloc = 'a';
```

The macro **memloc** can also correspond to a memory-mapped I/O location, but it must be **volatile** since each use has the side effect of reading or writing:

```
#define memloc (*(volatile char *)0x1A)
```

Calling a Function at an Absolute Address

This section describes how a C or C++ program can call an independently compiled and linked function that is placed at an absolute address without using assembly language. Use this technique when an application program needs to call a system routine at a fixed ROM address or if you are concerned about porting your program to a different processor.

To call a function at an absolute location, define a macro that declares a function.

Example:

```
#define ABS_FUNC (*(int (*)())0x124)
```

The macro defines `ABS_FUNC` to be an integer-valued function at location `0x124`. `ABS_FUNC` can now be called like an ordinary function:

```
i = ABS_FUNC(); /* ABS_FUNC can be in any expression */
```

If parameters are being passed, the routine you are calling must have a compatible calling sequence. The return value for integer functions is placed in register **D0**. For more information on function parameter passing and return values, see Chapter 10, *Interlanguage Calling*.

Reentrant Code

A routine is reentrant if it can be interrupted during its execution and reinvoked by subsequent calls (for example, from an interrupt service routine) any number of times. Reentrant routines also function correctly even if multiple threads of computation are executing a routine simultaneously. After each call completes its invocation, the prior invocation resumes processing without losing data.

The code generated by the compiler is reentrant, provided you follow these rules:

- Do not write to global variables.
- Do not write to local static variables.
- Do not perform any noninterruptible tasks (such as certain I/O operations).
- Use the non-reentrant library routines only in the manner described below.

A thread of computation (or “thread”) is a given sequence of instructions that can be interrupted by an external interrupt, and can give control to another thread at almost any time. Multiple threads may access the same executable code.

Interrupt service routines, when they are interrupted and reentered by another interrupt, constitute multiple threads of computation. In a multiprocessing or multitasking environment, each process or task is a thread of computation.

In general, the I/O routines that are declared in **stdio.h** are non-reentrant because they modify elements of the array `_iob`, which is declared in **stdio.h**. However, the three string I/O functions **sprintf**, **vsprintf**, and **sscanf** are reentrant. To make file I/O reentrant, each task should allocate its own file structure buffer on its stack instead of calling **fopen**. The following code can be used to write formatted output:

```
i = sprintf(buff, format_string, arguments);  
write(1, buff, i);
```

Description:

<code>sprintf</code>	Returns the number of characters written.
----------------------	---

<i>buff</i>	Is an area of memory where the formatted output is temporarily written.
<i>format_string</i>	Is a printf format string.
<i>arguments</i>	Is a list of arguments used by <i>format_string</i> .
<i>write</i>	Outputs <i>i</i> bytes from <i>buff</i> to the output port.
1	Causes <i>write</i> to write to stdout .

If each thread performs I/O through a unique **FILE *** or “stream,” the I/O routines are considered reentrant, because each thread modifies its own unique element of the **_iob** array. However, because **fopen** must scan the **_iob** array to find an unopened element, each stream must be opened in the application’s start-up code before individual threads are activated.

Functions that modify **errno** are also non-reentrant. Many of the functions declared in **math.h** modify this variable, along with the functions **strtod**, **strtol**, **strtoul**, and others. If a multi-threaded environment is required, avoid using or modifying **errno** in any user-written code.

Non-reentrant functions can be used in a multi-threaded environment, but only one thread may call any group of functions accessing the same static variable. For example, only one thread may call the functions **malloc**, **calloc**, **zalloc**, **realloc**, and **free**. These routines can be called by multiple threads only by using one of the techniques listed in the following sections.

Calling Non-Reentrant Functions from Multiple Threads

If multiple threads must call non-reentrant library functions during a context switch, save the variables in question. When the thread resumes, the saved data is restored. This approach requires a memory storage area for each thread, although an interrupt service routine can save and restore the data from the stack. Also, if a large amount of data must be stored, the save and/or restore operation may take too much time.

Multitasking/Multi-Threaded Environments

There are two common types of applications that use multitasking or multi-threaded execution. There are also applications that have aspects of both styles.

- A single set of code serves a number of different tasks or execution threads. (for example, a communications controller that uses the same code to service a number of communication ports)

- Different code for each of the tasks. (for example, a laser printer, where one task manages communication and another task manages the print engine)

A multi-threaded application frequently accesses memory that is shared by all of the threads as well as memory that is used only by one specific thread (thread-local storage). Often, an operating system allocates the thread-local and stack memory. Access shared variables with care, since if one thread updates the shared variable while another is reading it, race conditions and bugs may occur. Control these accesses with semaphores or monitors.

The **-Mdn** option makes all memory access relative to **An**. This is used for all thread-local storage access. When the thread is created, **A5** is loaded with the address of a block of memory that contains the thread-local memory. The functions that make up the thread are compiled with this option. The **-Khreg** option prevents the compiler from generating code that uses the specified register. Compile files that contain functions shared by all threads with the **-KhA5** option.

Data in the thread-local sections is addressed as an offset from **A5**. The linker allocates these sections at location zero in memory, even though the actual location is somewhere else. When the thread is created, the linker allocates a block of memory large enough to hold the **vars** and **zerovars** sections of thread-local memory.

Keep thread-local data separate from shared data. Ensure that the two types of data sections do not have the same names. Use the **-NZ** and **-NI** options to specify unique names for the **vars** and **zerovars** sections.

Avoid creating a **vars** section by not using static data. Otherwise, the **tvars** section is initialized when the thread is created. The **INITDATA** section contains a copy of the initialized values for the **initvars** sections that are specified in the linker command **initdata**. The linker can create initialization data for global and thread-local data, but these are not identified in the **INITDATA** section with the target section names. Do not use the **initcopy** routine, since it copies values over the “virtual” location of the **vars** section instead of the actual location. Use the **initcopy** routine if you have only thread-local data and do not want to initialize any global data. Most applications initialize variables in executable code when the thread starts.

Global variables can be accessed in two ways. The preferred way is to create shared functions that can obtain or modify the values of global variables. These routines may contain the semaphore code necessary to ensure that the value is consistent. The other method is to have a shared function return the address of the global variable to the thread. Use this method if the variable is only read by the thread.

Two examples are provided to demonstrate how to build a multitasking/multi-threaded application.

First Approach

Follow these steps to build a multi-threaded application where one set of functions is shared by several threads:

1. Compile all shared user-written routines with **A5**-relative addressing. Avoid using static data in these routines, but if used it must be uninitialized. The **vars** section must be empty; the **INITDATA** linker command will not initialize this data correctly. Use these options:

```
-Md5 -Xp
```

2. Compile user modules not shared between threads with absolute addressing. The **A5** register is reserved and the **vars** and **zerovars** sections are renamed. Unless a variable is intended to be used in inter-thread communication, there should be no common static variable names between threads. Do not use weak externals in these modules. Use the following options:

```
-Kha5 -Xp -NZuserzero -NIuservars
```

3. Modify the code in **entry.c** to dynamically allocate memory for each thread for the **zerovars** section created by the libraries and any user data created in Step 1. If a thread requires stack space, it must be allocated at this time also. Set nonstack memory to zero at start-up time.

The sample code in **entry.c** for reentrant library support shows how memory might be allocated for one thread.

4. If static data from a library function or user-written shared function is to be accessed by a function compiled with absolute addressing, use one of the following techniques:
 - Using **A5**-relative addressing, compile a function that returns the address of the static data. Call this function from the function compiled with absolute addressing to determine the address of the data.
 - Use a macro that inserts assembly code using **A5**-relative addressing. A macro to access **errno** might be written as follows:

```
#define errno (*(ASM (" xref `errno`"),\
ASM (" lea (`errno`).w(a5),a0"),\
ASM (int *, " move.l a0,d0")))
```

5. Do not use the macro version of any I/O routine. Use the function version of an I/O routine by placing the function name in parentheses.

6. If the **INITDATA** feature of the linker is to be disabled, set the command line option **-D_EXCLUDE_INITDATA** to prevent **csys.c** from initializing user static data.
7. When linking, do not use the default linker command file because the wrong library will be linked. Use the **LOAD** command to load the correct **A5**-relative addressing library. Use the **-e** option to specify the linker command file. The following is a good starting point for writing a linker command file:

```

listabs      publics,internals
listmap      publics
format       ieee
extern       ENTRY
index        ?A5,zerovars,$8000
              ; entry.c will point A5
              ; to section zerovars

initdata     uservars
sectsize     heap=$8000
sectsize     stack=$1000
sect         zerovars=0
sect         literals=$10000
order        zerovars
order        literals,strings,const,initfini,code    ; ROM
order        ??INITDATA                             ; ROM
order        userzero,uservars,ioports,heap,stack    ; RAM
load         /usr/mri/lib/mcc68ka5XXX.lib            ; library

```

In this example, **zerovars**, used by the thread, is allocated at location zero. The actual location is wherever memory is allocated when the thread is initialized. Since there may be several sections allocated at location zero, the linker may issue warning messages. Verify that these messages are the result of overlapped thread-local data sections.

8. At run time, when a context switch occurs, the **A5** register is set to address the data space of the thread that is about to run.
9. The routines **_sbrk (s_sbrk.c)** and **_chkstk (s_chkstk.c)** perform run-time checks for heap and stack overflows. These routines work only when the stack is located adjacent to the heap and is at a higher address. When this arrangement does not occur, do not use the **-Ks** command line option, and modify **_sbrk** to change the overflow checking algorithm.

Figure 13-1 shows the memory configuration used by this technique.

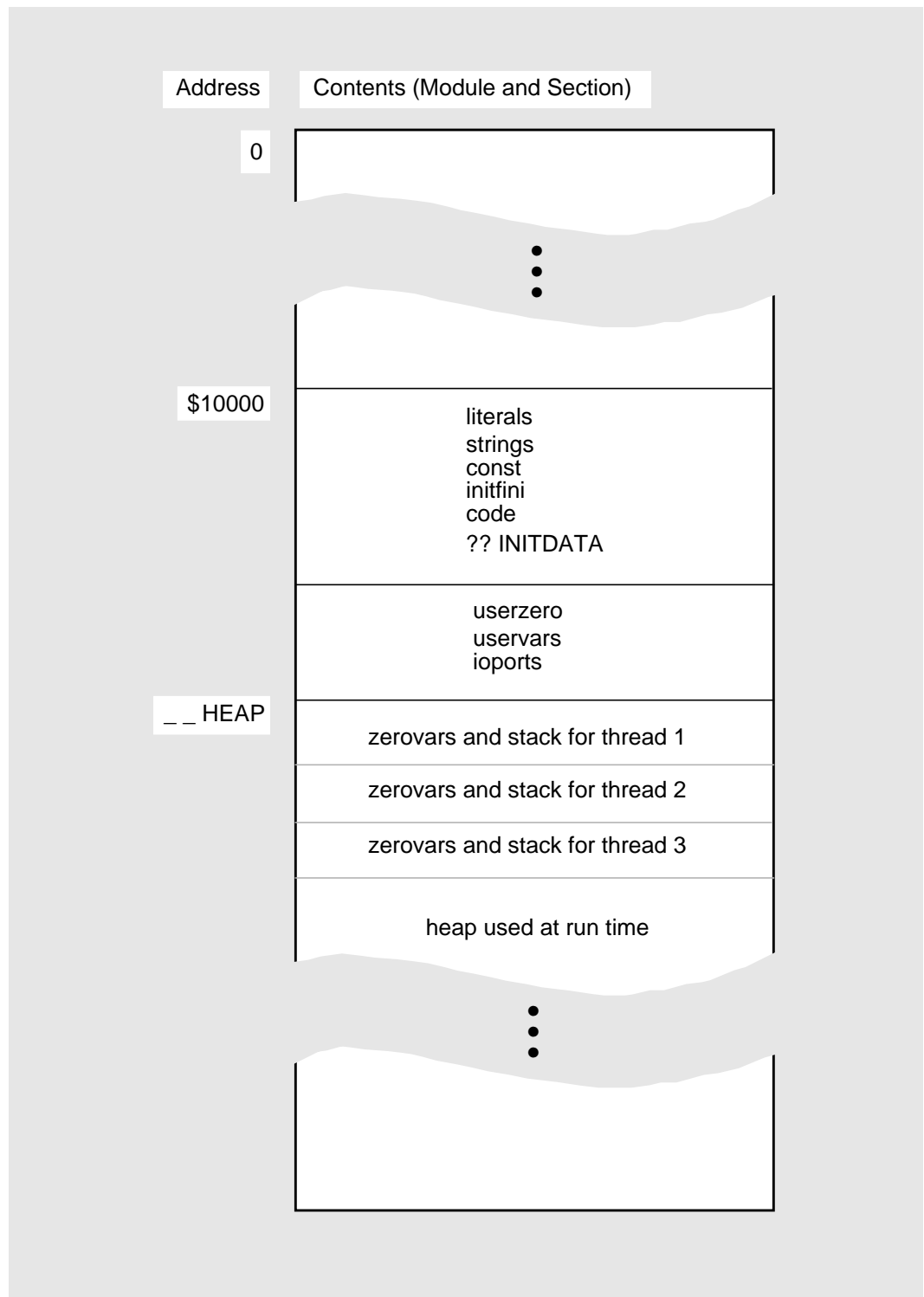


Figure 13-1. Memory Configuration for Multi-Threaded Environments (Approach 1)

Second Approach

Follow these steps to build an application with different code for each thread:

1. Compile the modules with **A5**-relative addressing and do not define any initialized data in them. The **initvars** section must be empty for the modules. Rename the **zerovars** section to a name unique to each thread and be sure there are no common static variable names between threads. Do not use weak externals. Use the following compiler options:

```
-Md5 -Mcp -NZname -Xp
```

2. Modify the code in **entry.c** to dynamically allocate memory for each thread for the **zerovars** section and the user data sections created in Step 1. If a thread requires stack space, allocate it at this time. Set nonstack memory to zero at start-up.
3. Link to the appropriate **A5**-relative addressing library. Set the starting point address of each data section created in Step 1 to the same value. Disregard the error messages the linker generates when multiple sections are placed in the same memory space. Here is one way to begin a linker command file:

```
listabs      publics,internals
listmap      publics
format       ieee
extern       ENTRY
index        ?A5,zerovars,$8000
                                ; entry.c must set A5
                                ; to section zerovars.

sectsize     heap=$8000
sectsize     stack=$1000
sect zerovars=0          ; position zerovars sect.
sect task1zero=554       ; position data for each
sect task2zero=554       ; thread at the same memory
sect task3zero=554       ; location.
sect literals=$10000
    order literals,strings,const,initfini,code    ; ROM.
    order ioports,heap,stack                      ; RAM.
load /usr/mri/lib/mcc68ka5XXX.lib                ; library.
```

4. At run time, when a context-switch occurs, the **A5** register is set to address the data space of the thread which is about to run.
5. The routines **_sbrk** (**s_sbrk.c**) and **_chkstk** (**s_chkstk.c**) perform run-time checks for heap and stack overflows. These routines work only when the stack is located adjacent to the heap and is at a higher address. If this is not the case, do not use the **-Ks** command line option, and modify **_sbrk** to change the overflow checking algorithm.

Figure 13-2 shows the memory configuration used for this technique.

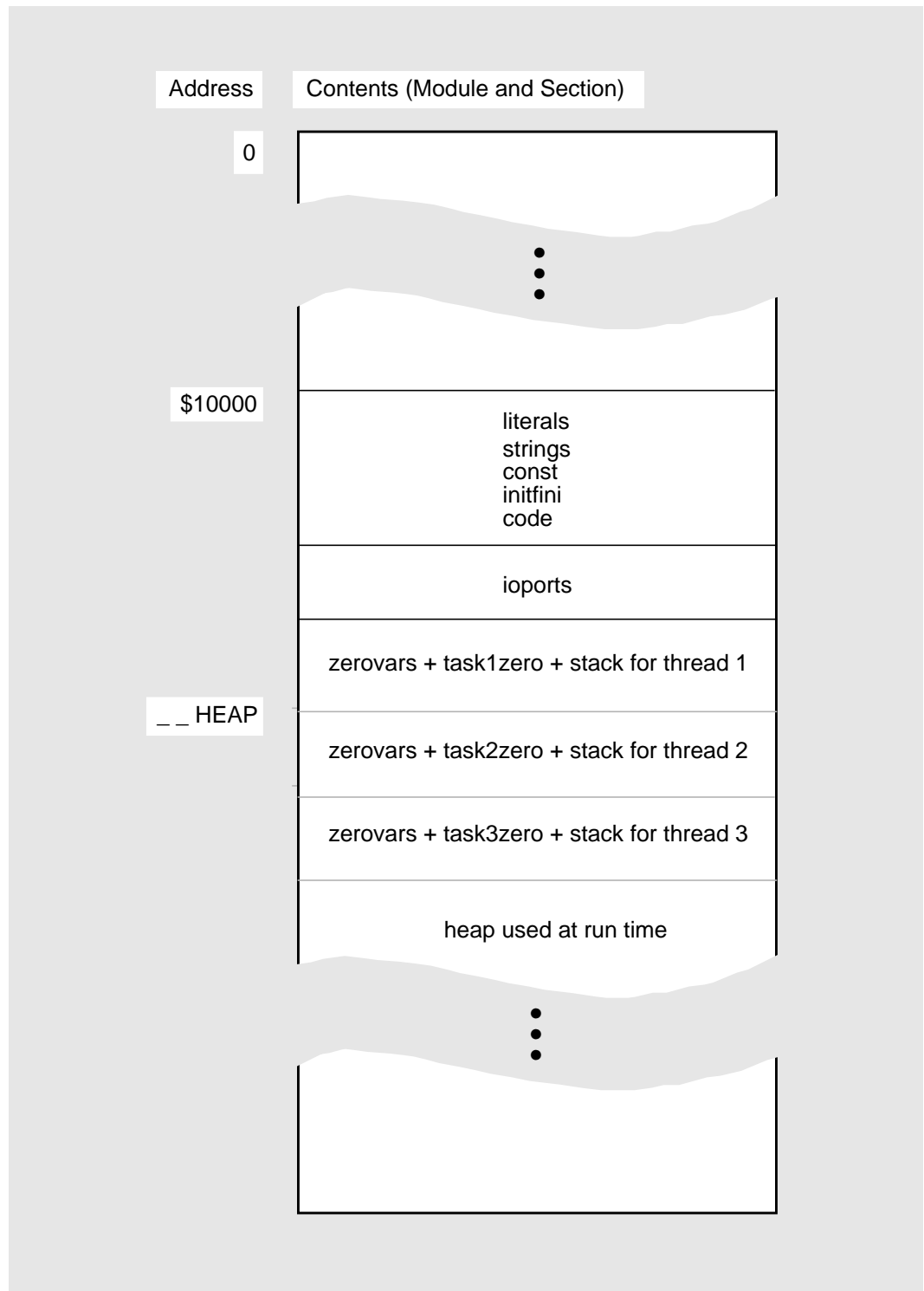


Figure 13-2. Memory Configuration for Multi-Threaded Environments (Approach 2)

Position-Independent Code and Data

This section describes position-independent code and data, and how they relate to the Microtec C/C++ compiler, the C and C++ programming languages, and the 68000 and ColdFire families of microprocessors.

By default, the compiler generates references to absolute code and data. However, the command line options shown in Table 13-1 let the compiler generate references to position-independent code and data.

Table 13-1. Position-Independent Code Options

Option	Meaning
-Mcp	Generates position-independent code by using program counter-(PC) relative addressing for all references to code
-Mdp	Generates position-independent data by using program counter-(PC) relative addressing for all references to global data
-Mdn	Generates position-independent data by using An -relative addressing for all references to global data (An is A2 , A3 , A4 , A5 , or A6)

Position-Independent Versus Position-Dependent

Position-independent code and data references are used for multitasking systems and other programs that are loaded at different addresses for different executions. A program that is position-independent can be loaded and run by the operating system at any available memory location.

This position-independent flexibility may not be necessary for a stand-alone program targeted for a specific CPU board and memory configuration because it is always loaded at the same memory location.

A position-independent program specifies its code and data addresses using register-relative addressing. The **-Mdp** option uses the program counter as the base register. The **-Mdn** option uses **A2**, **A3**, **A4**, **A5**, or **A6** as the base register.

The effective address is generated by adding a displacement (or offset) to the base. The base displacement is a two- or four-byte value that immediately follows the operation code in the instruction. This procedure for generating the effective address works because the relative positions of instructions and data in a program remain the same even though their absolute addresses can change each time the program is loaded.

Compiler Considerations

The compiler follows these rules when generating position-independent code references with **-Mcp**:

- The compiler does not use immediate addressing to load address constants.
- To load the address of a function, the compiler uses the PC-relative form of the Load Effective Address (**LEA**) instruction.
- To push the address of a function, the compiler uses the PC-relative form of the Push Effective Address (**PEA**) instruction.
- The compiler uses PC-relative addressing for all jumps and internal subroutine or function calls.

The compiler follows these rules when generating position-independent data references with **-Mdp** and **-Mdn**:

- The compiler does not use immediate addressing to load address constants.
- To load the address of a global data item, the compiler uses the register-relative form of the Load Effective Address (**LEA**) instruction.
- To push the address of a global data item, the compiler uses the register-relative form of the Push Effective Address (**PEA**) instruction.
- The compiler uses register-relative addressing for all references to static or global variables.

Note

Local automatic variables are already position-independent because they are located on the stack and are addressed relative to the stack pointer.

Programmer Considerations

To generate position-independent code, follow these rules:

- Do not call or jump to absolute code addresses within the program.
- Do not use pointers initialized to absolute code addresses within the program.

- Do not position a data reference and its target far apart. The target of a data reference must be in the range of -32768 to 32767 bytes (or -2147483648 to 2147483647 bytes if you use the **-Ml** option). For more information, see *Generating Position-Independent Code and Data* in Chapter 3, *Using Command Line Options*.

For example, a call at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but a call at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

The subroutine call at location 40000 (see Figure 13-3) cannot be made position-independent because it tries to reference an address outside the range of 7234 to 72769. The range is calculated using the following equations:

$$40002 - 32768 = 7234$$

$$40002 + 32767 = 72769$$

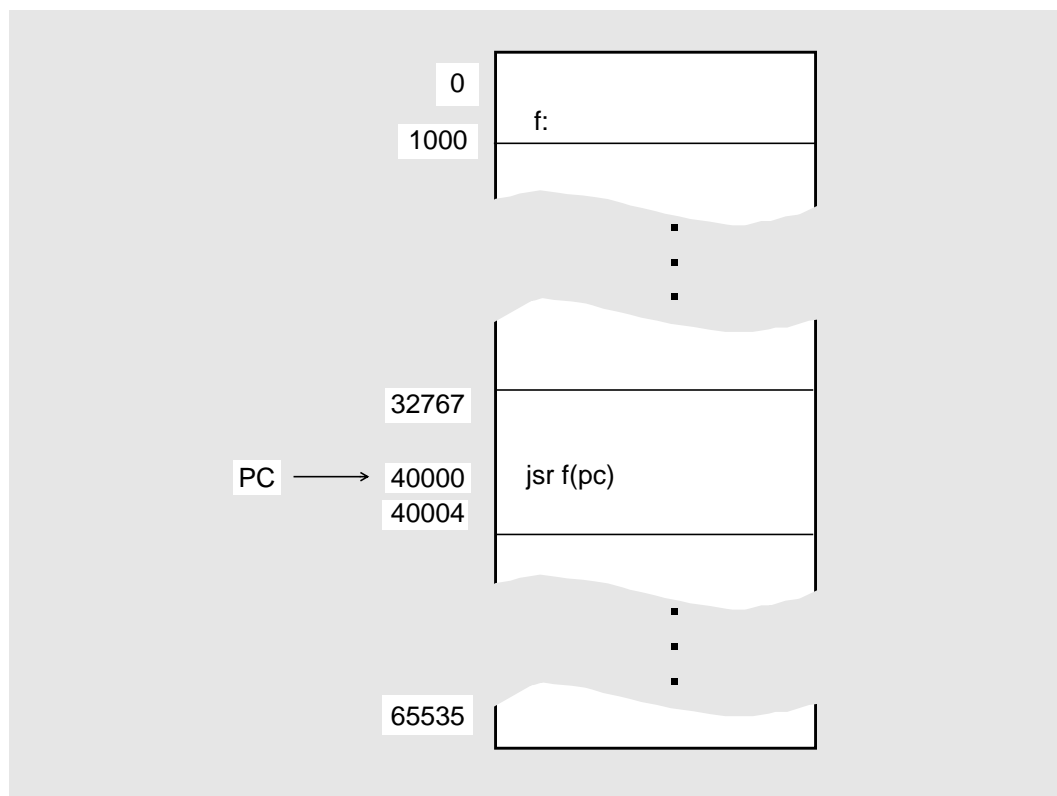


Figure 13-3. Position-Independent Code (Example 1)

To generate position-independent data, you must adhere to the following rules:

- Do not make data references to absolute data addresses within the program. The following code cannot use register-relative data addressing because it uses an absolute data address:

```
/* reference to absolute data address */
read_from(0x02); /* read from data address 0x02 */
```

- Do not use pointers initialized to absolute data addresses within the program. The following code fragment cannot use register-relative data addressing because it initializes pointers to absolute data addresses:

```
/* pointers initialized to absolute addresses of
strings */
char *table[3] = {"table", "of", "strings"};

/* pointer initialized to absolute address of data */
char c;
char *p_c = &c;
```

- Do not position a data reference and its target far apart. The target of a data reference must be in the range of -32768 to 32767 bytes (or -2147483648 to 2147483647 bytes if you use the **-Ml** option). For more information, see *Generating Position-Independent Code and Data* in Chapter 3, *Using Command Line Options*.

For example, an instruction at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but an instruction at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

The instruction at location 40000 (see Figure 13-4) cannot be made position-independent because it tries to reference an address outside the range of 7234 to 72769. The range is calculated using the following equations:

$$40002 - 32768 = 7234$$

$$40002 + 32767 = 72769$$

If you are using *An*-relative data, the default linker command file must be modified.

- The default linker command file contains the following:

```
; if using A5-relative data addressing,
; enable the next 2 lines:
;
; index ?a5, vars ; A5 will be pointed to sect vars
; load mcc68ka5XX.lib
```

Follow these steps to make your own command file:

1. Copy the default linker command file to your directory.
2. Uncomment the `index` and `load` commands.
3. Specify the correct library and path (see Chapter 2, *Using the Compilers*, for the location of the libraries on your host).
4. Invoke **mcc68k** with the **-e** option (see *Pass Command File to Linker* in Chapter 3, *Using Command Line Options*).

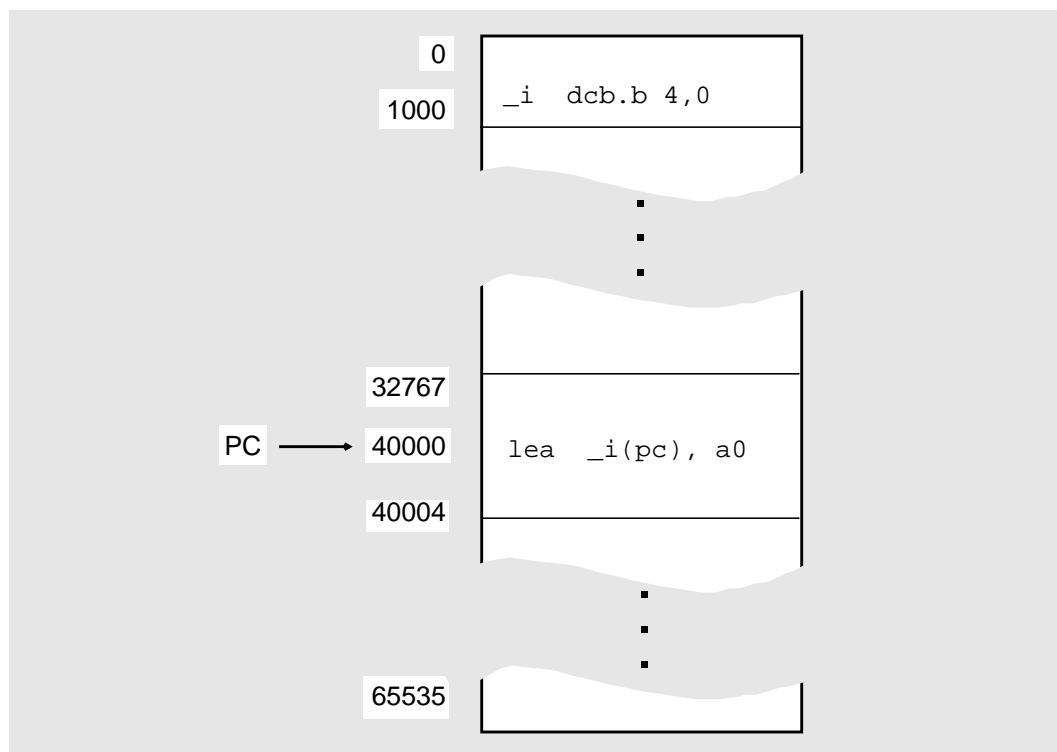


Figure 13-4. Position-Independent Data (Example 2)

Assembler and Linker Considerations

The assembler and linker automatically handle position-independent code, so no special line options need to be specified. When code is made position-independent, the run-time linker adds the address assigned to the first instruction to all the load addresses in the executable, allowing the program to execute at these new locations.

Absolute Versus Register-Relative Addressing

The following program fragment example shows the different assembly language output generated by the compiler depending on the specified options.

```
int data_1; /* data_1 is a global variable */
main()
{
    function_1();
}
function_1()
{
    data_1 = 3;
}
```

The code generated (see Table 13-2) is position-dependent when absolute addressing is used, and is position-independent when register-relative addressing is used.

Table 13-2. Absolute Versus Register-Relative Addressing

Position-Dependent Code (CODE=ABS and DATA=ABS)	Position-Independent Code (CODE=PC and DATA=PC)
...	...
L1: jsr _function_1	L1: jsr _function_1(pc)
...	...
_function_1:	_function_1:
...	...
L2: moveq #3, d0	L2: lea _data_1(pc), a0
move.l d0, _data_1	move.l #3, (a0)
...	...

Absolute Addressing

Figure 13-5 shows absolute addressing for code and data. The **JSR** (jump to subroutine) instruction uses an absolute code address, and the **MOVE.L** (move long) instruction uses an absolute data address. The absolute addresses immediately follow the operation codes in memory, and are also used as effective addresses.

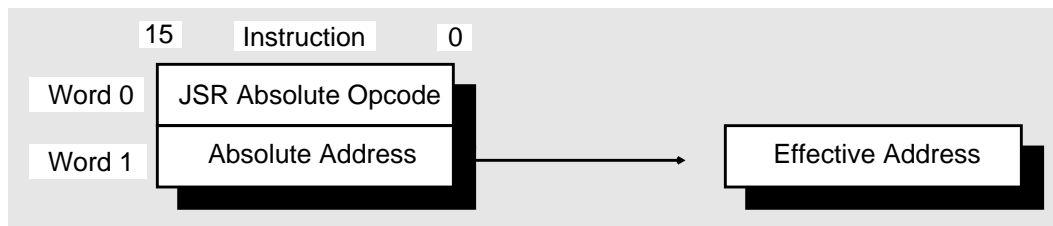


Figure 13-5. Absolute Addressing for Code and Data

Figure 13-6 shows how absolute addressing will position code in memory. Table 13-3 shows the commands for this example. The base address (**BA**) is the address where the program is loaded. When the target of the **JSR** is computed, the Program Counter (**PC**) contains the address of the word containing the absolute address.

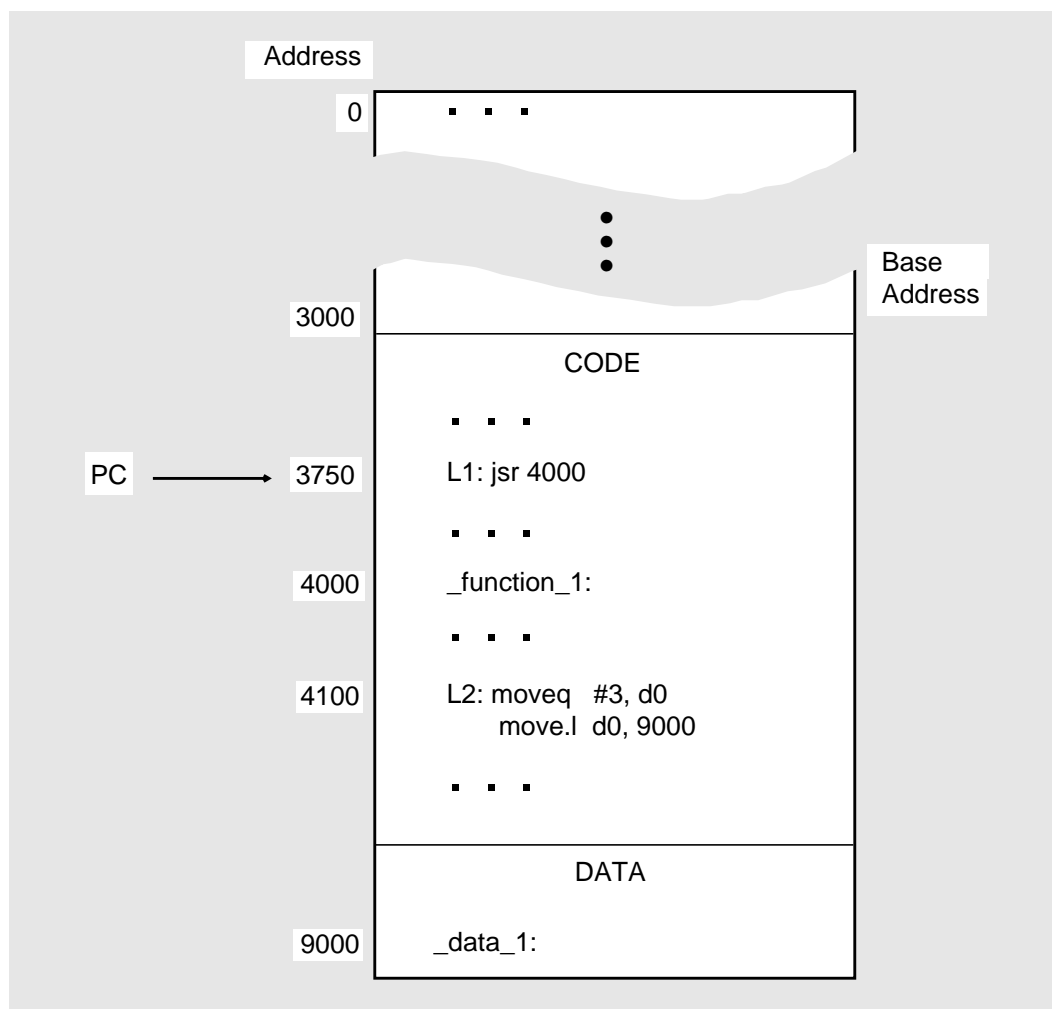


Figure 13-6. Absolute Addressing Example

Table 13-3. Commands for Absolute Addressing Example

PC	Description
3750 = L1	Calls <code>_function_1</code> : <code>jsr _function_1</code>
4000 = <code>_function_1</code>	The next section of code “...” is executed
4100 = L2	The value 3 is assigned to <code>_data_1</code> : <code>moveq #3,D0</code> <code>move.l d0,_data_1</code>

PC-Relative Addressing

Figure 13-7 shows PC-relative addressing for code and data. The **JSR** (jump to sub-routine) instruction uses a PC-relative code address, and the **LEA** (load effective address) instruction uses a PC-relative data address. The effective address is determined by adding the contents of the program counter register to the displacement (or offset). When the target of the **JSR** is computed, the program counter contains the address of the word containing the offset.

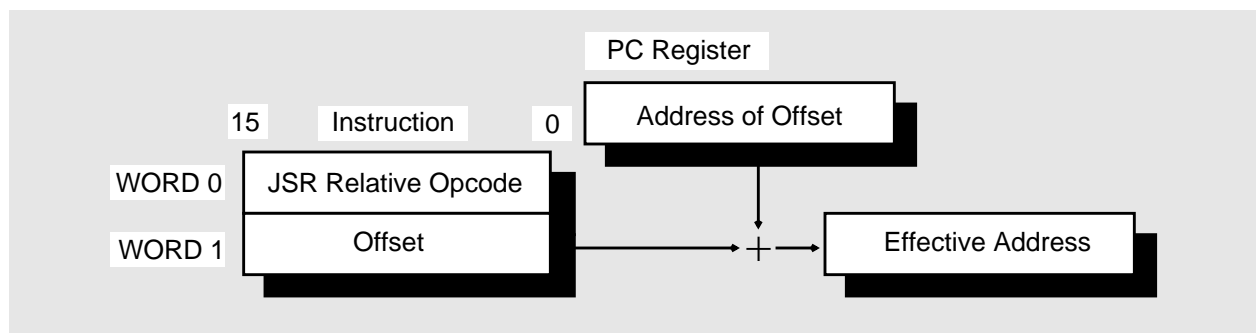
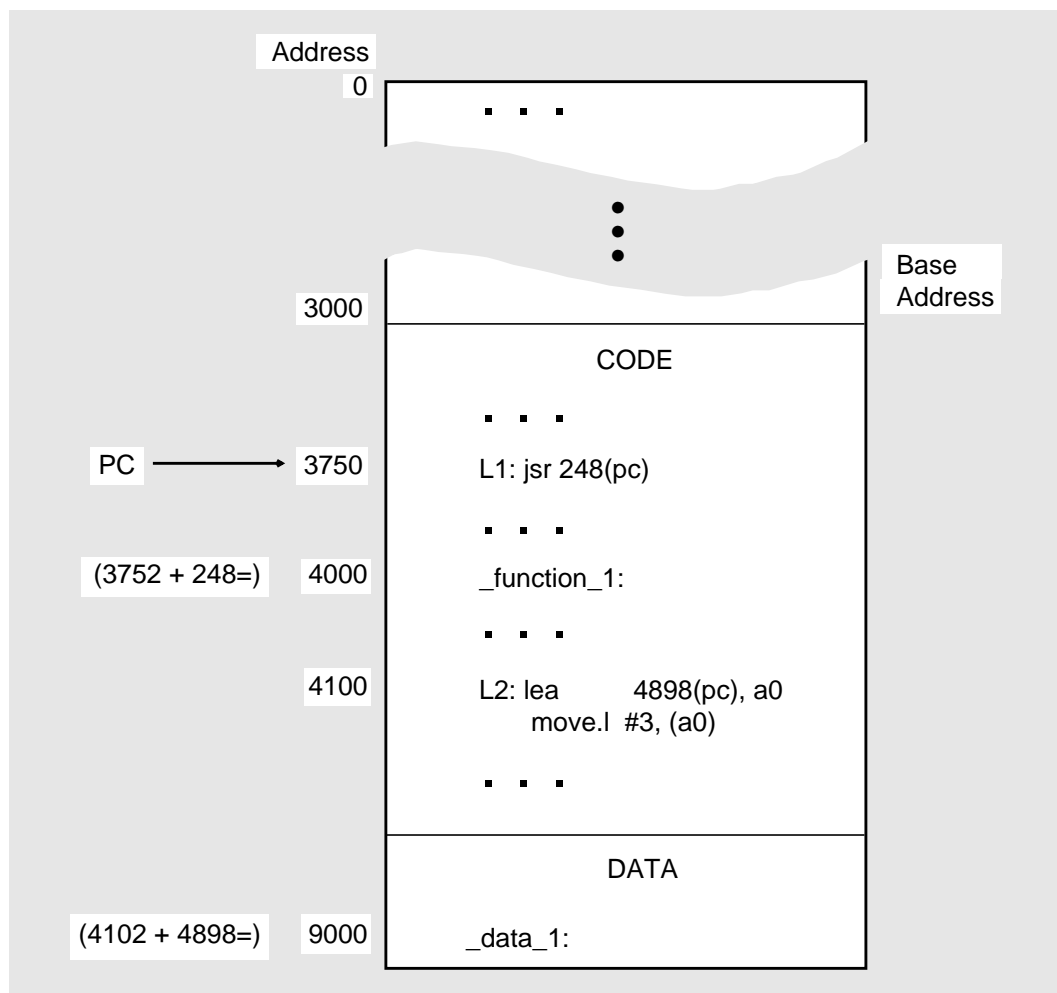
**Figure 13-7. PC-Relative Addressing**

Figure 13-8 illustrates how PC-relative addressing will position code in memory. Table 13-4 shows the commands for this example. The base address is the address where the program is loaded (that is, determined at load time). Regardless of where the program is loaded, the relative positions of code and data remain the same.

Table 13-4. Commands for PC-Relative Addressing Example

PC	Description
3750 = L1	Calls <code>_function_1</code> : <code>jsr _function_1(pc)</code>
3752 + 248 = 4000 = <code>_function_1</code>	The next section of code “...” is executed
4100 = L2	The value 3 is assigned to <code>_data_1</code> : <code>lea _data_1(PC), a0</code> <code>move.l #3, (a0)</code>

**Figure 13-8. PC-Relative Addressing Example**

User-Modified Routines

This section describes the routines that need to be modified when using the Microtec C/C++ compiler in certain environments. Sample versions of these routines are provided on the distribution, but it is assumed that these sample routines will be modified extensively or used only as examples.

User-Modified Routines for Embedded Systems

If you are running on a non-UNIX or an embedded system, the routines **inchrw**, **outchr**, **entry**, **_START**, and the functions **sbrk** and **_exit** must be modified. These routines are described in the following sections. Simple versions of these routines and functions are provided in distribution files. These sample routines work with the XRAY Debugger and are sufficient for initial debugging in that environment.

In Character Routine

If the SysHost feature is being used (XRAY version 4 and beyond), then all input is automatically provided through the read function. The XRAY Simulator **stdin** command can be used to specify the source of input data:

```
stdin f=<filename>; get input from file <filename>
```

or

```
stdin win ; fetch from stdio window
```

The SysHost feature can be disabled by compiling the libraries with the symbol **EXCLUDE_SYSHOST** defined.

If SysHost is not used, the **inchrw()** routine reads characters from the **_simulated_input** variable. The XRAY Debugger **INPORT** command may be used to specify the source of input data for **_simulated_input**:

```
inport &_simulated_input [,input_source]
```

If no **INPORT** command is issued, the XRAY Debugger reads from the standard input device.

inchrw() returns an integer with a value between 0 and 255, or -1 if an error occurs.

Out Character Routine

If the SysHost feature is being used (XRAY version 4.0 and beyond), then all output is automatically provided through the **write** function. The following XRAY Simu-

lator **OUTPORT** command can be used to specify the destination of standard output data:

```
stdout f=<filename> ; write output to file <filename>
```

Similarly, the following command can be used to specify the destination of standard error data:

```
stdout,err f=<filename> ; write errors to file <filename>
```

Output can be rerouted to the **stdio** window using the following command:

```
stdout[,err] win
```

The SysHost feature can be disabled by compiling the libraries with the symbol **EXCLUDE_SYSHOST** defined.

If SysHost is not used, the **outchr(ch)** routine writes characters to the **_simulated_output** variable. The XRAY Debugger **OUTPORT** command may be used to specify the destination of the output data for **_simulated_output**:

```
outport &_amp;_simulated_output [ ,output_destination]
```

If no **OUTPORT** command is issued, the XRAY Debugger writes to the standard output device. Its return value is ignored.

Start Routine

The start routine (**_START**) does the following:

- Initializes **_randx**
- Completes initialization of the heap
- Initializes the I/O system
- Opens the standard files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**
- Initializes global variables
- Places **_simulated_input** and **_simulated_output** in the **ioports** section. This section should be located in RAM.
- Can call the **initcopy** routine for copying initialized data in the **vars** section from ROM to RAM (see *Using the Linker* in this chapter for a description of the LNK68K/CF **INITDATA** command)
- Calls the **main** function with the three arguments **argc**, **argv**, and **_environ**
- Calls **exit** whenever the **main** function returns. The **exit** function flushes and closes all opened files and then calls **_exit** to terminate the program.

Exit Routine

A simple version of the `_exit` routine is included in the file **entry.c**. The **exit** routine is implicitly called at the end of each **main()** function to return control to the operating system. In this implementation, **exit** closes all opened files then calls `_exit`.

Initialization Routine

The initialization routine initializes the embedded system environment. It should meet the following requirements:

- Be written in assembly language
- Set up values for the stack pointer
- Provide a static variable for the heap pointer
- Call the **_START** routine

The sample initialization routine (**entry**) is the main entry point. It does the following:

- Initializes **SP** as **.STARTOF(STACK) + .SIZEOF(STACK)**, which will be defined at link time
- Initializes **__HEAP** as the starting address of the **heap** section, which is located after all other sections
- Initializes the frame pointer
- Clears the **zerovars** section
- Sets the **a5** register for **a5**-relative data addressing (optional, done in **mcc68ka5xx.lib**)
- Calls the C initialization routine **START**

Heap Management Routine

The **sbrk** function keeps track of the heap's growth and allocates space from the heap. It increments the **__HEAP** pointer by *size* bytes and returns the previous heap pointer value in register **D0**. If there is not enough space on the heap, **sbrk** returns -1.

System Functions and SysHost feature

A part of the set of functions provided in the Microtec C/C++ Compiler run-time library includes low level UNIX-style system functions, which are usually called indirectly through other library functions. How these functions are implemented

relies on the nature of the execution environment. Most of the system functions provided are designed to work with the XRAY Debugger (Simulator or Monitor) SysHost feature. These routines (and the functions that call them) are intended to be used only during debugging.

The SysHost feature is provided to allow the application access to host resources (such as the file system) during the development phase. This feature works by filling a transfer buffer with data indicating the desired function to be performed on the host. XRAY then stops the application and performs the requested function on the host system.

The SysHost feature is enabled in XRAY version 4.0 or later. If the SysHost feature is not enabled, then the system routines, except **read** and **write**, essentially act as stubs. A table of system functions is provided in Chapter 5, *Using Libraries*. The source code for the system functions is provided with the libraries. The SysHost-specific code can be removed by compiling the libraries with the symbol **EXCLUDE_SYSHOST** defined.

If you wish to use these routines outside of the debugger, you will probably need to modify them to suit your environment or obtain them from a different library suitable to your environment. System routines coming from alternate sources can supersede the compiler's libraries by specifying the file names on the command line or in the linker command file, provided that they are in a compatible format. If the compiler library pathname is also specified, then the alternate source names must precede it. Other system routines which will require changes include the initialization routine and **_exit()** in **entry.c**; **_START()** in **csys.c**; and **_sbrk()** in **s_sbrk.c**. A description of these is provided in the section *User-Modified Routines* in this chapter.

If the SysHost feature is not used, then the **read** function, located in **s_read.c**, reads from the variable **_simulated_input**. **read()** returns the number of bytes read, 0 for **EOF**, or -1 to indicate an error. The **write** function, located in **s_write.c**, writes to the variable **_simulated_output** or returns -1 to indicate an error. To replace any of these functions with one of your own, link in your implementation before the run-time library.

The **creat** function uses the **open** function, which is implemented as a stub. You must provide your own **open** function to allow **creat** to work in your environment.

The **sbrk** function is provided as a minimal implementation. If you are developing code for an environment that requires a different **sbrk**, you can provide a replacement for the **sbrk** function by linking in your implementation of **sbrk** before the run-time library.

Removing Unneeded I/O Support

You can reduce code size by not linking in the full set of UNIX system functions included in the distribution, but you may still need formatted I/O. In some embedded applications, the formatted I/O requirements are not extensive enough to involve a UNIX-style system. In other applications, the formatted I/O is only necessary during project development and is removed before the code is finished.

In these cases, make the library routine **printf** call your character output routine directly. Library routines like **printf** implicitly write to **stdout** by passing characters from the library routines **putc** and **flushbuf** to the **write** routine. Typically, you would modify the source code for the **write** routine to work with your particular system hardware. If the UNIX system functions are not linked in, **write** is unavailable.

To allow formatted output to be sent directly to a low-level output routine, use one of the following two methods:

- Use **sprintf** rather than **printf** for formatted output. The **sprintf** function does the same formatting as **printf** but places the output in a user-specified string followed by the **NULL** character. The string may then be copied a character at a time to your output routine in a simple function such as:

Example:

```
str_out (char *str)
{
    while (*str)
        _simulated_output = *str++;
}
```

- Rewrite the **putc** routine so that all of the output characters are available directly from **printf**. In the C library, the **printf** routine calls **putc** to output individual characters. Your **putc** routine will also intercept the characters output by other library routines that use **putc**, such as **fprintf**, **fprintf**, **vfprintf**, and **vprintf**.

Example:

```
#include <stdio.h>
#undef putc /* shut off putc macro */
volatile extern char _simulated_output;
int putc(int c, FILE *stream)
{
    _simulated_output = c;
    return(c);
}
```

Note

The sample code provided here is not portable and relies on calls between library functions. This example may need to be modified for your version of the compiler.

Removing Unneeded Floating-Point Support

If **printf**, **fprintf**, or **sprintf** is used and no floating-point values are ever read, you can avoid linking in full floating-point support by adding the following stub for **_ftoa** and compiling and linking it before you link in the library:

```
int _ftoa() { return 0; }
```

Alternatively, you could add the following stub for **_sfef** and compile and link it before you link in the library:

```
void _sfef() { return; }
```

In both cases, this technique reduces the size of the executable code.

Reserving a Register

Since it can be useful to reserve a register for shared data, system data, reentrant programs, or shared programs, the Microtec C compiler allows you to reserve **An** registers (from **A2**, **A3**, **A4**, **A5**, or **A6**) and **Dn** registers (from **D2**, **D3**, **D4**, **D5**, or **D6**). Up to three **An** and up to three **Dn** registers can be reserved during a given compilation. The compiler does not generate any code that uses the reserved register. If the **A6** register is reserved, **A5** will be used as the frame pointer.

A reserved register is usually used in conjunction with a Multitasking Environment (MTE). A multitasking environment handles one or more programs executing simultaneously. An MTE can be thought of as an operating system; however, it might also be part of the application program or part of an embedded system.

To reserve a register, use the **-Khreg** option described under section *Producing Minor Code Generation Variations* in Chapter 3, *Using Command Line Options*. Enter the option on the command line followed by the registers to be reserved. When this option is used, the generated code can be slightly less efficient in some cases.

Shared Program

When a register is reserved for a shared program, a single physical copy of the program can be shared by one or more users simultaneously. These can be single-tasking, multitasking, or interrupt-based programs, but only local variables and ROM variables (global or static) can be used. In this case, the reserved register is used as a pointer to the currently used data area (it points to only one data area at a time). Shared programs must be reentrant (see Figure 13-9).

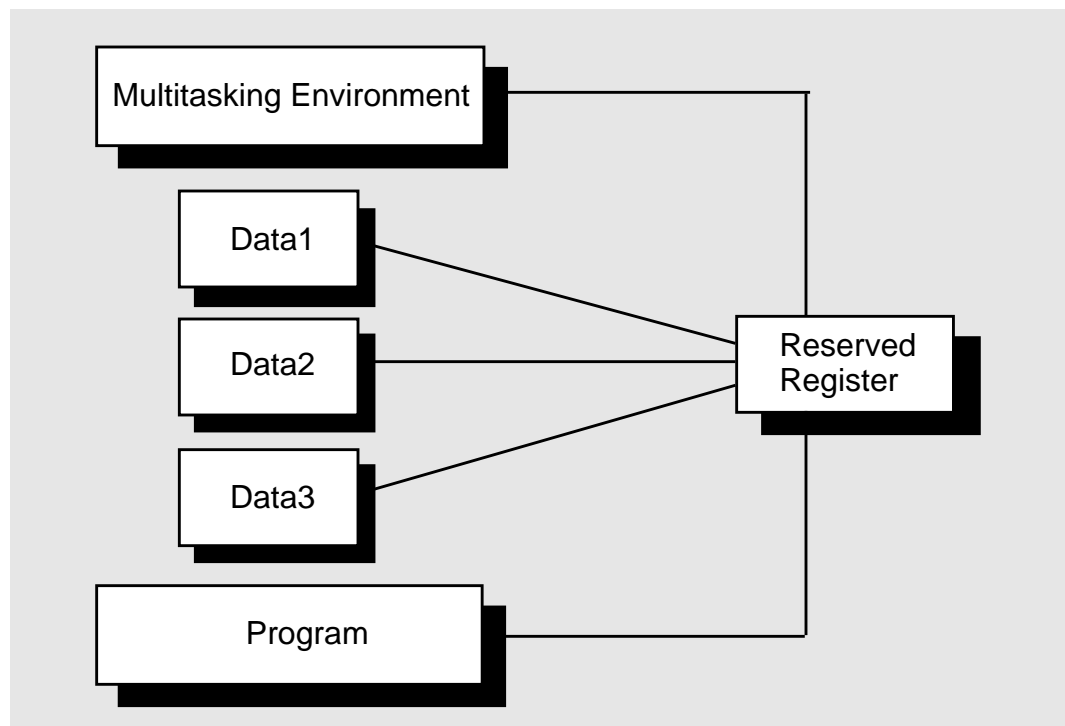


Figure 13-9. Shared Programs

Shared Data

When a register is reserved for shared data, a single physical copy of data in a multitasking environment can be shared among more than one program. The reserved register is used as a pointer to the shared data area as shown in Figure 13-10.

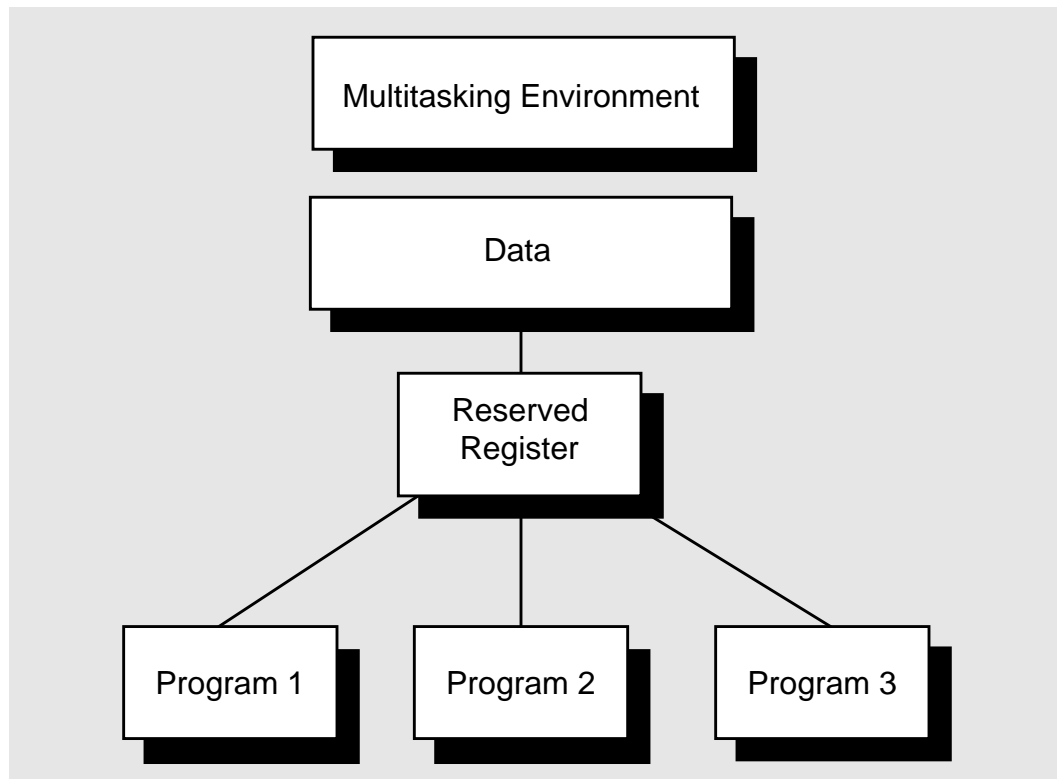


Figure 13-10. Shared Data

System Data

When a register is reserved to point to a system data area, special system data can be manipulated by a program. The multitasking environment sets up the system data area before the program is loaded. The multitasking environment must also load the address of the system data area into the reserved register. This configuration is shown in Figure 13-11

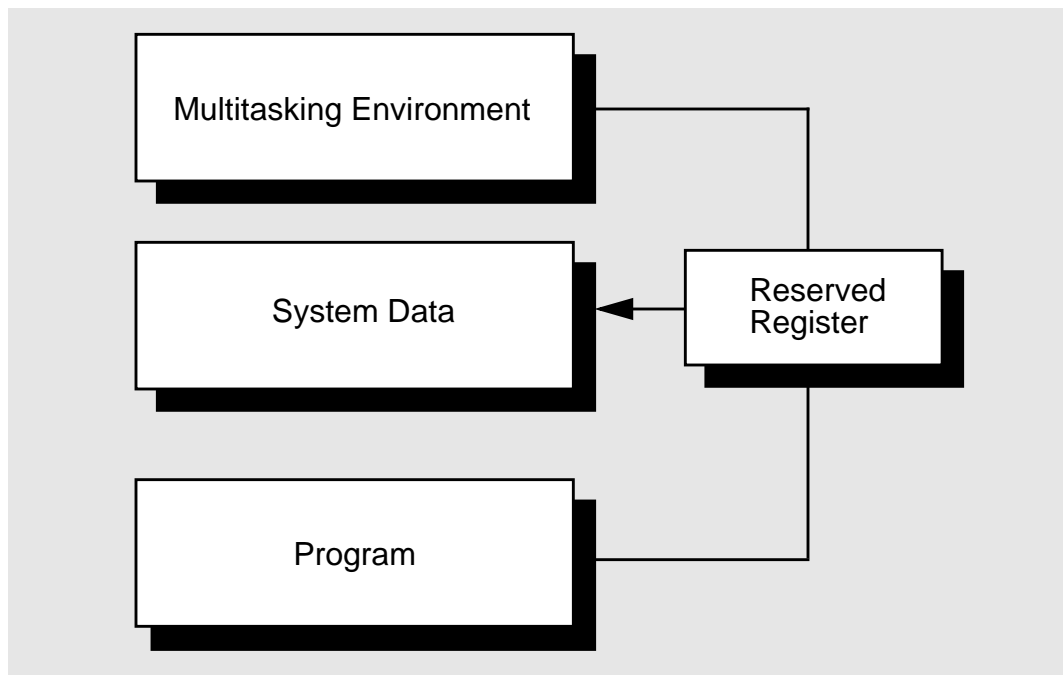


Figure 13-11. System Data

Special Purposes

It can be useful to reserve a register for special purposes. For example, a register can be used exclusively by an assembly language routine that is to be linked with C or C++ code. A register might also be reserved for a fast recall of a certain address or value, for locating data for ROM-based routines, and for memory management.

When a register is reserved for special purposes, it can either point to data or store the area that contains a special value. In Figure 13-12, a ROM-based routine reserves a register for a “hook” to find its data.

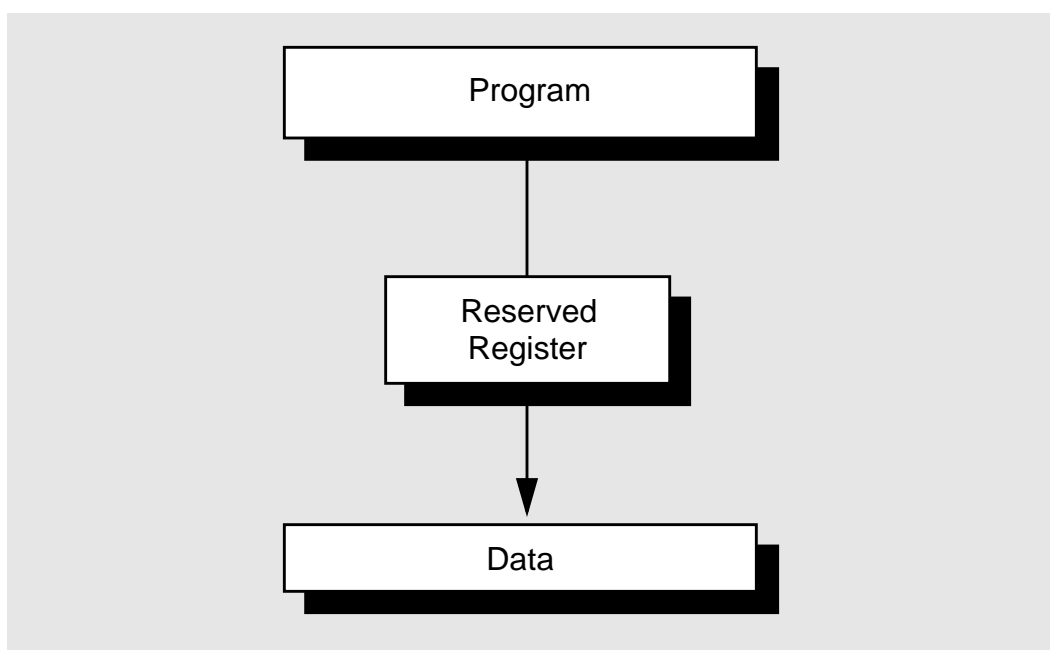


Figure 13-12. Reserved Register as a Pointer to Data

In Figure 13-13, the reserved register acts as a storage area containing a value used by the assembly language routine.

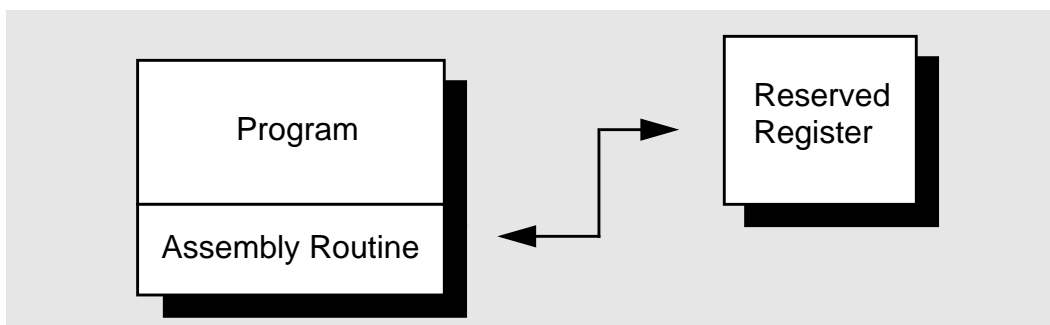


Figure 13-13. Reserved Register as a Storage Area

Data Initialization

Variable values can be initialized at run time or compile time.

Example:

Run-Time Initialization

```
int i;  
i = 15;
```

Compile-Time Initialization

```
int i = 15;
```

Code is always ROMable when variables are initialized at run time. In this example, the constant value 15 is stored in the **code** section.

When a variable is initialized at compile time, the compiler allocates the variable, along with its value, in the **vars** section. The **vars** section is often located in RAM. For most ROM-based embedded systems, this is a problem. When the program executes, RAM is typically not initialized with the variable values contained in the **data** section. The variable needs to be in ROM.

The run-time libraries provided with the C/C++ Compiler contain initialized variables. So the problem exists for run-time library initialized variables, as well as for user-initialized variables.

For more information about the initialization process, refer to the C system initialization routine file **csys.c**, located in **\$COMPILER_HOME/etc/mcc68k/rtl/src/** under UNIX or **%COMPILER_HOME%\etc\mcc68k\rtl\src** under Windows.

Saving Initialized Variables in ROM

Since RAM is not usually initialized with the expected variable values when the program starts executing, many embedded programmers use the convention that only constants can be initialized on the variable declaration. The value of the variables cannot be changed at run time. In this case, the **vars** section can be placed in ROM, along with the **code** section.

If you do not want variables initialized at compile time on the data declaration to be constants, use the **INITDATA** facility available in most ROM/RAM systems. This facility initializes the values of these variables when the program restarts.

Using the Linker

This section provides linker command examples, including an example using a ROM-based system. See Chapter 12, *Run-Time Organization*, for a description of the default section names the linker uses.

Default Sections

Refer to Chapter 12, *Run-Time Organization*, in this manual for information about default linker sections. Refer to the section *Choose Address Mode for Sections* in Chapter 3, *Using Command Line Options*, for information on locating and addressing the various code and data sections.

Libraries

For information about which libraries to use, refer to Chapter 5, *Using Libraries*.

Messenger Symbols

A messenger symbol is a compiler-generated symbol that is resolved by the run-time library. This symbol is identified by four preceding underscores.

The **-f** compiler option generates the messenger symbol `____FPU`. If this messenger symbol is present, the XRAY Debugger sets the `@fpu` pseudo-register to 1 (indicating the presence of the 68881 floating-point coprocessor).

INITDATA Command

The linker command **INITDATA** provides a method for initializing program variables in RAM at program start-up time. If the **INITDATA** command is used, the linker creates the section **??INITDATA**. This section contains the initialization values for the section(s) specified with the **INITDATA** command. At program start-up time, the **initcopy** function copies the initialization values from the **??INITDATA** section to the section(s) specified with the **INITDATA** command.

The Microtec Compiler run-time libraries contain the **initcopy** function. By default, the **initcopy** function is called from the **START** function in **csys.c**. The **csys.c** file on the distribution includes a call to **initcopy**.

To disable the **INITDATA** command, follow these steps:

1. Edit your linker command file and remove this command:

```
initdata vars
```

2. Do one of the following:
 - a. Define the preprocessor macro **EXCLUDE_INITDATA** either on the compiler command line with the **-D** option or in your program with the following directive:


```
#define EXCLUDE_INITDATA 1
```
 - b. Edit the file **csys.c** and remove the **#if EXCLUDE_INITDATA** preprocessor directives and their associated **#endif** directives. Then recompile the file and link it in before the library.

Linker Command Example

This sample linker command file creates the memory configuration in Figure 13-14.

```
listabs publics, internals
listmap publics
format ieee           ; Output file format
extern ENTRY          ; Force load of initialization
                      ; routine
sectsize heap=$8000    ; Set size of heap
sectsize stack=$1000   ; Set size of stack
common stack=$F000     ; Set start of stack section

order stack            ; Stack section
order literals,strings,const,code ; ROM Section
order vars,zerosvars,ioports,heap ; RAM Section
load sieve             ; User application routine
load mcc68kab.lib      ; C run-time library
end                    ; End of command file
```

The compiler places code and data in the sections described in the section *Default Sections* in this chapter. The example assembly module **ENTRY** uses section **heap** for the heap.

By default, the linker begins assigning memory at address 0, which is unsuitable for many real applications because addresses 0 to 3FF are used as interrupt vectors and space needs to be allocated for the stack. Use the **COMMON** command to specify the starting point address of the **stack** section.

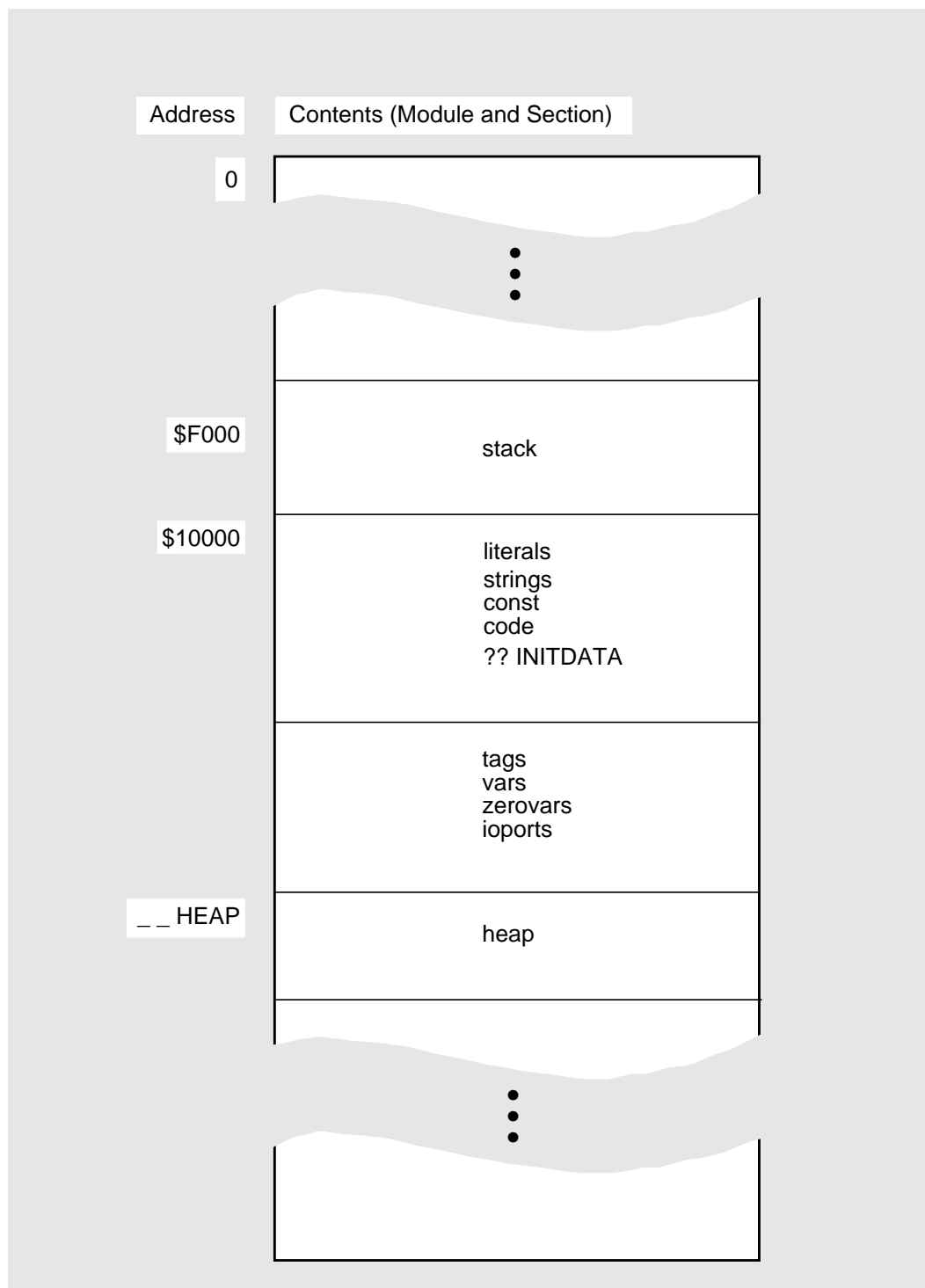


Figure 13-14. Memory Configuration

Linker Command Example (ROM-Based System)

In some applications, such as ROM-based dedicated systems, code, data, and stack sections are assigned to preallocated areas of the processor's address space. In the following linker command file, it is assumed that the **literals** section is placed in the ROM area starting at F000, the **zerovars** section starts at location 2000, the **vars** section starts at location A000, and the stack resides in the 0 to 2000 region.

```
listabs publics, internals
listmap publics           ; Linker options
format ieee               ; Output file format
extern ENTRY               ; Force load of
                           ; initialization routine

sect ??INITDATA=$E000
sect literals=$F000        ; Code section address
sect zerovars=$2000        ; Uninitialized data
                           ; section address
sect vars=$A000            ; Initialized data
                           ; section address

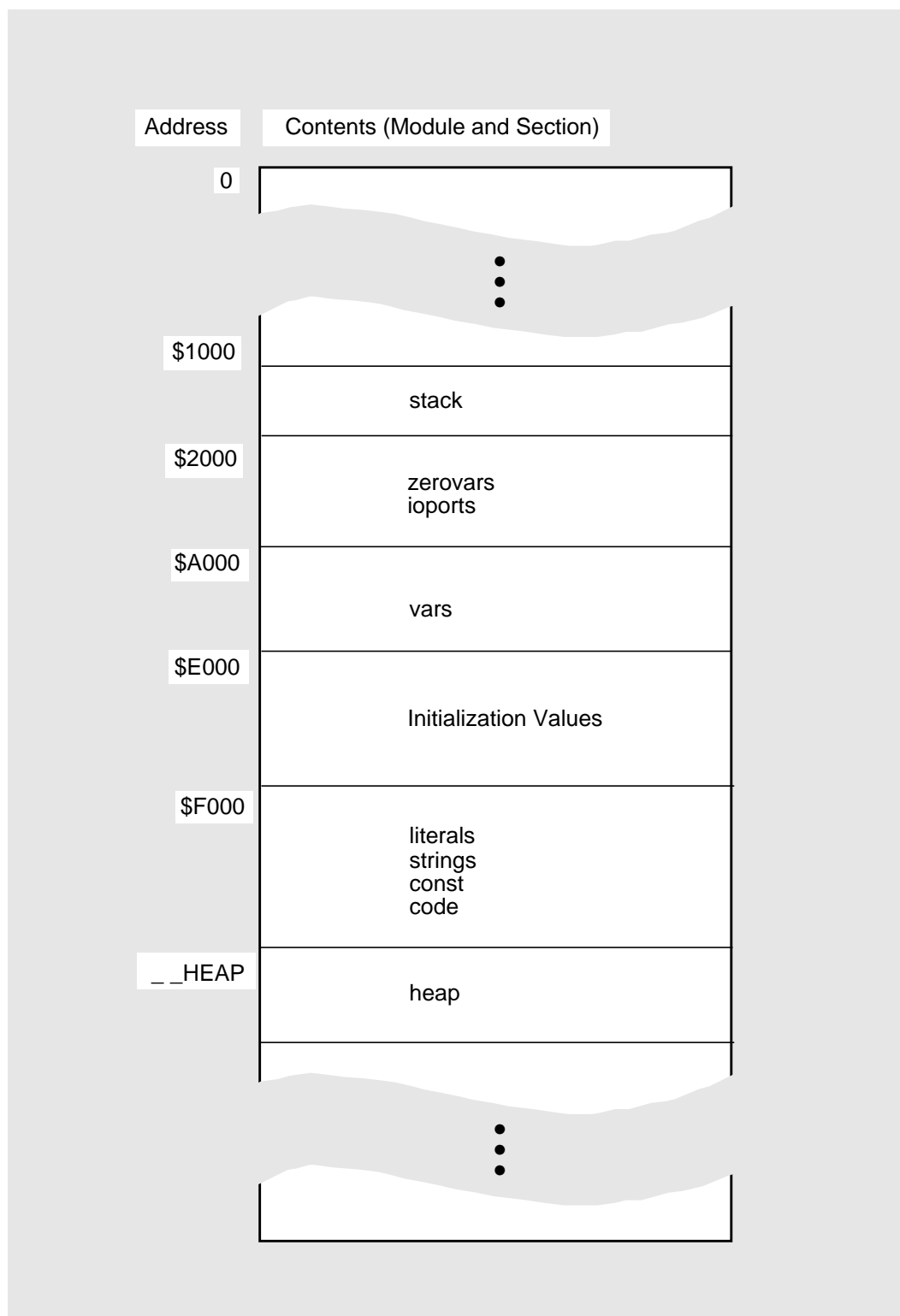
common stack=$1000        ; Set start of stack section
sectsize heap=$8000       ; Set size of heap
sectsize stack=$1000      ; Set size of stack
;
order stack                ; Stack section
order zerovars,ioports,vars,??INITDATA
order literals,strings,const,code,heap
initdata vars              ; Put init values in ??INITDATA
load entry                 ; Main entry point
load sieve                 ; User-application routine
load mcc68kab.lib          ; C run-time library
end                         ; End of command file
```

Figure 13-15 shows this memory configuration.

The initialized variables are allocated in section **vars** and their initialization values are allocated in the **??INITDATA** section, which is placed at E000. To copy the initialization values from **??INITDATA** (ROM) to **vars** (RAM), **initcopy** should be called from the **entry** function.

Using the allocation map generated by the linker, further adjustments to the starting addresses might be necessary to ensure sufficient space for each program section.

The **ABSOLUTE** linker command can be used to control what sections are placed in the output file. For more information about this and other linker commands, see the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the 68000 and ColdFire Families*.

**Figure 13-15. Memory Configuration (ROM Based)**

Interrupt Handlers

Interrupt handlers perform the necessary functions when an interrupt occurs. They preserve the values in various registers and storage locations and transfer control to routines to service the interrupt.

If the **-KF** option is used, the interrupt handler saves the contents of **FP0** and **FP1**, the status and control registers (**FPSR** and **FPCR**), and the internal state of the FPU (**FSAVE**).

By default, the Microtec C/C++ compiler assumes that the values in registers **A0**, **A1**, **D0**, and **D1** may be destroyed by each called function. You can declare a function as an interrupt handler to force the values of these registers to be saved on the stack on entry to the function and to be restored on exit.

A function can be declared as an interrupt handler by using the **interrupt** keyword. Interrupt functions should have no parameters and should return the **void** type.

The following example shows how to set an interrupt function for a 68020 address error when the interrupt vector table begins at memory location 0x1000.

Example:

```
interrupt void foo_int(void)
{ ...
}

void set_vector(void)
{
    /* setting interrupt function for address
       error Vector(3) */
    *(int *)0x100c = (int)foo_int;
}

main()
{
    /* set Vector Base Register */
    asm("      move.l #1000,d0");
    asm("      movec  d0,vbr");

    set_vector();
    ...
}
```

By default, a function declared as an interrupt handler returns to the calling function with the **Return To Exception (RTE)** instruction instead of the standard **RTS** instruction. If the **-Kr** option is used, the interrupt handler will return with an **RTS** instruction.

For more information about interrupt vector tables, refer to the Motorola reference manual appropriate to your chip.

Static Initialization and Termination

C++ static objects need to be initialized during program start-up or before the first reference to the static object. C++ static objects also need to be destroyed upon program termination in the reverse order of their construction or initialization.

Static Constructor

To ensure proper construction (initialization) of the static objects, the C/C++ compiler generates the following code:

- A static constructor routine that calls constructors or performs necessary initialization code to construct each static object in a given file
- A function pointer to the static constructor in the predefined section **initfini** for the C++ run-time library to call at program start-up time

All static constructors are typically prefixed with `__sti__`. One static constructor is generated on a per-file basis. If the given file has no static objects, a static constructor is not generated.

Static Destructor

To ensure proper destruction of the static objects, the C/C++ compiler emits the following code:

- A static destructor routine that calls appropriate destructors or destroys each static object in a given file
- A function pointer to the static destructor in the predefined section **initfini** for the C++ run-time library to call at program termination

Static destructor names are typically prefixed with `__std__`. At most, one static destructor is generated for each C++ source file. If there are no static objects in the source file, no static destructor is generated.

initfini Section

The **initfini** section in an object file contains a list of pointers to the `__sti__` and the `__std__` functions; these functions must be called to initialize the static data of a given file. The linker combines the **initfini** sections from all files in a program into

a single **initfini** section. Consult the linker command file to see the exact placement of the **initfini** section (this section is usually placed with other constant data).

If you do not want to use the default linker command file or if you want to load sections in a different order, specify this information in your linker command file. Specify the **initfini** section to ensure enough space is allocated in ROM if this section is produced.

The entries into the **initfini** section are like a pair of pointers pointing, respectively, to the static constructor and destructor for a given C++ file. For example, for the two files **file1.c** and **file2.c**, the C/C++ compiler would generate, at most, one static constructor and one static destructor for each file. If the C/C++ compiler generated one static constructor and one static destructor for **file1.c** and only one static constructor for **file2.c**, the constructors and destructors would be assigned to the individual file's **initfini** section before linking, as shown in Figure 13-16.

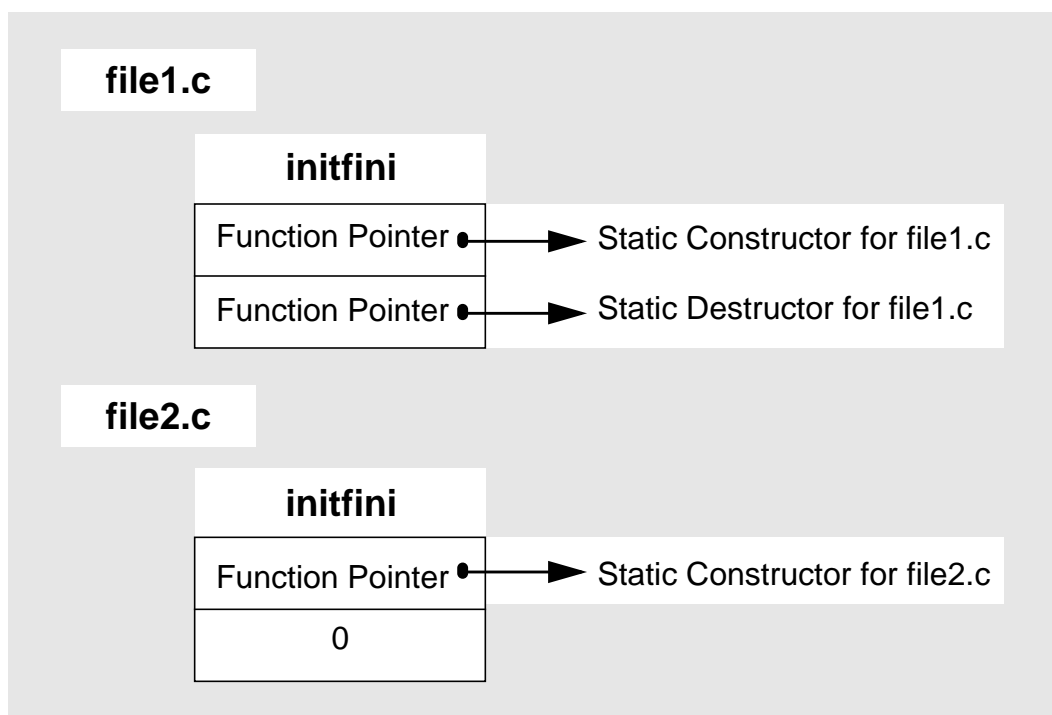


Figure 13-16. initfini Section Before Linking

Figure 13-17 shows the **initfini** section after linking. The constructors for **file1.c** and **file2.c** are invoked and executed during program startup by **_START**, the run-time startup routine. Upon program termination, the destructor for **file1.c** is called by **_exit()**.

Byte Offset	initfini
0	Pointer to Static Constructor for file1.c
4	Pointer to Static Destructor for file1.c
8	Pointer to Static Constructor for file2.c
12	0

Figure 13-17. initfini Section After Linking

If you use the C++ I/O stream classes and objects, such as **cin** and **cout**, you will notice that the static constructors and destructors for stream classes and objects are called by **_START** and **_exit()**. Figure 13-18 shows the initialization call sequence (filename extensions are UNIX-specific and can be different for other operating systems).

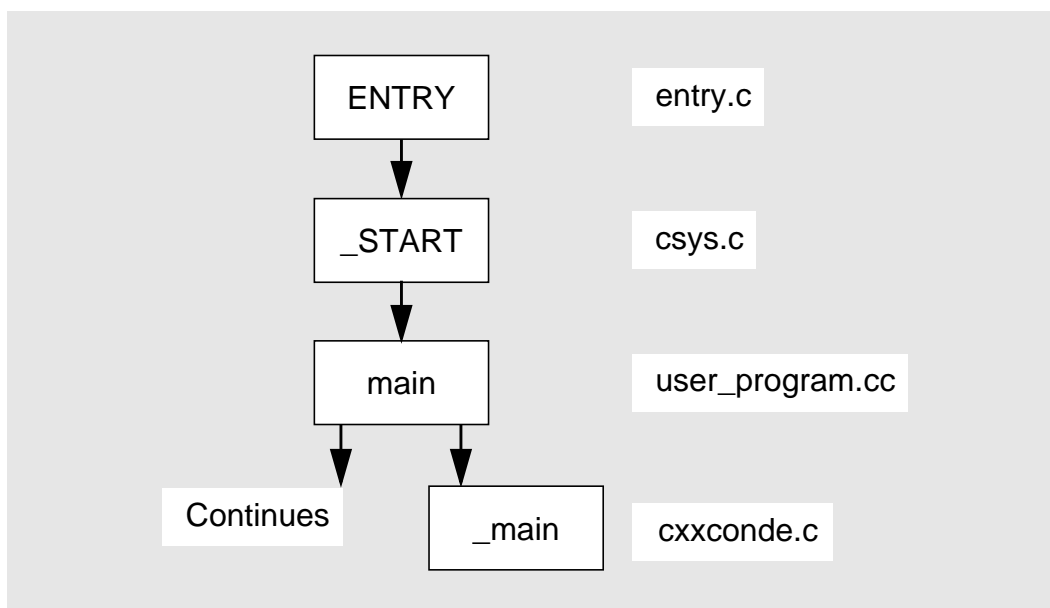


Figure 13-18. C++ Initialization Call Sequence

The start-up routine **_START** is invoked to initialize library globals and open standard files at the beginning of run time and before the **main()** routine of your program is called. **main** calls the **_main** routine before any application code is executed. At this time, all your static constructors are initialized, and the C++ I/O layer level 1 (referred to as layer 1+) is initialized.

Figure 13-19 shows the termination sequence (filename extensions are UNIX-specific and can be different for other operating systems).

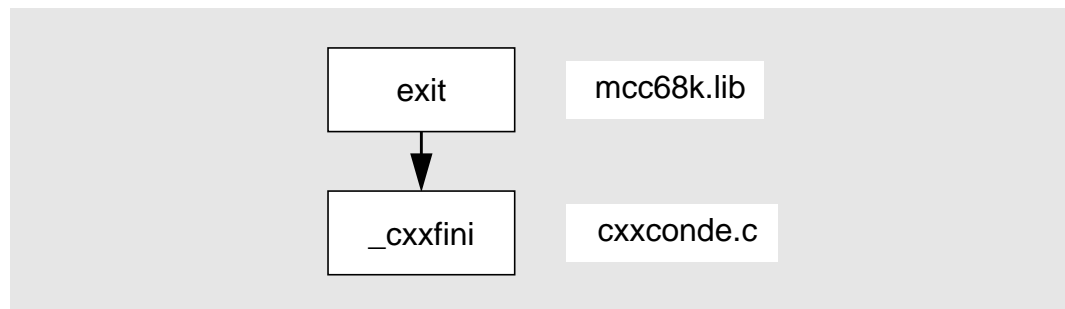


Figure 13-19. C++ Termination Call Sequence

The standard **exit** routine is invoked from the compiler library when you exit your program. This routine calls the C++ routine **cxxfini**, which calls all static destructors to deinitialize all C++ static objects in your program.

pixinit Section

The internal virtual tables that are generated by the compiler are read-only tables and so are put into the **const** section. However, when position-independent code (PIC) is specified, then the virtual tables are not made **const** and are initialized during PIX initialization. The virtual tables contain offsets and pointers to code. For position-independent data (PID), the virtual tables do not need initialization, and can be placed in the **const** section.

Virtual tables used with position-independent code (PIC) require initialization. When PIC is specified, the virtual tables generated are placed in the **vars** section.

The C++ compiler generates a “run-time position-independent code initializer” routine for each module that contains virtual functions. The initializer routine reinitializes pointers to the virtual functions in that module. The name of the initializer routine is the name of the module with the prefix **_PICsti_**. There is one initializer routine per module.

Figure 13-20 shows the compiler placing a pointer to each **_PICsti_** routine in the **pixinit** section. The **_main** routine uses the pointers in the **pixinit** section to call each **_PICsti_** routine and update all virtual function addresses before any static constructor is called. By dynamically initializing these addresses after the user program is loaded, position-independent addresses to virtual functions are properly resolved before the virtual functions are called.

The **pixinit** section must be placed in ROM.

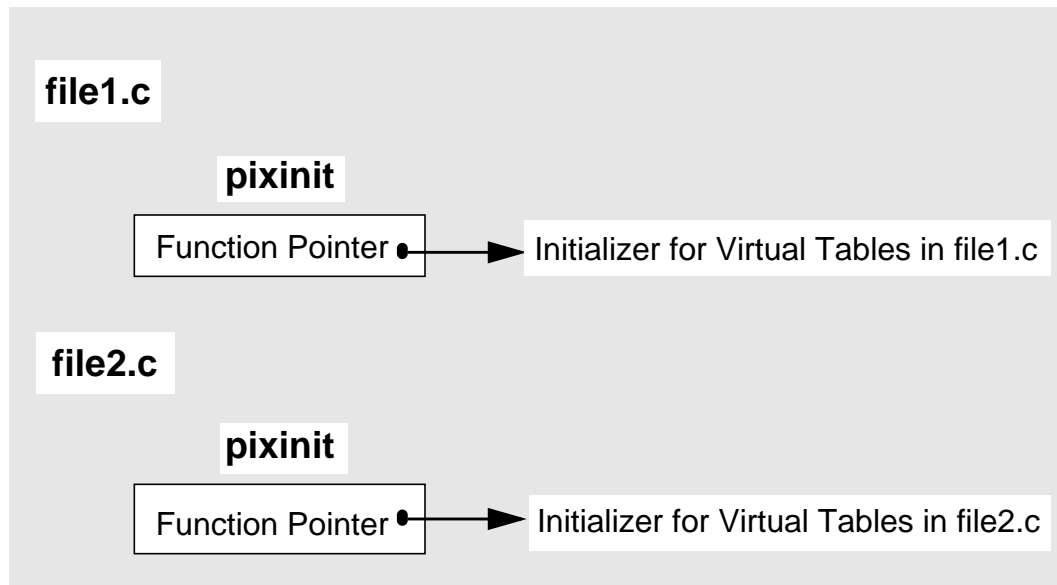


Figure 13-20. `pixinit` Section before Linking

I/O Static Initialization and Termination

The C++ static I/O objects **`cout`**, **`cin`**, **`cerr`**, and **`clog`** are defined in the C++ I/O stream class libraries. The I/O channel for **`cout`** is tied with **`stdout`**, **`cin`** with **`stdin`**, and **`cerr`** with **`stderr`**.

I/O channel connections between **`stdout/stdin/stderr`** and **`cout/cin/cerr`** are part of the static initialization of **`cout/cin/cerr`** objects. If you use static I/O stream objects in the start-up code of your application, static constructors for the I/O stream objects are invoked to initialize the I/O channels, buffers, states, and so forth, for the I/O static objects. As the application terminates, all corresponding static destructors are invoked to flush the I/O buffers before the I/O channels are closed.

Exception Handling

This section describes the issues specific to developing exception handling applications in embedded environments. The compiler constructs a type database and a function return-address table for the program; this information must be placed properly in memory to be accessed by the exception handling run-time system (EHS).

Type Database

The compiler constructs a database for the source file that contains static type information of the following type:

- Throw object types, type hierarchy information, and type access and conversion information
- Catch argument types and type hierarchy information
- Exception specification types
- Local object construction lists

The exception handling run-time system uses this information to perform the following tasks:

- Check the exception specification
- Destroy the appropriate local objects including the destruction of partially constructed objects and arrays
- Match the type of the throw object to the type of the handler arguments

Save/Restore Functions

The exception handling system (EHS) save/restore functions allow exception handling to be used in a multi-threaded environment. In this environment, the state of non-stack EHS run-time variables must be saved when a context switch occurs. The library user implements the context switch mechanism and memory management routines; you must tell the run-time EHS when a context switch is occurring and what memory region can be used to save the previous EHS context. Inform the EHS run time of these events with the EHS save/restore interface functions.

Set aside a memory buffer for each thread's EHS context before forking the execution into multiple threads. Required size is fixed and does not vary from thread to thread. The size varies between targets and ranges from around 60 to 100 bytes. To determine the required size of this buffer, use the following interface function:

```
unsigned int _ehs_save_size(void);
```

Once you have allocated an EHS context buffer, initialize it before its first use. Do this with the following interface function:

```
void _ehs_init_area( void * areaPtr);
```

where `areaPtr` is a pointer to the EHS context buffer you allocated.

When a context switch occurs (by interrupt or other means), the context switching code must perform the following call prior to exiting the current thread's context:

```
void _ehs_save_to_area( void * oldareaPtr);
```

`oldareaPtr` should point to the EHS context buffer for the currently executing thread. Once the thread context has been saved, a context switch can be performed. Once the context has been switched to the context of the target thread, the context switch code calls the following function to restore the EHS context of the thread:

```
void _ehs_restore_from_area ( void * newareaPtr);
```

`newareaPtr` should point to the EHS context buffer for the target thread. Once the restore has completed, execution of the target thread can start.

Below is a two-thread example of the process:

```
unsigned int ehs_size=_ehs_save_size();
void *area1 = malloc(ehs_size);
void *area2 = malloc(ehs_size);
_ehs_init_area(area1);
_ehs_init_area(area2);

/* Start Thread 1 */
.
.
.
_ehs_save_to_area(area1);

/* Switch to Thread 2 */
_ehs_restore_from_area(area2);
.
.
.
_ehs_save_to_area(area2);

/* Switch to Thread 1 */
_ehs_restore_from_area(area1);
.
.
.
```

The above example presumes all threads start at an initial state outside of an exception handling region. If a thread starts via a fork operation occurring in an exception handling context, then the example changes as follows:

```
/* Prepare to Fork Thread 2 */
_ehs_save_to_area(area1);
memcpy(area2,area1,ehs_size);
```

These two statements duplicate the current EHS context. From this point forward, the two threads can continue on independent flows of execution and you can use the normal save/restore mechanism outlined above to switch between the threads.

Below is an example of an interrupt and a main execution flow:

```
// Main Start:

unsigned int ehs_size=_ehs_save_size();
void *area1 = malloc(ehs_size);
void *area2 = malloc(ehs_size);
_ehs_init_area(area1);
_ehs_init_area(area2);
// Enable Interrupt
.
.
.
// Interrupt Occurs, Entering Interrupt Routine

_ehs_save_to_area(area1);
_ehs_init_area(area2);
_ehs_restore_from_area(area2);

// Interrupt Context Established, Interrupt Body Begins
.
.
.
// End of Interrupt Body

_ehs_restore_from_area(area1);

// End of Interrupt, Main Routine Continues
.
.
.
```


Compiler Output Files 14

The Microtec C and C++ compilers provide the option to have information sent from the compilers in assembly language to aid in debugging a program.

Assembly Source File

The C/C++ compiler produces an assembly source file that is in Microtec Assembler and Motorola compatible format. There are several assembler directives that can optionally pass variable name and data type information from the compiler directly to the XRAY Debugger.

Advantages of Producing an Assembly File

You can examine and optimize the generated assembly code file by hand. Be careful when you modify the assembly code; an incorrect change can produce unpredictable results.

A compiler command line option is available to intermix the high-level source statements as comments with the generated assembly code in the output file. See the section *Specify Format of Listing Files* in Chapter 3, *Using Command Line Options*.

C/C++ variable and line numbers are mapped into symbol names and placed in the output object module as described in the following sections.

Variable Names

The Microtec C/C++ compiler alters the names of external, public, and static variables to avoid possible conflicts. These naming changes are described in the following sections.

External and Public Variable Names

The compiler alters the names of all variables and functions declared in the C/C++ input source. Variables and functions are prefixed with a leading underscore to avoid possible conflicts with assembler reserved words. The options described in the section *Modifying Naming Conventions* in Chapter 3, *Using Command Line Options*, can change this behavior. These options are not recommended because they prevent access to the standard C/C++ library functions (all library functions have a leading underscore).

For a C or a C++ program to access an assembly language routine or data item, the item name must begin with an underscore and be declared as public in the assembly routine. The C/C++ program should refer to the assembly language item without the underscore.

The following example shows function naming and global data in C and assembly language.

Example:

C Program	Assembly Language
<code>int errflag;</code>	<code>...</code>
<code>extern task();</code>	
	<code>XCOM _errflag,4</code>
<code>driver()</code>	<code>SECTION code,,C</code>
<code>{</code>	<code>XDEF _task</code>
<code>...</code>	<code>...</code>
<code> task (i);</code>	<code>_task:... move.l _errflag,d0</code>
<code>...</code>	<code>...</code>
<code>}</code>	

The compiler reserves some identifiers beginning with two underscores for internal use. Avoid using identifiers with two leading underscores in assembly routines. Also, avoid using identifiers with one leading underscore in C/C++ source files; these identifiers may conflict with internal names.

Some assemblers and linkers translate all lower-case letters to uppercase or vice versa. C and C++ are case-sensitive, so check your assembler and linker documentation for further information on this topic.

The Microtec C/C++ compiler and assembler toolkits retain the original upper- and lower-case status of all letters within a symbol.

Static Variable Names

The compiler automatically prefixes a string to all static variable names declared within a function. The string is `_.Sn_`, where *n* is a number. For example, if the variable name *var* has been declared within a function, it will be changed to `_.Sn_var` in the generated assembly language code.

Line Numbers

If you specify the command line option for debugging, the compiler generates actual C or C++ source lines and line numbers and includes them as comments in the assembly language source file.

If the **-GI** option is specified, the compiler generates line number symbols as labels for debugging purposes. These line number symbols have the form **LLnn**, where *nn* is the line number. Line number information is generated for all executable lines of code.

Code and Data Sections

See Chapter 12, *Run-Time Organization*, in this manual for information on section names, placement, and method of addressing.

Compiler Output Listing

The compiler can generate an output listing file when you use the appropriate command line option (see the section *Generate Listing File* in Chapter 3, *Using Command Line Options*, for more information). When a listing is generated, the compiler assigns a line number, beginning with number 1, to each source line in your C or C++ module. When files are included in the C/C++ source module, the compiler will also assign a number, beginning with number 1, to the include file lines. When the last line of the include file has been numbered, the compiler resumes numbering the source file lines from the line prior to the included file.

The compiler output lists error messages if any occurred during compilation. An arrow points to the error in the listing. The error message is printed immediately below the line where the error was encountered.

The output listing also includes a code summary for each routine and the entire module. These code sizes are estimates since it is not known at compile time whether certain address references will require 16 or 32 bits.

Appendices

Error Statements: Appendix A

This appendix lists the messages produced by Microtec compilers. A description of the error and an example of an error are provided whenever possible.

Message Format

Messages have the following format:

```
"filename", line line_num pos pos_num; (severity) [msg_num] msg  
<line of source code with a circumflex pointing to the error position>  
      ^
```

where *filename* is the name of your file. The location of the error in that file is indicated by the line number (*number*) and column position (*pos_number*). The message severity, described in the next section, is shown in parentheses. The message number is only available in the C++ compiler error messages. It is followed by the error message itself. The offending line of source code is shown on the next line. A circumflex (^) points to the approximate error position.

Note

This appendix contains explanations of error messages that can occur during the typical operation of this product. However, messages that are outside the scope of this product (such as operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for this product for additional information on error messages.

If you encounter an undocumented error message, please contact Mentor Graphics Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

By default, error messages are generated with both numbers and offending source lines. To suppress error message numbers, use the **-Qfn** option when compiling. To suppress offending source lines, use the **-Qfs** option.

Your compiler may not generate all of the error messages listed in this appendix. Note that the examples in this appendix do not show the filename, line number, position number, or message number.

Message Severity Levels

There are two classes of errors that can occur during linker program execution. The first class is fatal, and processing is abandoned. The second class is nonfatal, and processing proceeds after the error is reported.

Messages are followed with a letter in parentheses indicating the severity type of the message. Table A-1 lists the message severity levels.

Table A-1. Message Severity Levels

Type	Severity	Meaning
(F)	Fatal	Always fatal. Processing aborts. Fatal errors arise from conditions that make further compilation impossible, such as missing source and include files or premature End-Of-File . Most often, these errors can be corrected by use of a more complete search path or a quick examination of the files and options used to invoke the compiler.
(E)	Error	Usually fatal, but processing continues for diagnostic purposes. In general, an error message indicates that the compiler has encountered a syntax or logical error. By default, the compiler adopts the view that it should not generate code after an error (E) occurs.
(W)	Warning	Not fatal. This message indicates code that is syntactically correct, but can result in unexpected program behavior.
(I)	Informational	Informative and often appears in conjunction with another error message. An informational message is similar to a warning message but is less likely to alter user program behavior. By default, informational messages are suppressed but can be displayed by the appropriate compiler option (see the section <i>Suppressing Error Messages</i> in this chapter).

Error Position Marker

For any line of code containing an error, the compiler puts a circumflex (^) either directly under the offending element or as close to the problem as possible.

Example:

```
1      main()
2      {
3          int i
4          printf("hello, world\n");
           ^
>> (E) syntax error; unexpected symbol: "printf"
5      }
```

This example shows how the omission of a semicolon on line 3 generates an error message with the error position marker pointing to the first statement after the error occurred.

Suppressing Error Messages

You can suppress error messages of a certain severity level and below using the compiler options shown in Table A-2. Fatal errors cannot be suppressed.

Table A-2. Using Compiler Options to Suppress Diagnostic Messages

Option	Suppresses
-Qi (default)	(I)
-Qw	(I), (W)
-Qe	(I), (W), (E)
-QA	(C++ only) all information
-Qs	summary
-Qmsgnum	(C++ only) message with the indicated message number

By default, only informational messages are suppressed. There is a limit to the number of (E) messages that can be produced in every compilation. This limit is 20 by default but can be changed with the **-Qnumber** compiler option. By setting *number* to zero (0), compilation stops if any (E) errors are detected.

Example:

```

1      array[100];
      ^
>> (I) [ANSI] missing type; "int" assumed
2      int *if;
      ^^
>> (E) syntax error; unexpected symbol: "if"
>> (F) too many errors

2 Errors      1 Informational
```

Messages of type (W) and (I) are not considered in the error count.

Example:

```

1      foo()
2      {
3          undefined[*undefined](undefined);
              ^
>> (E) undeclared identifier "undefined"
4      }

1 Error

```

Although this example shows several uses of the term `undefined` (all of them errors), the compiler avoids multiple error messages and simply identifies the primary error rather than print multiple error messages regarding the same error. The final error count is shown in a summary message at the end of the listing.

The next example shows the results when the summary message is not suppressed, warning and informational messages are suppressed, and the error message limit is set to 5. The compiler options are as follows:

```
-nQs -Qw -Q5
```

Example:

```

1      array[100];
2      int *p=7;
3      int vet[10;
              ^
>> (E) syntax error; unexpected symbol: ";"
1 Error  1 Warning (suppressed) 1 Informational (suppressed)

```

The next example shows the results when the summary message is suppressed, warning and informational messages are suppressed, and the error message limit is set to 0. The compiler options are as follows:

```
-Qs -Qw -Q0
```

Example:

```

1      array[100];
2      int *p=7;
3      int vet[10;
              ^^
>> (E) syntax error; unexpected symbol: ";"
>> (F) too many errors

```

The occurrence of any error terminates compilation.

Reporting Problems

Mentor Graphics performs extensive tests on its compilers. However, if you do find a problem, you should first check the *Release Notes* to see if it contains any useful information. If there is no mention of your problem in the *Release Notes*, please report your problem promptly to Mentor Graphics Technical Support. If possible, have a test case prepared.

Preparing a Test Case

A small test case that reproduces your problem can help Mentor Graphics Technical Support find a solution quickly. Unfortunately, the complexity of a program can make it difficult to isolate a problem and produce a clean test case.

Therefore, it is important that you provide complete information to Mentor Graphics Technical Support.

Calling Technical Support

Follow these steps:

1. Prepare a test case source file that exhibits the problem when compiled.
2. Add comments to your source file to indicate where the problem occurs.
3. Verify that the problem is still there.
4. Call Mentor Graphics Technical Support. The Technical Support representative will make arrangements for you to deliver the test case source file.

C Compiler Error Messages

The messages are listed alphabetically according to the following rules:

1. The message severity is placed at the end of the message to help you scan alphabetically for the message you want.
2. Messages are listed based on the first significant word of the message (words like “a,” “an,” “the,” and so forth are not considered significant) and punctuation is ignored for alphabetizing purposes.
3. If the first word is a variable that is supplied at the time of the message, the message is indexed by the next significant word. For example, the message “*name* undefined” would be listed alphabetically under “undefined.”

4. Messages preceded with the word **[ANSI]** indicate that an error is related to an ANSI feature. These messages are listed based on the text of the message.

Following are the various possible error messages that the compiler can return, along with an example and short explanation of each error.

does not appear before a parameter; ignored, (W)

The **#** operator does not appear before a parameter. This message is from the preprocessor.

Example:

```
#define g(b) # y
```

has no left argument; ignored, (W)

The **##** operator has no left-hand argument. This message is from the preprocessor.

Example:

```
#define f(a) ## a
```

has no right argument; ignored, (W)

The **##** operator has no right-hand argument. This message is from the preprocessor.

Example:

```
#define f(a) a ##
```

& operator has no effect, (W)

The address operator is redundant in the expression. This usually occurs whenever there is an assignment of a function pointer, since the compiler automatically assigns the address of the function.

Example:

```
int foo() { int (*p)(); f(); p = &f; }
```

/* inside a comment, (I)

Comments cannot be nested. This informational message indicates that you are trying to nest comments; this occurs when the **-Qi** compiler command line option to allow informational messages is set. This message is from the preprocessor.

Example:

```
/* /* inner */ */
```

address arithmetic performed with "int" precision, (W)

A **long** value is used as a subscript or added to a pointer. The value is truncated to 16 bits of precision. This warning applies to processors where **sizeof(int) < sizeof(long)**.

Example:

```
int a[100];  
long i;  
a[i]=0;
```

address expected in initialization of *variable*, (W)

Whenever a pointer is initialized by a nonpointer value that is a character or an integer constant, the compiler generates the initializing assignment and issues this warning.

Example:

```
int *p = 0xFFFF;
```

address/pointer extended, (I)

The size of a pointer or an address has been extended. This message is an informational message produced only if the extra checking option is specified on the command line (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#pragma option -v -nQ  
(long) &c; /* informational message issued. */
```

This statement attempts to extend the size of the address of variable *c* to the size of **long**.

adjacent string literal concatenation conflicts with "no ansi" option, (W)

The concatenation of adjacent strings is an ANSI feature. Your file has been compiled with ANSI features disabled (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
char *str = "abc" "cdf";
```

aggregate initialization conflicts with "no ansi" option, (W)

Aggregate initializations are ANSI extensions. Your file has been compiled with the ANSI features disabled, but the compiler found a function with an aggregate initialization (see the **-A** option in Chapter 3, *Using Command Line Options* for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
struct S { int i; int j; } s0 = {1, 2};
struct S s1 = s0;
```

always false, (W)

If an expression will always evaluate to false, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#pragma option -v
main(i)
{
    if (sizeof(int)==sizeof(char));    /* always false */
}
```

always true, (W)

If an expression will always evaluate to true, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options* for further information).

Example:

```
#pragma option -v
main(i)
{
    if (sizeof(int)==sizeof(int *));    /* always true */
}
```

[ANSI] argument is incompatible with formal parameter type, (W)

Whenever the arguments passed to a function do not match the function prototype specification (for example, passing pointers when integers are expected and vice versa), a warning is issued to highlight the type incompatibility.

Example:

```
int foo(int i);
int baz(int *p);

int bar() { int *v; foo(v); baz(100); }
```

argument is void, (E)

The compiler attempts to apply the C type conversion rules whenever it finds function arguments that do not match the function prototype. In K & R C, no function prototypes are available, so the compiler applies default conversions to type **int**. However, when a function returning a **void** is used as an argument, the compiler cannot make any conversions; hence, the compiler generates an error that the argument is of type **void**.

Example:

```
void foo();
void zap();

int bar() { int i; zap(foo()); }
```

arithmetic type expected, (E)

The compiler checks to see if the expression that is evaluated contains types on which arithmetic operations can be performed, such as **char**, **int**, **long**, **double**, and **float**. Only addition and subtraction operations are defined for pointers. Other operations on pointers, like unary operations, multiplication, division, and so forth, are incorrect.

Example:

```
int foo() { int *r = 0; int k; k = (int)(r*4); }
```

array has invalid null dimension, (E)

A function parameter or local variable was declared as an array with the null array dimension.

Example:

```
void f(int ia[][]) /* error */
{
    int arr[][]; /* error */
}
```

assembler inlining function incompatible with binary object output, (E)

This message indicates that the **asm** pseudofunction cannot be used with binary object output. The usage of **asm** implies an assembly step.

Example:

```
f() { asm(); }
```

assembler inlining function interpreted as user function due to "no MRI extension" option, (I)

ASM directives are only supported if the Microtec Compiler extensions option flag is on. This is a default option. Negate this option to compile the file with no Microtec Compiler extensions (see the **-x** option in section *Enabling Microtec Compiler Extensions* in Chapter 3, *Using Command Line Options*, for further information about disabling Microtec Compiler extensions).

Example:

```
#pragma option -nx
int foo() { ASM("4"); }
```

assignment to array or function, (E)

In most cases, the compiler treats an array as if it were equivalent to a pointer; however, it does not permit an assignment to an array.

Example:

```
int foo() { int a[10]; int *p; p = a; a = p; }
```

[ANSI] attempt to modify a "const," (E)

A variable declared with the **const** qualifier was used as an lvalue, thus there was an attempt to modify the **const** variable.

Example:

```
int foo() { const int x; x=5; }
```

auto allocation for XXXX exceeds 32K, (E)

The total size of local variables for function XXXX is larger than 32K. This limitation is imposed by the addressing of the frame pointer with a negative offset.

To resolve this problem, reduce the size of local variables by declaring the non-recursive variables using the **static** keyword.

break statement is not inside a loop or switch, (E)

A **break** statement has appeared outside of a **while**, **for**, **do**, or **switch** statement.

Example:

```
int foo() { break; }
```

call of a nonfunction, (E)

A function name is treated as a pointer to the function of its return type. The compiler allows calls to be made only if it is a pointer to a function.

Example:

```
int foo() {void *vvar = 0; void (*vfunc)(); vfunc(); vvar();}
```

can't open file *filename*, (F)

The compiler cannot open the named file, either because the input file does not exist in the specified directory or, if the file exists, because the compiler encountered an access error. On UNIX systems, the error is due to an incorrect read permission on the file.

can't open *filename* for read, (E)

filename does not exist.

can't open *filename* for write, (E)

The code generator cannot open the *filename* for output (source or mixed assembled listing or object file). The most common reason is that you are out of directory entries or disk space (Windows) or do not have write permission (UNIX).

can't open #include file *file*; directive ignored, (W)

The specified include file *file* cannot be opened. This message is from the preprocessor.

Example:

```
#include "XYZ:FILE"
```

can't open output file *file*, (F)

If you invoke the preprocessor on the command line and direct the output to a file that the preprocessor cannot write to, you will get this fatal error message.

Example:

If you invoke the preprocessor, specify `/ouch.i` as the output file, and you do not have write permission to the `/` directory, you will get the following error message:

```
(F) can't open output file /ouch.i
```

can't open source file *file*, (F)

This message is from the preprocessor.

Example:

If you invoke the preprocessor, specify **ab.c** as the input file, and **ab.c** does not exist, you will get the following error message:

```
(F) can't open source file ab.c
```

can't write to output file: "*filename*", (F)

The compiler cannot write an intermediate, temporary file. Check to see if you have the following conditions:

- Filenames such as *file.1* or *file.2* where *file* is the name of the input source file
- Enough space in the temporary directory for the temporary files

cannot exec *code_generator*: error *nn*, (E)

When the global optimizer runs out of memory, the code generator tries to invoke itself to compile the program again without global optimization. This error occurs when the code generator is unable to invoke itself. Refer to **error.h** for an explanation of the error code *nn*.

cannot negate option *option_name*, (W)

A command line option that does not have a negative form was specified on the command line with a negative form.

Example:

```
mcc68k -nFlp test.c
```

cannot reserve more than *number* register(s) of class *class*, (W)

The compiler will not allow you to reserve more than *number* registers with command line options. See the section *Producing Minor Code Generation Variations* in Chapter 3, *Using Command Line Options*, for more information.

cannot reserve *reg_name*, (W)

The compiler cannot reserve register *reg_name* because it is required to adhere to function calling conventions.

cannot shorten relocatable to less than *x* bytes, (W)

No pointer can be shortened to less than the size of the smallest available pointer. *x* represents the minimum number of bytes.

Example:

```
int near a;
main (void) { (char) &a; }
```

case label is not inside a switch statement, (E)

case is a reserved keyword and can only appear within an enclosing **switch** statement.

Example:

```
int foo() { int i; case 0: i = 1; }
```

cast of a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, and so forth. A variable of **struct** type is not a scalar type and hence cannot be an rvalue (right hand assignment) in a cast expression.

Example:

```
struct tag { int i; } s;
int foo() { int i; i = (int)s; }
```

cast of void type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion.

These conversions can only occur on scalar types like integers, pointers, and so forth. A **void** type cannot be used in a cast expression.

Example:

```
int foo() { double x; x = (double)(void)0; }
```

cast to a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, and so forth. A variable of **struct** type is not a scalar type and hence cannot be an lvalue (left hand of assignment) in a cast expression.

Example:

```
struct tag { int i; } s; int foo() { int i; s = (struct tag)i; }
```

[ANSI] cast used as an lvalue, (E)

Casts that would potentially change the underlying type of lvalues (the left hand of an assignment) may lead to undefined behavior. Therefore, the compiler generates a warning if your code is compiled with the strict ANSI option (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -A -nx  
int foo() { unsigned short ch; (signed int)ch = -1; }
```

Compilation aborted, (E)

You hit **CTRL-C** during compilation.

compilation terminated: reason, (F)

This error is caused by an internal problem with the compiler. Please contact Mentor Graphics Technical Support with your test case and the error message that was generated.

compiler and input hopelessly out of sync, (F)

An unrecoverable syntax error was encountered. The error was severe enough to prevent the compiler from parsing the rest of the program. Correct the error and recompile.

constant *constant* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer specified exceeds the machine capacity, the compiler generates a warning.

Example:

```
int i = 0x100000000;
```

constant expression expected, (E)

The expression associated with a **case** label in a **switch** statement needs to be a constant expression that can be evaluated at compile time. The compiler does constant folding and evaluates the expression during compile time.

Example:

```
switch(i-j);
```

constant in string *string* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer within a string exceeds the machine capacity, the compiler generates a warning.

Example:

```
char *p = "str\x1000000000";
```

continue statement is not inside a loop, (E)

A **continue** statement has appeared outside of a **while**, **for**, or **do** statement.

Example:

```
int foo() { continue; }
```

data allocation for XXX exceeds 64K, (E)

The size of an aggregate is larger than 64K.

data allocation for XXX segment exceeds 64K, (E)

The total size of data allocated for the particular segment is larger than 64K. Force the large aggregates to far segments by explicitly using the **far** keyword or specifying the big data option (**-Zb**). XXX is one of the following: **bss**, **const**, or **data**.

Example

```
char aa[40000];  
char bb[40000];
```

default label is not inside a switch statement, (E)

default is a reserved keyword and can only appear within an enclosing **switch**.

Example:

```
int foo() { default: ; }
```

#define or #undef of identifier ignored, (W)

A **#define** or **#undef** statement is ignored. The preprocessor ignores the following **#undefs**:

```
__DATE__  
defined  
__FILE__  
__LINE__  
__STDC__  
__TIME__
```

Example:

The preprocessor ignores the following **#define** macro defined:

```
#define defined
```

division by 0; result assumed 0, (W)

A division by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1/0
```

division by zero, (W)

Division by zero in many computer architectures results in an arithmetic exception. However, if undetected, division by zero errors may lead to undefined execution behavior. The compiler tries to detect if the divisor in a divide operation is zero; if so, it generates a warning.

Example:

```
int foo() { int i; i = 4/0; }
```

duplicate case label *number*, (E)

The constant expression associated with a **case** label should evaluate to unique constant integer values. Any duplication of **case** label expressions is reported by the compiler.

Example:

```
int foo()
{
    int i;
    i = bar();
    switch (i) { case 1: break; case 3-2: break; }
}
```

duplicate default label, (E)

There can only be one default label associated with a **switch** statement. Any duplication of default labels is reported by the compiler.

Example:

```
int foo()
{
    int i;
    i = bar();
    switch (i) { default: break; default: break; }
}
```

duplicate statement label *label*, (E)

Statement labels should be unique within a function scope. Any duplication of a statement label is reported by the compiler.

Example:

```
int foo()
{
    int i;
    i = bar();
    a : if (i == 0) {
        a: ;
    }
}

int bar()
{
    int r; a:;
}
```

#elif used outside #if/#endif pair; ignored, (W)

An **#elif** statement does not occur within a **#if/#endif** sequence. This message is from the preprocessor.

Example:

```
#elif 1
```

#else used outside #if/#endif pair; ignored, (W)

An **#else** statement does not occur within a **#if/#endif** sequence. This message is from the preprocessor.

Example:

```
#ifdef __STDC__
    volatile int a;
#endif
#else
```

empty character literal, (W)

Empty character literal symbols were used. This message is from the preprocessor.

Example:

```
int a = '';
```

[E] empty conditional statement

This warning message is reported when the **-v** option for extra checks is enabled and there is an **if** statement with a null statement in the conditional part.

Example:

```
main(){  
    if (1) ;  
}
```

[ANSI] empty declaration, (W)

A declaration is missing an identifier. Fix the error by declaring an identifier.

Example:

```
int;
```

empty hexadecimal constant; "0x0" assumed, (W)

The hexadecimal prefix appears without an associated number. This message is from the preprocessor.

Example:

```
int a = 0x;
```

[ANSI] empty hexadecimal sequence; 'x' assumed, (W)

The escape sequence `\x` is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with the ANSI features disabled (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
int foo() { char p; p = '\x'; }
```

#endif does not have a matching #if; ignored, (W)

A **#endif** statement does not have a matching **#if** statement. This message is from the preprocessor.

Example:

```
#if __STDC__
    volatile int a;
#endif
#endif
```

enum value is not a valid constant, (E)

Only constant expressions can be used to assign enumerator values. The compiler performs constant folding and evaluates the expression during compile time.

Example:

```
int foo() { int x = 1; enum { a =0, b = x }; }
```

[ANSI] enumerated data type in bit-field declaration; accepted, (W)

An enumerator was used in a bit field specification. Bit fields are specified for integers only.

Example:

```
#pragma option -nx
struct S { enum color e1 : 3; };
```

#error: *message*, (F)

The **#error** directive is used. This directive causes the ANSI C-compliant preprocessor to issue a diagnostic message that includes the tokens specified in the **#error** directive. This message is from the preprocessor.

Example:

```
#error oops
```


[ANSI] escape sequence '\character' reserved for future use; accepted as 'character,' (I)

The escape sequence `\character` is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with ANSI features disabled (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
int foo() { char *p; p = "\s"; }
```

extra files specified; remainder ignored, (W)

The preprocessor was invoked with an extra (third) file specified on the command line.

[ANSI] extra parameter(s) following a void prototype, (E)

This is a function prototype specification error. Only function pointers of type **void** can be passed as function arguments. Fix the error by including a parameter name for the **void**.

Example:

```
int foo(void , int );
```

far pointer/address truncated to near pointer/address, (I)

When you convert a far pointer to a type that is only large enough to hold a near pointer, the compiler will convert the far pointer to **near**. The **-nQ** option must be activated to trigger this message.

Example:

```
int far F();  
int a = (int) F;  
(near *a2)() = F;
```

fewer arguments than formats, (W)

A format string has fewer arguments than the number of parameters passed to the I/O statement. The message is only available under the **-v** option for extra checks.

Example:

```
printf("%d %d %d %d", 1, 2);
```

file not found *filename*

The compiler cannot find the specified file.

floating-point overflow/underflow, (E)

This message is issued if a floating-point value is outside of the floating-point range.

format/argument mismatch, (W)

A format string does not match the type of the corresponding I/O argument. This message will be a warning (**W**) if the I/O statement is likely to produce the wrong result; the message will be informational (**I**) if the I/O statement would probably produce the correct result but would print a wrong result on another target. This message is only available under the **-v** option for extra checks, and the informational messages are suppressed unless you enable them with the **-nQ** option.

Example:

```
printf("%f, 1");  
printf("%d, 1L");
```

function *identifier* declared as *keyword*, (E)

The **auto** and **register** keywords apply to variables and specify a location. They cannot be used to specify the return type in a function declaration nor in its parameter types.

Examples:

```
int foo() { auto int g(); }
```

function *function* has incomplete return type, (E)

A function can only return with a complete return type although the return statement is not explicitly stated. The compiler tries to return the correct type but finds the type has not been fully specified.

Example:

```
struct S foo() { }
```

function *function_name* is redefined, (E)

A function can be defined only once although it can be declared a number of times. In C, the file is the separate compilation module; therefore, only the linker can report if a function has been defined in more than one file.

Example:

```
int bar();  
int bar();  
int foo() { int i; i = 0; }  
int foo() { int i; i = 0; }
```

GOTO into a LOOP, (W)

This message is produced if a **GOTO** statement causes program execution to branch into the body of a loop. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
goto_into_loop()  
{  
  int i;  
  
  if (x)  
    goto L3;  
  for (i=0; i<=4; i++)  
  {  
    xxx();  
L3:  
    yyy(i);  
  }  
}
```

GOTO into a SWITCH statement, (W)

This message is produced if a **GOTO** statement causes program execution to branch into the body of a **SWITCH** statement. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
goto_into_switch(  
{  
    if (x)  
        goto L3;  
    switch(x=test())  
    {  
    case 1: test(x); break;  
    case 2: xxx(x); break;  
L3:  
    case 3: yyy(x); break;  
    case 4: zzz(x); break;  
    }  
}
```

identifier *identifier* declared as void, (E)

A formal parameter was incorrectly declared to be of type **void**.

Example:

```
int foo(int x, void y);
```

identifier expected; assumed undefined, (W)

#ifdef should be supplied with an identifier as a parameter. If it is incorrectly supplied, the **#ifdef** construct is assumed to be undefined; hence, the enclosing code between the **#ifdef** and its corresponding **#endif** is left out. This message is from the preprocessor.

Example:

```
#ifdef 1  
int i;  
#endif
```

#if does not have a matching #endif; supplied, (W)

A **#if** or a **#ifdef** should have a corresponding **#endif** construct. This message is from the preprocessor.

Example:

```
#if 1
int i;
```

IF expression evaluated to constant *number*, (W)

If an **IF** expression will always evaluate to a constant, this warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#define FLAG 0

if_exp_evaluated_to_const()
{
    int i = FLAG;

        if (i)          /* i == 0 */
            test();
}
```

In this example, the **IF** expression evaluates to a constant 0.

illegal argument to predefined macro *macroname*, (W)

The predefined preprocessor macro does not accept the argument supplied. This message is from the preprocessor. Refer to Chapter 3, *Using Command Line Options*, of this manual for more information.

Example:

```
__OPTION_AVAIL (a b c d e f g);
```

illegal char in macro definition; symbol *symbol* deleted, (W)

A comma was expected; the unexpected symbol was deleted to correct the situation. In the following example, **f(a)** and **f(a;)** both yield **abc**. This message is from the preprocessor.

Example:

```
#define f(a;) abc
```

ill-formed number, (E)

A preprocessing number is not a valid number.

Example:

```
int x = 1A;
```

incompatible type returned, (E) or (W)

The type returned is incompatible with the declared type of the function.

Example:

```
int foo() { char *p = "abc"; return p; }
struct tag { int i; } bar() { return 0; }
```

inconsistent packing attributes for tag "*tag*"; "packed" prevails, (W)

A structure has been declared as both packed and unpacked.

Example:

```
struct tag;
packed struct tag;
```

inconsistent packing attributes for tag "*tag*"; size would change from *size1* to *size2*, (E)

A structure has been declared inconsistently. The conflicting sizes are shown in bytes.

Example:

```
struct tag {char a; short sh;};
packed struct tag {char a; short sh;};
```

inconsistent redeclaration of *identifier*, (E)

An identifier of a declared type cannot be redeclared to be of a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
int x; char x;
```

This error will also occur if a function prototype uses a structure that has not been previously declared. The scope of the structure terminates at the end of the prototype definition, and a later definition of the structure is interpreted as a redefinition of that structure. To avoid this problem, declare the structure prior to the function prototype.

Example:

```
extern void f(struct s);
...
struct s{           /* interpreted as redefinition of s */
    ...
}
```

inconsistent redeclaration of tag *tag*, (E)

A structure tag of a declared type cannot be redeclared to be a structure tag for a different structure. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S { int x; }; struct S { char ch; };
```

inconsistent use of tag *tag*, (E)

A tag of a declared type cannot be used as a tag for a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S ; enum S;
```

increment/decrement of array or function, (E)

Pointer arithmetic operations cannot be performed on an array or function pointer.

Example:

```
int foo() { int a[10]; a++; }
```

inefficient alignment (offset for member "*variable*" not a multiple of *number*), (W)

A misaligned member of a packed struct will be loaded and stored inefficiently. The access will generally be “burst” into a byte-by-byte or word-by-word load or store. On the 68000 and 5200, the bursting is done by the compiler, but on the 68020 and other machines than support misaligned access, the chip handles the bursting transparently. In either case, the burst access is less efficient than an aligned access.

Note that this warning is issued only when parsing a declaration of a packed struct and is not issued in all cases of misaligned access. For instance, the 68000 compiler bursts access to all fields of an element of an array of packed structs (since the alignment of any element is generally unknown), but the warning is not issued.

[ANSI] inefficient evaluation in "double" because of "double" constant, (I)

Whenever double constants are used instead of integer constants, the compiler generates an informational message to indicate that the conversion is an inefficient one.

Example:

```
int foo() { float f1 = 0; f1 = f1 + 1.0; }
```

int expected in initialization of *variable*, (W)

The initializer supplied in an initializer list should be type compatible with the variable it is expected to initialize.

Example:

```
int i = &i;
```

int overflow for "*number*," (W)

This message is issued if an integral value outside of the integer range is supplied.

Example:

```
void f(void) {  
    0100000000000;  
    4573741824;  
    0xFF000000;  
}
```

int overflow in string "*string*", (W)

This message is issued if you supply an integral value in a quoted string, and the integral value is outside of the integer range.

Example:

```
void f(void) {  
    "string\xaaaaaaaaaaaaa";  
}
```

int type expected, (E)

The compiler expects an **int** type or a type that can be implicitly converted to an integer.

Example:

```
int foo() { int i = ~1.0; }
```

int value expected; real value truncated, (W)

Whenever an lvalue is of type **int** and the rvalue is of type **float** or of an arithmetic type that is larger than the **sizeof(int)**, the compiler generates a warning that the rvalue is being truncated to suit the **sizeof** the integer. This truncation can lead to unexpected run-time behavior.

Examples:

```
int foo() { int array[3.4]; }  
int foo() { int *p; float q; p+q; }
```

integer line number expected; directive ignored, (W)

A line directive must contain a reference to a line number. This message is from the preprocessor.

Example:

```
#line =  
int i;
```

Internal Compiler Error: *reason*, (F)

This error is caused by an internal problem with the compiler. Please contact Mentor Graphics Technical Support with your test case.

internal error: *reason*, (F)

This error is caused by an internal problem with the compiler. Please contact Mentor Graphics Technical Support with your test case.

interrupt function has parameter(s), (W)

An interrupt function cannot have parameters. The compiler will generate a warning whenever it detects such a condition.

Example:

```
void interrupt foo(int i);
```

interrupt function returns non-void type, (W)

An interrupt function must always return a **void**. The compiler will generate a warning whenever it detects a return type other than **void**.

Example:

```
int interrupt foo();
```

interrupt procs not supported, (W)

The compiler does not support declaring functions as interrupt handlers with the **interrupt** keyword in this release.

Example:

```
int interrupt foo();
```

invalid argument for assembler inlining function, (E)

An invalid argument was used with the **asm** pseudofunction.

Example:

```
f() { asm(3); }
```

invalid array component type, (E)

An array cannot be of type **void**.

Example:

```
void array[1000];
```

invalid array size, (E)

The size of an array should be a positive integer and the result of a constant expression. The compiler evaluates the expression and checks to see that it is a positive integer. Any incorrect size specification will result in this error.

Example:

```
int array[-1000];
```

invalid digit in octal constant *constant*; assumed decimal, (W)

In C, all octal numbers are identified by a leading zero and must contain only octal digits. This message is from the preprocessor.

Example:

```
int a = 08;
```

invalid #include file name *symbol*; directive ignored, (W)

An unacceptable filename has been chosen for an include file. This message is from the preprocessor.

Example:

```
#include 123
int i;
```

invalid initialization for *identifier*, (E)

The initialized variable requires an initializer of the same type. The compiler attempts to convert the types if a type conversion is possible. Otherwise, the compiler reports an error that the initialized variable has been supplied with an invalid initializer.

Example:

```
void (*p)() = 0;
int a[10] = p;
```

invalid insert *symbol* for assembler inlining function, (E)

An **ASM** instruction cannot contain references to undeclared variables in the C program. An **ASM** instruction should also be syntactically correct. If the variable is undeclared or a syntax error occurs within the **ASM** string, the compiler generates an error message.

Example:

```
int foo() { ASM("mov `undef`, `truncat"); };
```

invalid return type, (E)

The function return type cannot be an array. The compiler will generate an error whenever it detects this condition.

Example:

```
typedef int A[10];
A foo();
```

invalid size for bit-field *bit-field_name*, (E)

Bit field members cannot specify a field size greater than the size of the type to which the bit field belongs.

Example:

```
struct S { unsigned int a : 100; unsigned int b; };

/*
struct S1 { unsigned int a : sizeof(unsigned int) + 1;
unsigned int b; };
*/
```

[ANSI] invalid specifier(s) used with "void", (E)

No additional type specifiers are used along with the type **void** except for **const** and **volatile**. The compiler generates an error (ANSI or otherwise) if it detects such a condition.

Example:

```
unsigned void* a;
```

invalid storage class *storage_class* for parameter *parameter*, (E)

Function parameters are formals and cannot be associated with any storage specifications.

Example:

```
int f(static int i);
```

invalid storage class *storage_class* for parameter *parameter*, (W)

An invalid automatic storage class is being specified for a formal parameter.

Example:

```
int foo(x) auto x; { }
```

invalid suboption "*sub_option_name*" given for *option_name*, (W)

The compiler does not recognize the suboption entered on the command line.

Example:

```
-Fz
```

invalid type for member *member_name*, (E)

The members of a structure should be a valid type. Valid types include the built-in C types and any user-defined **structs**, **unions**, or **enums**. Members cannot be functions or of type **void**.

Example:

```
struct S { void b; int i; };
```

invalid value "*value*" given for *option_name*, (W)

An invalid value was given for a compiler command line option.

Example:

If you specify **-Za3** on the command line, you will receive the message:

```
(W) invalid value "3" given for -Za
```

***identifier* is not a parameter, (E)**

The parameter names in a function definition should match those in their declaration. The compiler detects an error if the match fails.

Example:

```
int foo(a) int b; { int k; }
```

***option* is not possible if there are functions whose total parameter size is > *number*, (W)**

Your file has been compiled with the compiler command line option to reserve a register, but this register is needed in order to adhere to function calling conventions.

option **is not possible if there are functions with n parameters in the file to be compiled, (W)**

Your file has been compiled with the compiler command line option to reserve a register, but this register is needed in order to adhere to function calling conventions.

[ANSI] keyword "*keyword*" in empty declaration, (W)

You cannot specify a storage class or qualifier for a **struct**, **union**, or **enum** tag declaration without declaring an identifier. Fix the error by declaring an identifier for that structure. The keyword **volatile** in an empty declaration causes a fatal error since the structure declared is not considered volatile.

Example:

```
static struct tag {int i; };
```

lexical error -- unexpected *cause*, (E)

An error was detected during the lexical analysis compilation phase. A short description of the lexical error (*cause*) is provided with the message.

local "extern" variable *identifier* is initialized, (E)

External variables declared in a function scope cannot be initialized. The compiler generates an error message when it detects this condition.

Example:

```
int foo() { extern int e = 0; }
```

[ANSI] local function *identifier* declared as "static," (I)

Whenever a function is declared **static** in a function scope, an informational message is generated.

Example:

```
int foo() { static f(); }
```

local variable *variable* never used, (W)

This message is produced when a local variable is never used or can be replaced by a constant. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
var_never_used( )
{
    int i, j;          /* all references of i are eliminated */

        i = 3;          /* value of i is not used; assignment
                        /* eliminated */
        x = i + 2;      /* i replaced by 3 */
}
```

"long double" interpreted as "double," (W)

A declaration with **long** and **double** type specifiers does not increase the precision of the type. Therefore, the compiler generates a warning that these type specifiers are interpreted to be of type **double**. This message is issued if the extra checking compiler command line option is turned on (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#pragma option -v
    long double f;
```

In this example, the data type for `f` is declared as `long double`. The compiler treats its data type as **double**.

"long float" interpreted as "double," (W)

A declaration with **long** and **float** type specifiers does not increase the precision of the type. Therefore, the compiler generates a warning that these type specifiers are interpreted to be of type **double**.

Example:

```
#pragma option -v
    long float f;
```


"long" truncated to "int," (W)

This message is issued when the compiler converts a variable of type **long** to **int**.

Example:

```
void g(void)
{
    int *p;
    long l;
    unsigned long ul;

    p+l;          /* warning */
    p+ul;         /* warning */
    p[l] + 1;     /* warning */
}
```

macro *name* invoked with wrong number of arguments, (E)

A macro has been invoked with an incorrect number of arguments. This message is from the preprocessor.

Example:

```
define f(a) a+1
void main() {
    int k = f(1,2);
    int j = f(1);
}
```

macro name *symbol* is not an identifier; directive ignored, (W)

The name given in a macro is not an identifier. This message is from the preprocessor.

Example:

```
#define 0
#define $6.0-U.S-Dollars
int j;
```

macro parameter *symbol* is not an identifier; ignored, (W)

The parameters of a macro should be identifiers. This message is issued if any of the macro parameters are not identifiers. This message is from the preprocessor.

Example:

```
#define f(0)
void main() {
    f(0);
}
```

meaningless "far" qualifier; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure member name, function name, and so forth, some specifier combinations will not appear meaningful in the given context. The keyword **far** should be applied only to pointers or objects.

meaningless "interrupt" qualifier for *identifier*; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure member name, function name, and so forth, some specifier combinations will not appear meaningful in the given context. The keyword **interrupt** should be applied only to functions.

Example:

```
int interrupt a;
```

meaningless "near" qualifier; ignored, (W)

Although C grammar allows a specifier list to be attached to any identifier, structure member name, function name, and so forth, some specifier combinations will not appear meaningful in the given context. The keyword **near** should be applied only to pointers or objects.

[ANSI] missing definition for static function *identifier*, (E) or (I)

A static function has been declared and used but not defined. This is an error when compiling with the strict ANSI option. If the declared static function is not used and the file has been compiled with the strict ANSI option, an informational message is

generated (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -A -nx
static foo(); int bar() { foo(); };
static zap();
```

[ANSI] missing definition for static function *function_name*; "extern" assumed, (W)

If you call a static function whose definition is missing at the end of the compilation, this warning is issued, and the keyword **extern** is assumed.

Example:

```
static foo(); int bar() { foo(); };
```

missing definition for (unused) static function "*function_name*," (W)

A static function was undefined. This message is issued if the extra checking compiler command line option is turned on (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for more information).

Example:

```
static int f(void);
```

missing exponent in real constant *constant*; E0 assumed, (W)

An exponent has been left off of a real constant. This message is from the preprocessor.

Example:

```
float a = 1.E;
```

missing format specifier, (W)

A format string contains a “%” without a following format specifier. This message is available only with the **-v** option for extra checks.

Example:

```
printf("%");
```

missing #include file name; directive ignored, (W)

The include filename is missing in a **#include** directive. This message is from the preprocessor.

Example:

```
#include
void main() {}
```

[ANSI] missing parameter name, (E)

A function definition in ANSI must contain the complete prototype specification. This error is caused by a missing parameter name.

Example:

```
int foo(int ) { }
```

[ANSI] missing prototype for *function*, (W)

A function definition in ANSI must contain the complete prototype specification. This error is caused by a missing prototype.

Example:

```
#pragma option -A -v
int proto2(int a)
{
    printf("no prototype");
}
```

missing return value, (W)

This warning message is produced when a function declared to return a value does not use the return statement consistently. Since a return statement is not required for any function, the compiler does not warn if no return statements are used. However,

when a return statement is used in a function, the compiler will issue a warning if all uses of the return statement are not consistent. If any return statement returns a value, a warning is issued unless all return statements return values. If a return statement is used within a function, a warning is issued for every function exit point that does not use a return statement. This warning is issued if the extra check option was specified (see the **-v** option in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
int returnless_func()  
{  
    if (x)  
        return(1);  
}  
/* missing return statement at end of  
/* function */
```

missing semicolon for last member, (W)

The last member in a structure or union member list must be terminated by a semicolon (;) in ANSI C. This message is issued if the strict ANSI option is turned on and the Microtec Compiler extensions command line option is turned off. For further information, see the **-A** option regarding setting ANSI-compliant modes and the **-x** option regarding setting Microtec Compiler extensions in section *Command Line Options — Summary* in Chapter 3, *Using Command Line Options*.

Example:

```
#pragma option -A -nx  
struct {  
    char c;  
    int u    /* warning */  
} sobj1;
```

[ANSI] missing type; "int" assumed, (I)

Whenever a declaration is made, the compiler expects a type specifier. However, if no type specifier is given, the compiler defaults the declaration to that of an integer and generates a warning.

Example:

```
#pragma option -nQ  
a;
```

mod by zero, (W)

Division or modulo by zero in many computer architectures results in an arithmetic exception. However, if undetected, it can lead to undefined execution behavior. The compiler tries to detect if the divisor in a divide operation is zero; if so, it generates a warning.

Example:

```
i = i % 0;
```

mod by 0; result assumed 0, (W)

Mod by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1%0
```

module is too large for available memory, (F)

The compiler is usually very efficient about its memory allocation. However, there are certain limiting factors, such as the number of macros defined, and so forth. The compiler generates this fatal message when it cannot allocate any more memory.

more arguments than formats, (W)

A format string has more arguments than the number of parameters passed to the I/O statement. The message is only available with the **-v** option for extra checks.

Example:

```
printf("%d %d", 1, 2, 3, 4);
```

more than 32767 arguments (compiler limitation), (E)

The compiler limits the number of function arguments to 32767. This is an internal compiler restriction.

more than 8192 labels in one switch (compiler limitation), (E)

The compiler limits the number of **case** labels that can be handled in a **switch** statement. It can process a maximum of 8192 **case** labels within this statement. This error message is generated when the limit has been exceeded.

name *name* specified for filler bit-field; ignored, (W)

The bit field specification lets you specify a bit field from 1 to the **sizeof** the type of its bit field. The bit field 0 is the filler bit field. Filler bit fields are used to pad the bit fields to the correct size. These fields cannot be referenced. Bit field 0 is the leading filler bit field. References to it cause the compiler to issue this warning.

Example:

```
struct S { unsigned int a : 0; };
```

"near" and "far" specified through indirection, (E)

A type cannot be qualified with both **far** and **near** keywords.

Example:

```
typedef int far INT;  
INT near a;           /* error */
```

near pointer/address extended to far pointer/address, (I)

When you convert a near pointer to a type that is large enough to hold a far pointer, the compiler will convert the near pointer to **far**, using the normal conversion method for the target processor. The **-nQ** option must be activated to trigger this message.

no member *identifier*, (E)

Whenever a reference is made to a nonexistent member of a structure, the compiler generates an error.

Example:

```
int foo() { struct S { int m; } s0; s0.p = 0; }
```

non-int in bit-field declaration; accepted, (W)

Bit fields are specified for integers only. The compiler accepts some of the bit field specifications but generates a warning.

Example:

```
#pragma option -nx  
struct S { char a : 3; };
```

non-interrupt call of interrupt function, (W)

A non-interrupt call to an interrupt function was made.

Example:

```
interrupt void bar();  
void foo() { bar(); }
```

non-prototype function declaration has formal parameter(s), (E)

A function declaration that does not have a prototype specification cannot have parameter names in its argument list. An error is generated by the compiler whenever it detects this condition.

Example:

```
int foo(a,b);
```

obj file write error, disk probably full, (E)

A system error occurred while writing the object file. The disk is probably full.

offset-only address assigned to long pointer, (I)

An “offset only” is being assigned to a full far pointer (segment + offset). A zero segment will be filled in by the compiler.

Example:

```
int far *PTR;  
int near *ptr;  
PTR = ptr;
```

operand has incomplete type, (E)

A type can be used only after it has been completely specified.

Example:

```
int foo() { struct S s0; s0 = 0; }
```


operand is an address, (E)

An address is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int x,a; int *p; p=&x+&a; }
```

operand is not an lvalue, (E)

The operand on the left of an assignment is not a location or variable that can be modified.

Example:

```
int foo() { int x; 5=x+2; }
```

operand is pointer to an object of unknown size, (E)

A pointer to an object of unknown size is incorrectly being used as an operand.

Example:

```
int foo() { struct tag *p; p++; }
```

operand is pointer to function, (E)

A pointer to a function is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int (*p) (); p++; }
```

operand of & is a bit field, (E)

A bit field is incorrectly being used as an operand of the “address of” operator (&). Fields do not have addresses, so the & operator cannot be applied to them.

Example:

```
int foo() { struct { int bit:1;} s; bar(&s.bit); }
```

operand of & is a register variable, (E)

A register variable is incorrectly being used as an operand of the “address of” operator (&). Register variables do not have addresses, so the & operator cannot be applied to them.

Example:

```
int foo() { register x; bar(&x); }
```

operand of & is not an lvalue, (E)

The operand of the “address of” operator is not legal since it is not a value or a location that can be modified.

Example:

```
int foo() { int x; x=&5; }
```

operands of *operator* have incompatible types, (W)

The operands of an expression are incompatible.

Example:

```
int foo() { int x; if (&x==1) x=7; }
```

option -Md requires runtime initialization of pointers, (W)

This warning indicates that a pointer has been assigned a compile-time initializer that will contain an absolute address.

option *option1* is being phased out; use *option2*, (W)

Some options are no longer used by the compiler. Use of these options does not produce the expected behavior. The compiler will warn about the use of the obsolete option.

Example:

```
#pragma option-68332  
int i;
```

option *option* is not implemented, (W)

The compiler does not recognize the option entered on the command line.

option *option_name* might provoke inefficient stack alignment, (W)

The specified option limits the size of the parameters passed to the stack.

***option_name* option not available, (W)**

The compiler does not recognize the option entered on the command line.

option *option_name* requires an argument, (W)

A command line option *option_name* that requires an argument was supplied without an argument.

Example:

```
mcc68k -I test.c
```

option *option* requires an argument; ignored, (W)

The compiler command line option for generating a listing file has been used without the filename argument. If the filename argument is not specified, the compiler issues this warning message and writes the listing file to standard output (see the **-l** option in section *Producing Listing Files* in Chapter 3, *Using Command Line Options*, for more information about generating a listing file).

options *option1* and *option2* conflict; *option2* ignored, (W)

Two options that conflict with each other have been supplied on the command line. The second (right-most) option is ignored. Refer to Chapter 3, *Using Command Line Options*, for more information about the options.

Out of memory (*address:operation*), (E)

The code generator ran out of memory at location *address* during phase *operation* of code generation. This error should only occur on PC hosts.

To resolve this problem:

- Check to see if the module has extremely large functions (more than two thousand lines per function, excluding comments).
- Check to make sure the PC has at the minimum amount of memory required. If not, try removing some memory-resident programs.

packed arrays may generate invalid access (member "*name*", (W))

A field in an array within a packed structure may not be aligned properly (it is located at an odd address).

Example:

```
packed struct {  
    char a[3];  
    short s[10];  
};
```

packed structs not supported, (W)

The compiler does not support declaring structures as packed with the **packed** keyword in this release.

Example:

```
packed struct S { int i; char c; };
```

[ANSI] parameter declaration outside() in a prototype definition, (E)

A parameter was declared outside of the parentheses in a function prototype declaration.

Example:

```
int foo(int a) int b; { }
```

parameter *identifier* is hidden by redeclaration, (E)

An identifier declared as a parameter has been redeclared as a local variable; this causes a conflict with respect to the retrieval of that identifier in the function scope.

Example:

```
int foo(x) { int x; bar(); }
```

PC-relative branch range exceeded (E)

The compiler generated a branch to an address that is greater than 32K bytes away. This error can occur when a **switch** statement contains several hundred **case** statements. This error can also occur when a call is made to another function within the same module but greater than 32K bytes away when the **-Mcp** option is used.

Calling external functions using PC-relative code causes a similar error to be produced at link time.

pointer or array type expected, (E)

Something other than a pointer or array type variable is incorrectly being used as an operand of unary ***** or **[]**.

Example:

```
int foo() { int x, y; x=*y; }
```

[ANSI] pointer to const becomes unrestricted, (W)

Whenever **const** pointers are passed as arguments to functions that require non-**const** pointers, there is a potential that the called function may change the object pointed to by the **const** pointer. Therefore, the compiler generates a warning whenever your program is compiled with the strict ANSI option.

Example:

```
#pragma option -A
int bar(int* ip);
int foo() { const int* jp; bar(jp); }
```

[ANSI] pointer to volatile becomes unrestricted, (W)

If a pointer to **volatile** data is passed as an argument to a function that requires a pointer to non-**volatile** data, the **volatile** object pointed to may be optimized if the called function is optimized. Since optimizations should not be performed on data declared as **volatile**, the compiler generates a warning when this condition is detected.

Example:

```
#pragma option -A
int bar(int* ip);
int foo() { volatile int* jp; bar(jp); }
```

**potential run-time error: address is not guaranteed to be properly aligned,
(W)**

The item being accessed does not necessary reside on the appropriate boundary. For example, if padding has not been added to elements in a packed structure to force alignment, elements in that structure might not be aligned on an even boundary.

Example:

```
packed struct PAK {
    int i;
};

packed struct C {
    char c;
    packed struct PAK pak;
    char cc;
};

void f() {
    packed struct C a[10];
    int i = 8;
    &a[i].pak;                /* WARNING */
}
```

Since `char c` is a single byte that can occur on an even or odd boundary, there is no guarantee that the packed structure `pak` will occur on an even boundary. If you change the definition of `struct C` to avoid the potential alignment problem, the compiler does not emit a warning message. For example, the compiler will not emit the warning if you define `struct C` as follows:

```
packed struct C {
    packed struct PAK pak;
    char c;
    char cc;
};
```

**potential run-time error: arbitrary expression for pointer requiring multiples
of *number*, (W)**

A value has been assigned to a variable through pointer dereferencing, and the value is not of the same type as the variable. This message is issued if the alignment requirement for the variable to which the pointer is pointing is greater than one.

Example:

```
char c;
double *dp = &c;            /* warning */
```

In this example, assigning `c` to `dp` may cause a run-time error because the alignment for type `double` (that `dp` points to) is greater than 1. To avoid this message, cast `c` to `double *`.

potential run-time error: improper alignment, (W)

This message is issued when the compiler detects a potential misalignment that could cause a run-time error.

Example:

```
void f(void) {  
    short *p=(short *) 7;  
}
```

This example is potentially illegal at run time because even-address alignment is required.

potential run-time error: target has more stringent alignment requirements than source, (W)

This message is issued when the compiler detects a potential misalignment that could cause a run-time error.

Example:

```
void f(void) {  
    char i; double *dp = &i;  
}
```

In this example, assigning `&i` to `dp` may cause a run-time error because the alignment requirement for `double` is greater than the alignment requirement for `char`.

recursive call *recursion* replaced by jump, (I)

This message is produced when a recursive function call is used. The optimizer replaces such function calls with a jump instruction.

Example:

```
tail_recursion()
{
    int i = test();

        if (i)
            return(tail_recursion());/* call to itself
*/
    return(1);
}
```

redeclaration of *identifier*, (E)

Redeclaration of identifiers is not allowed. Two identifiers that have the same name cannot share the same scope.

Example:

```
int foo() { int x; int x; }
```

redeclaration of enum constant *constant*, (E)

An identifier appeared in an enumeration list more than once. Enumeration lists provide a convenient way to associate constant values with names, so an identifier name cannot appear twice.

Example:

```
int foo() { enum {x,y,x}; }
```

redeclaration of member *field_name*, (E)

The elements mentioned in a structure (members) cannot be declared more than once.

Example:

```
int foo() { struct x {int y; int y;}; }
```


redeclaration of parameter *identifier*, (E)

The same parameter cannot be declared more than once in a function definition.

Example:

```
int foo(int x, int x);
```

redefinition of macro *identifier*, (W)

A macro has been defined more than once. All redefinitions (the second definition and the third definition if you supply three definitions for the same macro name) are ignored. This message is from the preprocessor.

Example:

```
#define x y  
#define x z
```

reinitialized variable *variable*, (E)

A variable has been incorrectly initialized more than once.

Example:

```
int x=1; x=2; int foo() { }
```

restriction: aggregates cannot be larger than *value*, (E)

The compiler imposes a restriction on the size of aggregates that can be allocated. This restriction arises due to architectural limitations in the target machine that are based upon the largest accessible piece of contiguous memory. The compiler does not allow an individual aggregate size to increase beyond this limitation. These limitations are fairly large, and it would be rare to reach this limit, especially with 32-bit architectures.

Example:

```
int foo() {  
char a[0xffffffff] ;  
}
```

restriction: bit-field "*field_name*" straddling *number* bytes ; previous field padded (W)

This message is issued when complex bit-field straddling occurs. Simple straddling happens when a bit field of a packed structure straddles a word unit but ends up spanning one word unit. Complex bit-field straddling happens when a bit field of a packed structure is larger than one word and straddles the previous word unit, the word unit with the bulk of the information, and the following word unit.

Example:

```
packed struct S {  
    int a : PART1;  
    int b : PART2;  
};
```

`struct S` has two bit fields: `a` and `b`. Assuming that `PART1` and `PART2` add up to slightly more than one word unit, the two bit fields require at least two words and possibly three (depending on the starting point of `struct S`). For more information on bit fields, refer to section *Alignment of Bit Fields* in Chapter 11, *Internal Data Representation*, in this manual.

restriction: identifier longer than *count* chars, (E)

The maximum identifier length has been exceeded.

restriction: line too long, (E)

The line length of 64K characters has been exceeded.

restriction: string too long, (E)

The length of a string has exceeded the number of bytes allowed by the code generator. Refer to Table 11-1 in Chapter 11, *Internal Data Representation*, in this manual for more information.

restriction: too many arguments (more than *number*), (E)

The compiler imposes a restriction on the number of function arguments.

restriction: too many case labels (more than 8192) in one switch, (E)

This message is issued if you have more than 8192 cases in your **switch** statement.

restriction: too many nested switch statements (more than 16), (E)

This message is issued if you have more than 16 nested **switch** statements in your code.

restriction: too many symbols, (F)

The compiler imposes a restriction on the number of symbols. The number of symbols permitted is based on the amount of string space symbols occupy. The maximum amount of string space the compiler can handle is approximately 255 * 64K bytes.

restriction: too many symbols for intermediate file, (F)

Compilers that do not support multiple string tables have a limit of 64K symbols.

scalar type expected, (E)

A variable of nonscalar type was used where one of scalar type was expected.

Example:

```
int foo() { struct {char x;} s; bar(s+1); }
```

segment-offset address assigned to short pointer, (W)

A full far pointer (segment + offset) is being assigned to an “offset only” pointer. The segment information is lost.

Example:

```
int far *p;  
int near *p;  
p=p;
```

shortening address may lose significance, (I)

The size of a pointer or an address has been shortened. This message is an informational message produced only if the extra checking option is specified on the command line (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#pragma option -v -nQ
void f(void) {
    double x;
    char c;
    (long) &c; /* INFORMATIONAL: address/pointer extended */
    (char) &x; /* INFORMATIONAL: shortening address of &x */
    (char) &c; /* INFORMATIONAL: shortening address of &c */
}
```

In this example, the size of the address of `c` has been extended to the size of `long`, which causes the compiler to generate the message “address/pointer extended.” The next two statements shorten the sizes of the addresses of `x` and `c` to the size of `char`, which causes the compiler to issue the message **shortening address may lose significance**.

[ANSI] "signed" and "unsigned" conflict, (E)

Two contrasting declarations (**signed** and **unsigned**) apply to this identifier. To correct the error, only specify one **signed** or **unsigned** for the declaration.

Example:

```
int foo() { signed unsigned int x; }
```

size of a bit-field is undefined, (E)

The **sizeof** operator cannot be applied to a bit field.

Example:

```
int foo() { struct {int bit:1;} s; int x; x=sizeof(s.bit); }
```

size of a function is undefined, (E)

The **sizeof** operator cannot be applied to a function.

Example:

```
int foo() { typedef bar(); int x; x=sizeof(bar); }
```

size of void is undefined, (E)

The **sizeof** operator cannot be applied to **void** since **sizeof(void)** is undefined.

Example:

```
int bar() { int x; x=sizeof(void); }
void *vp;
struct tag;
foo()
{
    sizeof(void);
    sizeof(struct tag);
    sizeof(*vp);
    vp + 2;
    vp - vp;
}
```

sizeof applied to an incomplete type, (E)

The **sizeof** operator was applied to an identifier of an incomplete type. This unary operator is used to compute the size of any object that has a known size; in this case, the size is unknown.

Example:

```
int foo() { struct tag s; int x; x=sizeof(s); }
```

source file contains no declarations, (W)

A source file is expected to contain at least one declaration. The compiler generates a warning if it finds a source file with no declarations.

Example:

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

***specifier* specified twice, (W)**

A type specifier is incorrectly being used twice in a declaration.

Example:

```
int foo() { const int const x; }
```

***specifier* specified twice through indirection; ignored, (W)**

An identifier has been used twice with the same type qualifier (**const** or **volatile**) in separate declarations.

Example:

```
int foo() { typedef const int INT; const INT x; }
```

specifiers *specifier1* and *specifier2* conflict, (E)

Two or more specifiers with conflicting or undefined meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { short char x; }
```

specifiers "short" and "long" conflict, (E)

Two contrasting declarations (**short** and **long**) apply to this identifier; the **short** declaration is always discarded.

Example:

```
int foo() { short long int x; }
```

statement has no effect, (W)

This warning message indicates that a statement has been declared that will have no effect once executed. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
no_side_eff_statement()  
{  
  int *p;  
  int x,y,z;  
  
  *p; /* no effect */  
  p++; /* no effect */  
  if (x == y)  
    x == z; /* no effect */  
}
```

statement not reached, (W)

This message is produced when a statement will never be executed. This warning is usually associated with a conditional statement that is always false. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
#define FLAG 0
int x;

unreached_statement()
{
    if (FLAG && x) /* FLAG is compile time constant 0 */
        x = 3;    /* never reached */
}
```

statement table is too large for available memory

The size of the statement table has exceeded the available memory.

storage class *storage_class* is invalid in file-scope, (E)

The **auto** or **register** keywords were used in a file-scope (external) declaration. The storage classes **auto** and **register** can only be used in a local scope (a function scope). These specifiers are not permissible on file scope variables.

Examples:

```
auto i;

register x;
```

storage classes *storage_class1* and *storage_class2* conflict, (E)

Two or more storage classes with conflicting meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { static auto x; }
```

string too long (compiler limitation), (E)

The compiler cannot accept a string because it is too long.

struct or union type expected, (E)

Something other than a **struct** or **union** type was used with the “.” operator, or something other than a pointer to a **struct** or **union** was used with the “->” operator.

Example:

```
int foo() { int x, y; x=(&y).a; }
```

switch expression evaluated to constant *number*, (W)

This warning message is produced when an expression used in a **switch** statement evaluates to a constant. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
relation_op_evaluated_to_const4()  
{  
  int i = FLAG;  
  
  switch(i+2)  
  {  
    case 1: test1(); break;  
    case 2: test2(); break;  
    case 0: test0(); break;  
  }  
}
```


switch statement has no cases, (W)

The **switch** statement does not contain a case.

Example:

```
foo (i,j) {  
    switch (0)                /* bad switch */  
        j=9;  
    switch(i)                 /* bad switch */  
    {  
        lab: break;  
    }  
    switch (j)                /* good switch */  
    {  
        case 1: break;  
    }  
}
```

switch statements nested more than 16 deep (compiler limitation), (E)

The **switch** statements have been nested deeper than the compiler can accept.

Example:

```
int foo() {  
    switch(0) switch(0) switch(0) switch(0) switch(0)  
    switch(0) switch(0) switch(0) switch(0) switch(0)  
    switch(0) switch(0) switch(0) switch(0) switch(0)  
    switch(0) switch(0) switch(0) switch(0) switch(0);  
}
```

symbol table is too large for intermediate file, (F)

This message indicates that the size of the symbol table exceeded available memory.

syntax error; unexpected *cause*, (E)

The compiler generates an error if correct C syntax is not followed in the file.

Example:

```
int c^D
```

Control-D indicates **End-Of-File**, which is unexpected at this point in the program.

[ANSI] tag "*tag_name*" is local to the prototype; match impossible, (W)

If you specify a structure tag name in the scope of a function prototype, the compiler will be unable to match the tag name in any call of the function. This message is issued if Microtec Compiler extensions are disabled on the command line (see the **-x** option in section *Enabling Microtec Compiler Extensions* in Chapter 3, *Using Command Line Options*, for further information about disabling Microtec Compiler extensions).

Example:

```
#pragma option -nx
void g(void) {
    void f (struct NAME { int i; } *a); /* warning */
}
```

[ANSI] tag "*tag_name*" is local to the prototype; promoted to file scope, (W)

If you specify a structure tag name in the scope of a function prototype, the tag is promoted to the file scope. This message is issued if Microtec Compiler extensions are enabled. Refer to the **-x** option in section *Enabling Microtec Compiler Extensions* in Chapter 3, *Using Command Line Options*, for further information on enabling Microtec Compiler extensions.

Example:

```
#pragma option -x
void f(struct tag *s); /* warning */
```

In this example, `tag` is specified in the function prototype of function `f`. The function `tag` is promoted to the file scope.

[ANSI] too few arguments, (E)

A function was called with too few arguments.

Example:

```
int foo(int x, int y) { int a; foo(a); }
```

[ANSI] too many arguments, (E)

A function was called with too many arguments.

Example:

```
int foo(int x) { int a,b; foo(a,b); }
```

too many files opened, (F)

The preprocessor can open at most 63 files. This message indicates that too many files were opened at the same time by the preprocessor.

too many initializers for *variable*, (E)

According to the declaration, too many initial values have been specified for a variable.

Example:

```
int foo() { int a[4] = {1,2,3,4,5,6}; }
```

***number* trailing padding byte added (size=*size*), (W)**

One byte of padding has been added to align a structure. The final byte size of the structure is indicated by *size*.

Example:

```
packed struct {  
    char c;  
    long x;  
};
```

To evenly align the structure, the compiler will add one byte after *c* to create a structure with a final size of 6 bytes.

***number* trailing padding byte[s] added (size=*size*); use *option*, (W)**

One or more (*number*) padding bytes has been added to align a structure. The final byte size of the structure is indicated by *size*. For information on compiler options

that affect structure alignment, refer to the section *Modify Alignment (not a driver option)* in Chapter 3, *Using Command Line Options*.

Example:

```
packed struct {  
    char c;  
    long x;  
};
```

To evenly align the structure, the compiler will add one byte after `c` to create a structure with a final size of 6 bytes.

type used as argument; replaced with its sizeof: *size*, (W)

An object type is incorrectly being used as an argument; it is replaced by its size.

Example:

```
int foo() { bar(int); }
```

unclosed character literal, (W)

A literal character should be enclosed by two single quotes (*'char'*). This message is emitted if one of the quotes of the literal character is missing. This message is from the preprocessor.

Example:

```
int a = ';
```

unclosed comment, (W)

The end of the source file was reached without finding a closed comment symbol (**/*). This message is from the preprocessor.

Example:

```
int some_data;  
/*  
    This is an unclosed comment stream  
<EOF>
```

unclosed string literal; treated as empty string, (W)

A string literal is missing a double quote (") to close the literal string. This message is from the preprocessor.

Example:

```
void main() {  
    char *str = "abc;  
}
```

undeclared identifier *identifier*, (E)

An identifier was used without a prior declaration.

Example:

```
int foo() { x=5;}
```

undefined enum *identifier*, (W)

If an identifier has been declared to be a type of an undefined enumerator and it is subsequently used in an expression, the compiler generates this warning.

Example:

```
enum xx yy; int foo() { yy = 0; }
```

undefined label *label*, (E)

A label was used but not defined within a function.

Example:

```
int foo() { goto x; }
```

undefined statement label "*label_name*," (E)

This message is issued if an undefined label was supplied in the **goto** statement.

Example:

```
void f(void)  
{  
    goto L1;  
}
```

unexpected symbol; symbol inserted, (W)

An unexpected symbol was encountered, and the expected symbol was inserted to correct the situation. For example, if an **End-Of-Line** symbol is encountered before the ")" symbol while processing the "(" expression, a ")" symbol will be inserted. This message is from the preprocessor.

Example:

```
#if (1
```

[ANSI] unexpected symbol *symbol*; rest of line ignored, (I)

An unexpected symbol was encountered in processing the **#if/#ifdef/#include** and similar other directives. The unexpected symbol causes the rest of the line containing the directive to be ignored. This message is from the preprocessor.

Example:

```
#if 1)
```

unintuitive sizeof value: *size*, (I)

If you cast the type of a **sizeof** operand to a different type, this message is issued. *size* indicates the size of the operand.

Example:

```
char c;  
sizeof ((int) c);
```

unknown directive *directive*; ignored, (W)

The specified directive is not recognized. This message is from the preprocessor.

Example:

```
#stop
```

unknown option *option*, (W)

The compiler does not recognize the option entered on the command line.

unknown value "*unknown_suboption*" given for *option*, (W)

The compiler does not recognize the suboption entered on the command line.

Example:

```
-Fz
```

unpacked union "*union*" inside a packed struct (W)

This warning is generated when there is an unpacked union or struct nested inside a packed structure.

Example:

```
packed struct a {  
    int i;  
    struct b {          /* Unpacked struct in a packed */  
        int i;          /* struct */  
        int j;  
    }  
}
```

The compiler generates the nested structure as an unpacked structure.

Unresolved forward reference to struct/union type index *nnn*, (W)

When the debug option (**-g**) is specified and the program has complicated forward referenced **struct** types, this warning is issued. The situation is nonfatal to the linker and the XRAY Debugger.

use of "long double" conflicts with "no ansi" option; "double" assumed, (W)

The **long** and **double** specifiers conflict if the file has been compiled with the ANSI features disabled. The **long double** specifier is supported under the ANSI option (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA  
long double i;
```

use of prototypes conflicts with "no ansi" option, (W)

Function prototypes are ANSI extensions. Your file has been compiled with the ANSI features disabled, but the compiler found a function with a prototype specification. Function prototypes are supported under the ANSI option (see the **-A** option in Chapter 3, *Using Command Line Options*, for more information about enabling and disabling ANSI features).

Example:

```
#pragma option -nA
int foo(void);
```

value returned from a void function, (E)

A value was illegally returned from a **void** function.

Example:

```
void foo() { int x; x=5; return(x); }
```

variable *variable* has unknown size, (E)

A variable with an incomplete type was declared.

Example:

```
int foo() { struct tag x; }
```

variable *variable* may be used before set, (W)

This warning message indicates that a variable has been used before it has been defined. This warning is issued if the extra check option was specified (see the **-v** option in section *Perform Extra Checking (not a driver option)* in Chapter 3, *Using Command Line Options*, for further information).

Example:

```
var_used_before_set()
{
    int i,j,k;

    while(j)          /* j is not initialized in the */
                      /* first iteration of the loop */
    {
        if (i) /* i is used before set */
        {
            j = test();
        }
        else
        {
            k = test();
        }
        test2();
    }
    test(i,j,k);      /* i is used before set */
}
```

variable *variable* replaced by constant *number*, (I)

This message is produced when a local variable is used but can be replaced by a constant for some assignments.

Example:

```
const_prop()
{
    int i;

    i = 3;
    x = i + 2;          /* i == 3 */
    i = test();
    test2(i);
}
```

In this example, the variable `i` will be replaced by the constant 3.

variable *variable* set but not used; assignment eliminated, (I)

This informational message is issued if a variable has been defined but is never used. The compiler will remove this variable.

Example:

```
unused_definition_elimination()
{
    int i,j;

    i = xxx();
    j = yyy(); /* The value of the assignment of j is */
              /* not used */
    if (i)
    {
        j = zzz();
        ddd(j);
    }
}
```

[ANSI] void pointer treated as char pointer, (W)

A pointer to **void** has been illegally used as an operand.

Example:

```
int foo() { void *p; p++; }
```

void type

This message is issued if you supply a bad identifier and declare the type of the identifier as **void**.

Example:

```
void ();
```

void type returned, (E)

A value of **void** type was returned from a function.

Example:

```
int foo() { void bar(); return (bar()); }
```

C++ Compiler Messages

Table A-3 lists all possible C++ compiler messages. Following the table are expanded descriptions and examples for messages that are not self explanatory.

Table A-3. C++ Compiler Error Messages

Number	Message
B0000	unknown error
B0001 ^a	variable "xxxx" set but not used; assignment deleted
B0002 ^a	variable "xxxx" replaced by constant "nnnn"
B0003 ^a	recursive call on function "entity" replaced by jump
B0004	unsupported feature "entity"
B0005	unrecognized option "xxxx"
B0006	expected CIL version x.y, found m.n
B0007	function "entity" not inlined
B0008	pullback option -ON "nnnn" is reserved
B0009	unknown pullback option -ON "nnnn"
B0010	reserved register %s is required (unused in 68K)
B0011 ^a	variable "entity" may be used before set
B0012 ^a	unreachable statement
B0013	return value missing
B0014	branch into a loop
B0015	branch into a switch statement
B0016	unknown processor type "entity"
B0017 ^a	IF expression evaluates to constant
B0018 ^a	SWITCH expression evaluates to constant "number"
B0019	local variable "entity" never used (unused in 68K)
B0020	local static variable "entity" set before use (unused in 68K)
B0021 ^a	statement has no effect
B0022	register history optimization (-Oh) suppressed (unused in 68K)

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
B0023	code table coverage array size exceeded: %d (unused in 68K)
B0024 ^a	multiple " <i>entity_kind</i> " " <i>entity_1</i> " and " <i>entity_2</i> "
B0025	incorrect argument type for inline function: " <i>entity</i> " (unused in 68K)
B0026	inline function not declared as double: %s (unused in 68K)
B0027 ^a	PC-relative branch range exceeds 32k
B0028	Struct assignment exceeds 131071 bytes (unused in 68K)
B0029	Duplicate case labels after converting to switch type (unused in 68K)
B0030	Pathname too long for -Gf option
B0031	invalid register " <i>entity</i> " specified with -KR
B0032	incorrect use of " <i>entity</i> " with -KR
B0033	not enough A-registers available for code generation
B0034 ^a	" <i>entity_kind</i> " allocation for " <i>entity</i> " exceeds " <i>nnnn</i> "
B0035	out of memory
B0036	can't open file " <i>filename</i> " for " <i>operation</i> "
B0037	can't seek on file " <i>filename</i> "
B0038	file " <i>filename</i> " is empty
B0039	CIL: illegal " <i>record kind</i> " " <i>value</i> " at offset " <i>location</i> "
B0040	internal error
B0041	too many types (more than 1000)
B0042	switch stmt nested too deep (unused in 68K)
B0043	case outside a switch statement (unused in 68K)
B0044	too many case labels (unused in 68K)
B0045	IEEE object output not supported yet (unused in 68K)
B0046 ^a	FE/BE out of sync; use -Rg to press on
B0047	too many errors
B0048	_regparm parameter " <i>parameter</i> " is passed on the stack

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
B0049	type mismatch: " <i>node_1</i> " child of " <i>node_2</i> " cannot be " <i>node_3</i> " (internal error)
C0000	unknown error
C0001	last line of file ends without a newline
C0002	last line of file ends with a backslash
C0003	#include file " <i>filename</i> " includes itself
C0004 ^a	out of memory
C0005	could not open source file " <i>filename</i> "
C0006	comment unclosed at end of file
C0007	unrecognized token
C0008	missing closing quote
C0009	nested comment is not allowed
C0010	"#" not expected here
C0011	unrecognized preprocessing directive
C0012	parsing restarts here after previous syntax error
C0013	expected a filename
C0014	extra text after expected end of preprocessing directive
C0015	" <i>filename</i> " is not a file containing source text
C0016	" <i>filename</i> " is not a valid source filename
C0017	expected a "]"
C0018	expected a ")"
C0019 ^a	extra text after expected end of number
C0020	identifier " <i>xxxx</i> " is undefined
C0021	type qualifiers are meaningless in this declaration
C0022	invalid hexadecimal number
C0023	integer constant is too large

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0024	invalid octal digit
C0025	quoted string should contain at least one character
C0026	too many characters in character constant
C0027	character value is out of range
C0028	expression must have a constant value
C0029	expected an expression
C0030	floating constant is out of range
C0031	expression must have integral type
C0032	expression must have arithmetic type
C0033	expected a line number
C0034	invalid line number
C0035	#error directive: <i>xxxx</i>
C0036	the #if for this directive is missing
C0037	the #endif for this directive is missing
C0038	directive is not allowed -- an #else has already appeared
C0039	division by zero
C0040	expected an identifier
C0041	expression must have arithmetic or pointer type
C0042	operand types are incompatible (" <i>type</i> " and " <i>type</i> ")
C0043	expression must have integral or pointer type
C0044	expression must have pointer type
C0045	#undef may not be used on this predefined name
C0046	this predefined name may not be redefined
C0047	macro redefined differently
C0048	cast between pointer-to-object and pointer-to-function
C0049	duplicate macro parameter name

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0050	"##" may not be first in a macro definition
C0051	"##" may not be last in a macro definition
C0052	expected a macro parameter name
C0053	expected a ":"
C0054	too few arguments in macro invocation
C0055	too many arguments in macro invocation
C0056	operand of sizeof may not be a function
C0057	this operator is not allowed in a constant expression
C0058	this operator is not allowed in a preprocessing expression
C0059	function call is not allowed in a constant expression
C0060	this operator is not allowed in an integral constant expression
C0061	integer operation result is out of range
C0062	shift count is negative
C0063	shift count is too large
C0064	declaration does not declare anything
C0065	expected a ";"
C0066	enumeration value is out of "int" range
C0067	expected a "}"
C0068	integer conversion resulted in a change of sign
C0069	integer conversion resulted in truncation
C0070	incomplete type is not allowed
C0071	operand of sizeof may not be a bit field
C0072	operand of "&" may not be a constant
C0073	operand of "&" in an initializer must be static
C0074	invalid operand of "&"
C0075	operand of "*" must be a pointer

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0076	argument to macro is empty
C0077	this declaration has no storage class or type specifier
C0078	a parameter declaration may not have an initializer
C0079 ^a	expected a type specifier
C0080	a storage class may not be specified here
C0081	more than one storage class may not be specified
C0082 ^a	storage class is not first
C0083 ^a	type qualifier specified more than once
C0084 ^a	invalid combination of type specifiers
C0085 ^a	invalid storage class for a parameter
C0086 ^a	invalid storage class for a function
C0087	a type specifier may not be used here
C0088	array of functions is not allowed
C0089	array of void is not allowed
C0090	function returning function is not allowed
C0091	function returning array is not allowed
C0092	identifier-list parameters may only be used in a function definition
C0093	function type may not come from a typedef
C0094	the size of an array must be greater than zero
C0095 ^a	array is too large
C0096	a translation unit must contain at least one declaration
C0097 ^a	a function may not return a value of this type
C0098 ^a	an array may not have elements of this type
C0099	a declaration here must declare a parameter
C0100	duplicate parameter name
C0101	"xxxx" has already been declared in the current scope

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0102	forward-defined enum type is nonstandard
C0103 ^a	class is too large
C0104 ^a	struct or union is too large
C0105 ^a	invalid size for bit field
C0106 ^a	invalid type for a bit field
C0107	zero-length bit field must be unnamed
C0108	signed bit field of length 1
C0109	expression must have (pointer-to-) function type
C0110	expected either a definition or a tag name
C0111	statement is unreachable
C0112	expected "while"
C0113	this use of a default argument is nonstandard
C0114	<i>entity-kind</i> " <i>entity</i> " was referenced but not defined
C0115	a continue statement may only be used within a loop
C0116	a break statement may only be used within a loop or switch
C0117	non-void <i>entity-kind</i> " <i>entity</i> " (declared at line <i>xxxx</i>) should return a value
C0118	a void function may not return a value
C0119	cast to type " <i>type</i> " is not allowed
C0120 ^a	return value type does not match the function type
C0121	a case label may only be used within a switch
C0122	a default label may only be used within a switch
C0123	case label value has already appeared in this switch
C0124	default label has already appeared in this switch
C0125	expected a "("
C0126 ^a	expression must be an lvalue

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0127	expected a statement
C0128	loop is not reachable from preceding code
C0129	a block-scope function may only have extern storage class
C0130	expected a "{"
C0131	expression must have pointer-to-class type
C0132	expression must have pointer-to-struct-or-union type
C0133	expected a member name
C0134	expected a field name
C0135	<i>entity-kind "entity"</i> has no member "xxxx"
C0136	<i>entity-kind "entity"</i> has no field "xxxx"
C0137 ^a	expression must be a modifiable lvalue
C0138	taking the address of a register variable is not allowed
C0139	taking the address of a bit field is not allowed
C0140	too many arguments in function call
C0141	unnamed prototyped parameters not allowed when body is present
C0142	expression must have pointer-to-object type
C0143	program too large or complicated to compile
C0144	a value of type " <i>type</i> " cannot be used to initialize an entity of type " <i>type</i> "
C0145	<i>entity-kind "entity"</i> may not be initialized
C0146	too many initializer values
C0147	declaration is incompatible with <i>entity-kind "entity"</i> (declared at line <i>xxxx</i>)
C0148	<i>entity-kind "entity"</i> has already been initialized
C0149	a global-scope declaration may not have this storage class
C0150	a type name may not be redeclared as a parameter
C0151	a typedef name may not be redeclared as a parameter

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0152	conversion of nonzero integer to pointer
C0153 ^a	expression must have class type
C0154 ^a	expression must have struct or union type
C0155 ^a	old-fashioned assignment operator
C0156 ^a	old-fashioned initializer
C0157	expression must be an integral constant expression
C0158 ^a	expression must be an lvalue or a function designator
C0159	declaration is incompatible with previous " <i>entity</i> " (declared at line <i>xxxx</i>)
C0160	name conflicts with previously used external name " <i>xxxx</i> "
C0161	unrecognized #pragma
C0162	expression must have arithmetic, pointer, or void type
C0163	could not open temporary file " <i>filename</i> "
C0164 ^a	name of directory for temporary files is too long (" <i>xxxx</i> ")
C0165	too few arguments in function call
C0166	invalid floating constant
C0167	argument of type " <i>type</i> " is incompatible with parameter of type " <i>type</i> "
C0168	a function type is not allowed here
C0169	expected a declaration
C0170 ^a	pointer points outside of underlying object
C0171	invalid type conversion
C0172 ^a	external/internal linkage conflict with previous declaration
C0173	floating-point value does not fit in required integral type
C0174	expression has no effect
C0175	subscript out of range
C0176	constant string subscript out of range

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0177	<i>entity-kind "entity"</i> was declared but never referenced
C0178	"&" applied to an array has no effect
C0179	right operand of "%" is zero
C0180	argument is incompatible with formal parameter
C0181	argument is incompatible with corresponding format string conversion
C0182	could not open source file " <i>filename</i> " (no directories in search list)
C0183	type of cast must be integral
C0184	type of cast must be arithmetic or pointer
C0185	dynamic initialization in unreachable code
C0186	pointless comparison of unsigned integer with zero
C0187	possible use of "=" where "==" was intended
C0188	enumerated type mixed with another type
C0189	error while writing " <i>filename</i> " file
C0190	invalid intermediate language file
C0191	type qualifier is meaningless on cast type
C0192	unrecognized character escape sequence
C0193	zero used for undefined preprocessing identifier
C0194	expected an asm string
C0195	an asm function must be prototyped
C0196	an asm function may not have an ellipsis
C0197	asm may only be used to declare a function
C0198	an asm function may not have a storage class
C0199	asm return value size does not match function return type
C0200	asm parameter size does not match function parameter size
C0202	invalid combination of asm control specifiers

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0203	extra text after expected end of asm control line
C0204	expected an asm control specifier
C0205	this asm name is already defined
C0206	invalid register name
C0207	an asm parameter may not have void type
C0208	expected an asm type specification
C0209	invalid asm type specification
C0210	invalid asm type width
C0211	invalid asm constant
C0212	an asm temporary may not have this type
C0213	this parameter may not be referenced because it has no type
C0214	the return value may not be referenced because its type is void
C0215	invalid register specifier
C0217	the return value may not be referenced because it has no type
C0218	the return value may not have this asm type
C0219	error while deleting file " <i>filename</i> "
C0220	integral value does not fit in required floating-point type
C0221	floating-point value does not fit in required floating-point type
C0222	floating-point operation result is out of range
C0223	function declared implicitly
C0224 ^a	the format string requires additional arguments
C0225 ^a	the format string ends before this argument
C0226	invalid format string conversion
C0227	macro recursion
C0228	extra final comma is nonstandard
C0229	bit field cannot contain all values of the enumerated type

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0230	nonstandard type for a bit field
C0231	declaration is not visible outside of function
C0232	old-fashioned typedef of "void" ignored
C0233	left operand is not a struct or union containing this field
C0234	pointer does not point to struct or union containing this field
C0235	variable "xxxx" was declared with a never-completed type
C0236	controlling expression is constant
C0237	selector expression is constant
C0238	invalid specifier on a parameter
C0239	invalid specifier outside a class declaration
C0240	duplicate specifier in declaration
C0241	a union is not allowed to have a base class
C0242 ^a	multiple access control specifiers are not allowed
C0243	class or struct definition is missing
C0244	qualified name is not a member of class " <i>type</i> " or its base classes
C0245	a nonstatic member reference must be relative to a specific object
C0246	a nonstatic data member may not be defined outside its class
C0247	<i>entity-kind</i> " <i>entity</i> " has already been defined
C0248	pointer to reference is not allowed
C0249	reference to reference is not allowed
C0250	reference to void is not allowed
C0251	array of reference is not allowed
C0252	reference <i>entity-kind</i> " <i>entity</i> " requires an initializer
C0253	expected a ", "
C0254	type name is not allowed
C0255	type definition is not allowed

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0256	invalid redeclaration of type name "xxxx"
C0257	const <i>entity-kind</i> " <i>entity</i> " requires an initializer
C0258	"this" may only be used inside a nonstatic member function
C0259	constant value is not known
C0260	explicit type is missing ("int" assumed)
C0261 ^a	access control not specified ("xxxx" by default)
C0262	not a class or struct name
C0263	duplicate base class name
C0264	invalid base class
C0265	<i>entity-kind</i> " <i>entity</i> " is inaccessible
C0266	" <i>entity</i> " is ambiguous
C0267 ^a	old-style parameter list (anachronism)
C0268	declaration may not appear after executable statement in block
C0269	base class " <i>type</i> " is inaccessible
C0270	name is not a member of a base class of "xxxx"
C0271 ^a	access adjustment in a "private" section is not allowed
C0272	increasing an inherited member's access is not allowed
C0273	restricting an inherited member's access is not allowed
C0274	improperly terminated macro invocation
C0275 ^a	invalid access declaration -- " <i>entity</i> " is hidden by " <i>entity</i> "
C0276	name followed by "::" must be a class name
C0277	invalid friend declaration
C0278	a constructor or destructor may not return a value
C0279	invalid destructor declaration
C0280	invalid declaration of a member with the same name as its class
C0281	global-scope qualifier (leading "::") is not allowed

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0282	the global scope has no "xxxx"
C0283 ^a	qualified name is not allowed
C0284	NULL reference is not allowed
C0285	initialization with "{...}" is not allowed for object of type " <i>type</i> "
C0286	base class " <i>type</i> " is ambiguous
C0287	derived class " <i>type</i> " contains more than one instance of class " <i>type</i> "
C0288	derived class " <i>type</i> " has class " <i>type</i> " as a virtual base class
C0289	no instance of constructor " <i>entity</i> " matches the argument list
C0290	copy constructor for class " <i>type</i> " is ambiguous
C0291	no default constructor exists for class " <i>type</i> "
C0292	"xxxx" is not a nonstatic data member or base class of class " <i>type</i> "
C0293	indirect nonvirtual base class is not allowed
C0294	invalid union member -- class " <i>type</i> " has a disallowed member function
C0295	cannot overload functions -- parameter types are too similar
C0296 ^a	invalid use of non-lvalue array
C0297 ^a	expected an operator
C0298 ^a	inherited member is not allowed
C0299	cannot determine which instance of <i>entity-kind</i> " <i>entity</i> " is intended
C0300	a pointer to a bound function may only be used to call the function
C0301	typedef name has already been declared (with same type)
C0302	<i>entity-kind</i> " <i>entity</i> " has already been defined
C0303	type does not match any instance of <i>entity-kind</i> " <i>entity</i> "
C0304	no instance of <i>entity-kind</i> " <i>entity</i> " matches the argument list
C0305	type definition is not allowed in function return type declaration
C0306	default argument not at end of parameter list

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0307	redefinition of default argument
C0308 ^a	more than one instance of <i>entity-kind</i> " <i>entity</i> " matches the argument list:
C0309 ^a	more than one instance of constructor " <i>entity</i> " matches the argument list:
C0310	default argument of type " <i>type</i> " is incompatible with parameter of type " <i>type</i> "
C0311	cannot overload functions distinguished by return type alone
C0312	no suitable user-defined conversion from " <i>type</i> " to " <i>type</i> " exists
C0313	const or volatile qualifier on this function is not allowed
C0314	only nonstatic member functions may be virtual
C0315	function may not be called for const- or volatile-qualified object
C0316	program too large to compile (too many virtual functions)
C0317	type differs from base class virtual function by return type alone
C0318	override of virtual <i>entity-kind</i> " <i>entity</i> " is ambiguous
C0319	pure specifier ("= 0") allowed only on virtual functions
C0320	badly-formed pure specifier (only "= 0" is allowed)
C0321	data member initializer is not allowed
C0322	object of abstract class type is not allowed
C0323	function returning abstract class is not allowed
C0324	duplicate friend declaration
C0325	inline specifier allowed on function declarations only
C0326	"inline" is not allowed
C0327	invalid storage class for an inline function
C0328	invalid storage class for a class member
C0329	member function of local class -- definition is required
C0330	<i>entity-kind</i> " <i>entity</i> " is inaccessible

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0332	class " <i>type</i> " has no copy constructor to copy a const object
C0333	defining an implicitly declared member function is not allowed
C0334	class " <i>type</i> " has no suitable copy constructor
C0335 ^a	linkage specification is not allowed
C0336 ^a	unknown external linkage specification
C0337 ^a	linkage specification is incompatible with previous " <i>entity</i> "
C0338 ^a	more than one instance of <i>entity-kind</i> " <i>entity</i> " has "C" linkage
C0339	class " <i>type</i> " has more than one default constructor
C0340	value copied to temporary, reference to temporary used
C0341	"operatorxxxx" must be a member function
C0342	operator may not be a static member function
C0343	no arguments allowed on user-defined conversion
C0344	too many arguments for operator function
C0345	too few arguments for operator function
C0346	nonmember operator requires an argument with class type
C0347	default argument is not allowed
C0348 ^a	more than one user-defined conversion from " <i>type</i> " to " <i>type</i> " applies:
C0349	none of the available operator functions matches these operands
C0350 ^a	more than one operator "xxxx" matches these operands:
C0351	operator new() requires first argument of type "size_t"
C0352	operator new() requires return type of "void *"
C0353	operator delete() requires return type of "void"
C0354	operator delete() requires first argument of type "void *"
C0355	second argument of operator delete() must be of type "size_t"
C0356	type must be an object type

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0357	base class " <i>type</i> " has already been initialized
C0358	base class name required -- " <i>type</i> " assumed (anachronism)
C0359	<i>entity-kind</i> " <i>entity</i> " has already been initialized
C0360	name of member or base class is missing
C0361	assignment to "this" (anachronism)
C0362	"overload" keyword used (anachronism)
C0363	invalid anonymous union -- nonpublic member is not allowed
C0364	invalid anonymous union -- member function is not allowed
C0365	global anonymous union must be declared static
C0366 ^a	<i>entity-kind</i> " <i>entity</i> " provides no initializer for:
C0368 ^a	<i>entity-kind</i> " <i>entity</i> " defines no constructor to initialize the following:
C0369	<i>entity-kind</i> " <i>entity</i> " has an uninitialized const or reference member
C0370	<i>entity-kind</i> " <i>entity</i> " has an uninitialized const field
C0371	class " <i>type</i> " has no assignment operator to copy a const object
C0372	class " <i>type</i> " has no suitable assignment operator
C0373	ambiguous default assignment operator for class " <i>type</i> "
C0374	const or volatile qualifier is not allowed
C0375	declaration requires a typedef name
C0377	"virtual" is not allowed
C0378	"static" is not allowed
C0379 ^a	cast of bound function to normal function pointer (anachronism)
C0380	expression must have pointer-to-member type
C0381	extra ";" ignored
C0382	declaring a member constant is nonstandard
C0383	a pointer to const may not be deleted

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0384	no instance of overloaded " <i>entity</i> " matches these operands
C0385	operator delete() may not be overloaded
C0386	no instance of <i>entity-kind</i> " <i>entity</i> " matches the required type
C0387	delete array size expression ignored (anachronism)
C0388	operator->() requires pointer-to-class return type
C0389	a cast to an abstract class is not allowed
C0390	function "main" may not be called or have its address taken
C0391	a new-initializer may not be specified for an array
C0392	a member function may not be redeclared outside its class
C0393	pointer to incomplete class type is not allowed
C0394	reference to local variable of enclosing function is not allowed
C0395 ^a	single-argument function used for postfix "xxx" (anachronism)
C0396 ^a	access adjustment is not allowed -- mixed accessibility for <i>entity-kind</i> " <i>entity</i> "
C0397 ^a	implicitly generated assignment operator cannot copy:
C0398	cast to array type is nonstandard (treated as cast to " <i>type</i> ")
C0399	<i>entity-kind</i> " <i>entity</i> " has an operator new() but no operator delete()
C0400	<i>entity-kind</i> " <i>entity</i> " has an operator delete() but no operator new()
C0401	destructor for base class " <i>type</i> " is not virtual
C0402	<i>entity-kind</i> " <i>entity</i> " has no accessible constructors
C0403	<i>entity-kind</i> " <i>entity</i> " has already been declared
C0404	function "main" may not be declared inline
C0405	member function with the same name as its class must be a constructor
C0406	using nested <i>entity-kind</i> " <i>entity</i> " (anachronism)
C0407	a destructor may not have parameters

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0408	copy constructor for class " <i>type</i> " may not have a parameter of type " <i>type</i> "
C0409	function return type is incomplete
C0410	protected <i>entity-kind</i> " <i>entity</i> " is not accessible through a " <i>type</i> " pointer or object
C0411	a parameter is not allowed
C0412	an "asm" declaration is not allowed at this point
C0413	no suitable conversion function from " <i>type</i> " to " <i>type</i> " exists
C0414	delete of pointer to incomplete class
C0415	no suitable constructor exists to convert from " <i>type</i> " to " <i>type</i> "
C0416 ^a	more than one constructor applies to convert from " <i>type</i> " to " <i>type</i> ":
C0417 ^a	more than one conversion function from " <i>type</i> " to " <i>type</i> " applies:
C0418 ^a	more than one conversion function from " <i>type</i> " to a built-in type applies:
C0419	const <i>entity-kind</i> " <i>entity</i> "
C0420 ^a	reference <i>entity-kind</i> " <i>entity</i> "
C0421 ^a	<i>entity-kind</i> " <i>entity</i> "
C0422	built-in operator "xxx"
C0423	<i>entity-kind</i> " <i>entity</i> " (ambiguous by inheritance)
C0424	a constructor or destructor may not have its address taken
C0425	dollar sign ("\$\$") used in identifier
C0426	temporary used for initial value of reference to non-const (anachronism)
C0427	qualified name is not allowed in member declaration
C0428	enumerated type mixed with another type (anachronism)
C0429	the size of an array in "new" must be non-negative
C0430	returning reference to local temporary

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0431	const qualifier dropped in initializing reference to non-const
C0432	"enum" declaration is not allowed
C0433	initial value of reference has excess const/volatile qualifiers
C0434	initial value of reference to non-const has incorrect type
C0435	a pointer to function may not be deleted
C0436	conversion function must be a nonstatic member function
C0437	nonglobal template declaration is not allowed
C0438	expected a "<"
C0439	expected a ">"
C0440	template parameter declaration is missing
C0441	argument list for <i>entity-kind</i> " <i>entity</i> " is missing
C0442	too few arguments for <i>entity-kind</i> " <i>entity</i> "
C0443	too many arguments for <i>entity-kind</i> " <i>entity</i> "
C0444	template parameter for a function template must be a type
C0445	<i>entity-kind</i> " <i>entity</i> " is not used in declaring the argument types of <i>entity-kind</i> " <i>entity</i> "
C0446	two nested types have the same name: " <i>entity</i> " and " <i>entity</i> " (declared at line xxxx) (Cfront compatibility)
C0447	global " <i>entity</i> " was declared after nested " <i>entity</i> " (declared at line xxxx) (Cfront compatibility)
C0448	template parameter " <i>entity</i> " was declared but never referenced
C0449	more than one instance of <i>entity-kind</i> " <i>entity</i> " matches the required type
C0450	the type "long long" is nonstandard
C0451	omission of "xxxx" is nonstandard
C0452	return type may not be specified on a conversion function
C0456 ^a	excessive recursion at instantiation of <i>entity-kind</i> " <i>entity</i> "

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0457	"xxxx" is not a function or static data member
C0458	argument of type " <i>type</i> " is incompatible with template parameter of type " <i>type</i> "
C0459	initialization requiring a temporary or conversion is not allowed
C0460	declaration of "xxxx" hides function parameter
C0461	initial value of reference to non-const must be an lvalue
C0463	"template" is not allowed
C0464	" <i>type</i> " is not a class template
C0465	static data member may not be an anonymous union
C0466	"main" is not a valid name for a function template
C0467	invalid reference to <i>entity-kind</i> " <i>entity</i> " (union/nonunion mismatch)
C0468	a template argument may not reference a local type
C0469 ^a	tag kind of xxxx is incompatible with declaration of <i>entity-kind</i> " <i>entity</i> " (declared at line xxxx)
C0470	the global scope has no tag named "xxxx"
C0471	<i>entity-kind</i> " <i>entity</i> " has no tag member named "xxxx"
C0472	member function typedef (allowed for Cfront compatibility)
C0473	<i>entity-kind</i> " <i>entity</i> " may be used only in pointer-to-member declaration
C0475	a template argument may not reference a non-external entity
C0476	name followed by "::" must be a class name or a type name
C0477	destructor name does not match name of class " <i>type</i> "
C0478	type used as destructor name does not match type " <i>type</i> "
C0479	<i>entity-kind</i> " <i>entity</i> " may not be redeclared "inline" after being called
C0480	destructor name does not match left operand of "->" or "."
C0481	invalid storage class for a template declaration

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0482	<i>entity-kind "entity"</i> is an inaccessible type (allowed for Cfront compatibility)
C0483	a return type is not allowed
C0484	invalid instantiation pragma argument
C0485	<i>entity-kind "entity"</i> is not an entity that can be instantiated
C0486	compiler generated function <i>entity-kind "entity"</i> cannot be instantiated
C0487	inline function <i>entity-kind "entity"</i> cannot be instantiated
C0488	pure virtual function <i>entity-kind "entity"</i> cannot be instantiated
C0489	<i>entity-kind "entity"</i> cannot be instantiated -- no template definition was supplied
C0490	<i>entity-kind "entity"</i> cannot be instantiated -- a specific definition has been supplied
C0491	class " <i>type</i> " has no constructor
C0492	<i>entity-kind "entity"</i> must be used in a parameter without a default value in <i>entity-kind "entity"</i>
C0493	no instance of <i>entity-kind "entity"</i> matches the specified type
C0494	declaring a void parameter list with a typedef is nonstandard
C0495	global <i>entity-kind "entity"</i> used instead of <i>entity-kind "entity"</i> (Cfront compatibility)
C0496	template parameter " <i>xxx</i> " may not be redeclared in this scope
C0497	declaration of " <i>xxx</i> " hides template parameter
C0498	template argument list must match the parameter list
C0499	conversion function to convert from " <i>type</i> " to " <i>type</i> " is not allowed
C0500 ^a	extra argument of postfix " <i>operatorxxx</i> " must be of type "int"
C0501	an operator name must be declared as a function
C0502	operator name is not allowed
C0503	class template specific definition not at global scope

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0504	nonstandard form for taking the address of a member function
C0505	too few template parameters -- does not match previous declaration
C0506	too many template parameters -- does not match previous declaration
C0507	function template for operator delete() is not allowed
C0508	class template and template parameter may not have the same name
C0509	" <i>entity</i> " cannot be used to designate constructor for <i>entity-kind</i> " <i>entity</i> "
C0510	a template argument may not reference an unnamed type
C0511 ^a	enumerated type is not allowed
C0512	type qualifier on a reference type is not allowed
C0513	a value of type " <i>type</i> " cannot be assigned to an entity of type " <i>type</i> "
C0514	pointless comparison of unsigned integer with a negative constant
C0515	cannot convert to incomplete class " <i>type</i> "
C0516	const object requires an initializer
C0517	object has an uninitialized const or reference member
C0518	nonstandard preprocessing directive
C0519	<i>entity-kind</i> " <i>entity</i> " may not have a template argument list
C0520 ^a	initialization with "{...}" expected for aggregate object
C0521 ^a	pointer-to-member selection class types are incompatible (" <i>type</i> " and " <i>type</i> ")
C0522 ^a	pointless friend declaration
C0523	"." used in place of "::" to form a qualified name (Cfront anachronism)
C0524	non-const function called for const object (Cfront anachronism)

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0525 ^a	a dependent statement may not be a declaration
C0526	a parameter may not have void type
C0529	this operator is not allowed in a template argument expression
C0530	try block requires at least one handler
C0531	handler requires an exception declaration
C0532	handler is masked by default handler
C0533	handler is masked by previous handler for type " <i>type</i> "
C0534	use of a local type to specify an exception
C0535	redundant type in exception specification
C0536	exception specification is incompatible with that of previous <i>entity-kind</i> " <i>entity</i> " (declared at line <i>xxxx</i>):
C0540	support for exception handling is disabled
C0541	omission of exception specification is incompatible with previous <i>entity-kind</i> " <i>entity</i> " (declared at line <i>xxxx</i>):
C0542	could not create instantiation information file " <i>filename</i> "
C0543	non-arithmetic operation not allowed in nontype template argument
C0544	use of a local type to declare a nonlocal variable
C0545	use of a local type to declare a function
C0546 ^a	transfer of control bypasses initialization of:
C0547	<i>entity-kind</i> " <i>entity</i> " (declared at line <i>xxxx</i>)
C0548	transfer of control into an exception handler
C0549	<i>entity-kind</i> " <i>entity</i> " is used before its value is set
C0550	<i>entity-kind</i> " <i>entity</i> " was set but never used
C0551	<i>entity-kind</i> " <i>entity</i> " cannot be defined in the current scope
C0552	exception specification is not allowed

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0553 ^a	external/internal linkage conflict for <i>entity-kind</i> " <i>entity</i> " (declared at line <i>xxxx</i>)
C0554 ^a	<i>entity-kind</i> " <i>entity</i> " will not be called for implicit or explicit conversions
C0555	tag kind of <i>xxxx</i> is incompatible with template parameter of type " <i>type</i> "
C0556	function template for operator new(size_t) is not allowed
C0557	invalid access declaration -- inherited name " <i>xxxx</i> " is ambiguous
C0558	pointer to member of type " <i>type</i> " is not allowed
C0559	ellipsis is not allowed in operator function parameter list
C0560	" <i>entity</i> " is reserved for future use as a keyword
C0561 ^a	invalid macro definition:
C0562 ^a	invalid macro undefinition:
C0563	invalid preprocessor output file
C0564	cannot open preprocessor output file
C0569	cannot open C output file
C0570	error in debug option argument
C0571	invalid option:
C0574	invalid number:
C0576	invalid instantiation mode:
C0577	missing include file directory name
C0578	invalid error limit:
C0579	invalid raw-listing output file
C0580	cannot open raw-listing output file
C0582	cannot open cross-reference output file
C0583	invalid error output file
C0584	cannot open error output file

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0585	virtual function tables can only be suppressed when compiling C++
C0586	anachronism option (<i>option</i>) can be used only when compiling C++
C0587	instantiation mode (<i>option</i>) can be used only when compiling C++
C0590	exception handling (<i>option</i>) can be used only when compiling C++
C0591	strict ANSI mode is incompatible with K&R mode
C0592	strict ANSI mode is incompatible with Cfront mode
C0593	missing source filename
C0594	output files may not be specified when compiling several input files
C0595	too many arguments on command line
C0596	an output file was specified, but none is needed
C0598	a template parameter may not have void type
C0599	excessive recursive instantiation of <i>entity-kind</i> " <i>entity</i> " due to instantiate-all mode
C0600	strict ANSI mode is incompatible with allowing anachronisms
C0601	a throw expression may not have void type
C0602	local instantiation mode is incompatible with automatic instantiation
C0603	parameter of abstract class type is not allowed
C0604	array of abstract class is not allowed
C0605	floating-point template parameter is nonstandard
C0606	this pragma must immediately precede a declaration
C0607	this pragma must immediately precede a statement
C0608	this pragma must immediately precede a declaration or statement
C0609	this kind of pragma may not be used here

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0610	<i>entity-kind "entity"</i> does not match <i>"entity"</i> -- virtual function override intended?
C0611	overloaded virtual function <i>"entity"</i> is only partially overridden in <i>entity-kind "entity"</i>
C0612	specific definition of inline template function must precede its first use
C0613	invalid error tag:
C0614	invalid error number:
C0615	parameter type involves pointer to array of unknown bound
C0616	parameter type involves reference to array of unknown bound
C0617	interrupt specifier allowed on function declarations only
C0618	"interrupt" is not allowed
C0619	<i>entity-kind "entity"</i> may not be redeclared "interrupt" after being called
C0620	an interrupt function may not have parameters
C0621	an interrupt function may not have a non-void return type
C0622	a non static class member function may not be declared interrupt
C0623	a template function may not be declared interrupt
C0624	non interrupt call of an interrupt function
C0625	#pragma xxxx
C0626	#warning directive: xxxx
C0627	#informing directive: xxxx
C0628	no matching insert character "xxxx" found in asm string
C0629	pointer-to-member-function cast to pointer to function
C0630	expected "class", "enum", "struct" or "union"
C0631	invalid reference to <i>entity-kind "entity"</i> (packed/unpacked mismatch)
C0632	<i>entity-kind "entity"</i> assumed to be "xxxx" from a previous declaration

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0633	could not open source list file " <i>filename</i> "
C0634	could not open raw list file " <i>filename</i> "
C0635	struct or union must declare at least one named field
C0636	expected a string literal
C0637	unknown option <i>option_name</i> found during preprocessing
C0638	unknown processor <i>processor_name</i> ignored
C0639	cannot reserve more than <i>resource_name_limit</i>
C0640	cannot reserve register <i>register_name</i>
C0641	deprecated option <i>option_name</i>
C0642	option -A is meaningless for C++ programs in this release
C0643	option <i>option_name</i> is not implemented in this release
C0644	options conflict <i>conflicting_options</i>
C0645	invalid suboption <i>option_name</i>
C0646	exiting upon receipt of signal <i>signal_name</i>
C0647	option -d : cannot open options file
C0648	invalid option <i>option_name</i> in pragma
C0649	current working directory name too long or otherwise inaccessible
C0650	inter-option conflict
C0651	inter-option implication
C0652	nonstandard unnamed field
C0653	nonstandard unnamed member
C0654	a function type cannot be used as a template argument
C0655	invalid precompiled header output file
C0656	cannot open precompiled header output file
C0657	<i>name</i> is not a type name
C0658	cannot open precompiled header input file

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0659	precompiled header file <i>filename</i> is either invalid or not generated by this version of the compiler
C0660	precompiled header file <i>filename</i> was not generated in this directory
C0661	header files used to generate precompiled header file <i>filename</i> have changed
C0662	the command line options do not match those used when precompiled header file <i>filename</i> was created
C0663	the initial sequence of preprocessing directives is not compatible with those of precompiled header file <i>filename</i>
C0664	unable to obtain mapped memory
C0665	using precompiled header file <i>filename</i>
C0666	creating precompiled header file <i>filename</i>
C0667	memory usage conflict with precompiled header file <i>filename</i>
C0668	invalid PCH memory size
C0669	PCH options must appear first in the command line
C0670	insufficient memory for PCH memory allocation
C0671	precompiled header files may not be used when compiling several input files
C0672	insufficient preallocated memory for generation of precompiled header file (<i>number</i> bytes required)
C0673	very large entity in program prevents generation of precompiled header file
C0674	<i>name</i> is not a valid directory
C0675	cannot build temporary file name
C0676	"restrict" is not allowed
C0677	a pointer of reference to function type may not be qualified by "restrict"
C0678	name is an invalid __declspec attribute
C0679	a calling convention modifier may not be specified here

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0680	conflicting calling convention modifiers
C0681	strict ANSI mode is incompatible with Microsoft mode
C0682	cfront mode is incompatible with Microsoft mode
C0683	calling convention specified here is ignored
C0684	a calling convention may not be followed by a nested declarator
C0685	calling convention is ignored for this type
C0686	calling conventions may only be applied to function types
C0687	declaration modifiers are incompatible with previous declaration
C0688	the modifier <i>name</i> is not allowed in this declaration
C0689	transfer of control into a try block
C0690	inline specification is incompatible with previous declaration
C0691	closing brace of <i>statement</i> not found
C0692	command line option <i>option_name</i> disables PCH
C0693	precompiled header file <i>filename</i> was not generated for the same source file directory
C0694	invalid macro definition in #pragma option
C0695	invalid macro undefinition in #pragma option
C0696	no default for -p and no processor variants
C0697	option <i>option_name</i> requires run-time initialization of pointers
C0698	wchar_t keyword option can be used only when compiling C++
C0700 ^b	expected an integer constant
C0701	call of pure virtual function
C0702	invalid source file identifier string
C0703	a template cannot be defined in a friend declaration
C0704	"asm" is not allowed
C0707 ^b	ellipsis with no explicit parameters is nonstandard

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0708	"&..." is nonstandard
C0709	invalid use of "&..."
C0710	alternative token option can be used only when compiling C++
C0711	temporary used for initial value of reference to const volatile (anachronism)
C0712	initial value of reference to const volatile has incorrect type
C0713	initial value of reference to const volatile must be an lvalue
C0714	SVR4 C compatibility option can be used only when compiling ANSI C
C0715	using out-of-scope declaration of <i>name</i>
C0716	strict ANSI mode is incompatible with SVR4 C mode
C0717	call of <i>function</i> cannot be inlined
C0718	function cannot be inlined
C0719	invalid PCH directory
C0722 ^b	detected during instantiation of <i>template_name</i>
C0723	detected during implicit generation of <i>template_name</i>
C0724	detected during instantiation of <i>template_name</i>
C0725	detected during processing of template argument list for <i>template_name</i>
C0726	detected during implicit definition of <i>template_name</i>
C0729 ^b	RTTI option can be used only when compiling C++
C0730	<i>statement</i> , required for copy that was eliminated, is inaccessible
C0731	<i>statement</i> , required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue
C0732	<i>typeinfo</i> should be included before <i>typeid</i> is used
C0733	<i>statement</i> cannot cast away const or other type qualifiers
C0734	the type in a <i>dynamic_cast</i> must be a pointer or reference to a complete cast type, or void

(cont.)

Table A-3. C++ Compiler Error Messages (cont.)

Number	Message
C0735	the operand of a pointer <code>dynamic_cast</code> must be a pointer to a complete cast type
C0736	the operand of a reference <code>dynamic_cast</code> must be an lvalue of a complete cast type
C0737	the operand of a runtime <code>dynamic_cast</code> must have a polymorphic class type
C0738	<code>bool</code> option can be used only when compiling C++
C0739	invalid storage class for condition declaration
C0740	an array type is not allowed here
C0741	expected an "="
C0742	expected a declarator in condition declaration
C0743	name, declared in condition, may not be redeclared in this scope
C0744	default template arguments are not allowed for function templates
C0745	expected a ",", or ">"
C0746	expected a template parameter list
C0747	incrementing a <code>bool</code> value is deprecated
C0748	<code>bool</code> type is not allowed
C0749	offset of base class <i>name</i> within class <i>name</i> is too large
C0750	expression must have <code>bool</code> type (or be convertible to <code>bool</code>)
C0751	array <code>new</code> and <code>delete</code> option can be used only when compiling C++
C0752	<i>name</i> is not a variable name
C0756	the type in a <code>const_cast</code> must be a pointer, reference, or pointer to member to an object type
C0757	a <code>const_cast</code> can only adjust type qualifiers; it cannot change the underlying type

a. An extended description of this option appears after this table.

b. The skipped error numbers are not currently used.

Expanded Descriptions

This section provides further descriptions and examples of the error messages denoted in Table A-3.

B0001 variable "xxxx" set but not used; assignment deleted

The optimizer has detected a useless assignment and the **-id** and **-v** options are enabled.

Example:

```
main()
{
    int i;
    i = 1; /* i is set but not used */
}
```

B0002 variable "xxxx" replaced by constant "nnnn"

The optimizer has propagated a constant assigned to a variable and the **-ic** and **-v** options are enabled.

Example:

```
int j;
main()
{
    j = 1;
    foo(j); /* j here may be replaced by constant 1 */
}
```

B0003 recursive call on function "entity" replaced by jump

The optimizer has replaced a recursive function call by a jump to the beginning of the function, also known as tail recursion elimination. The options **-it** and **-v** must be enabled.

Example:

```
void foo(int i)
{
    if (i == 0)
        return;
    else {
        bar();
        foo(i - 1); /* This function call may be replaced by a
                     jump to the beginning of the function */
    }
}
```

B0011 variable "*entity*" may be used before set

The optimizer has detected a variable that was used before its value was set and the -v option is enabled.

Example:

```
void foo()
{
    int j;
    i = j; /* j is used without setting its value */
}
```

B0012 unreachable statement

The optimizer has detected a statement that will never be executed and the -v option is enabled.

Example:

```
int foo()
{
    if (i)
        return 1;
    else
        return 2;
    bar(); /* this statement is unreachable */
    return 3;
}
```

B0017 IF expression evaluates to constant

The optimizer has detected an “if” expression whose value is a constant and the -v option is enabled.

Example:

```
int i;
void foo()
{
    i = 1;
    if (i)      /* i is always 1 here */
        bar();
}
```

B0018 SWITCH expression evaluates to constant "*number*"

The optimizer has detected a “switch” expression whose value is a constant and the -v option is enabled.

Example:

```
int i;
void foo()
{
    i = 1;
    switch (i) {      /* i is always 1 here */
    case 1:
        f(); break;
    case 2:
        g(); break;
    }
}
```

B0021 statement has no effect

The optimizer has detected a useless statement and -v option is enabled.

Example:

```
int i;
void foo()
{
    i;      /* this statement has no effect */
}
```

B0024 multiple "*entity_kind*" "*entity_1*" and "*entity_2*"

The compiler has detected an internal inconsistency. There are too many entities of the same kind in the intermediate language.

B0027 PC-relative branch range exceeds 32k

The code in a function required generation of a PC-relative branch to a distance more than 32K. Modify the function to shorten this distance.

B0034 "entity_kind" allocation for "entity" exceeds "nnnn"

The memory allocation requirement has exceeded the maximum limit. Try reducing symbols of category “entry_kind”.

B0046 FE/BE out of sync; use -Rg to press on

Compiler has detected an internal inconsistency.

C0004 out of memory

The compiler failed to get enough memory from the system. Refer to the *Release Notes* for specific information on memory requirements.

C0019 extra text after expected end number

A number was terminated by an invalid character.

Example:

```
i = 111aaa; // extra text after expected end of number
      ^
```

C0079 expected a type specifier

An expected type specifier was not found.

Example:

```
char * not_a_type;
int * ip = new not_a_type; // expected a type specifier
      ^
int * ip = new int;        // OK: 'int' is a valid type
```

C0082 storage class is not first

The storage class was not first in a declaration. This is an informational message in normal mode and a warning in strict ANSI mode.

Example:

```
int static i; // storage class is not first
  ^
```

C0083 type qualifier specified more than once

A type qualifier (**const** or **volatile**) was specified more than once in a declaration.

Example:

```
// type qualifier specified more than once
const const int i = 1;
  ^
```

C0084 invalid combination of type specifiers

An invalid combination of type specifiers (such as **short** and **float**) was used in the declaration.

Example:

```
// short cannot be used with float
short float f;
  ^
```

C0085 invalid storage class for a parameter

The storage class specified for a parameter is not valid. Only **register** (C/C++) and **auto** (C++) are valid storage classes for parameters.

Example:

```
// invalid storage class for a parameter
extern int f(static int i);
             ^
```

C0086 invalid storage class for a function

The storage class specified for the function is not valid. Only **extern** and **static** are valid storage class specifiers for a nonmember function.

Example:

```
// register cannot be specified as the storage class
// for a function
register int f();
      ^
```

C0095 array is too large

The size of the array, which is the product of the size of one element multiplied by the declared number of elements, is larger than the largest value of type **size_t**. Therefore, the array cannot fit in the address space of the target processor. For example, in 68K systems the maximum size is 4,294,967,295 bytes.

C0097 a function may not return a value of this type

Legal return values are:

- **void**
- Object types other than arrays
- References to object types other than arrays

You cannot return functions or arrays, but you can return pointers to functions or pointers to the first element of an array.

C0098 an array may not have elements of this type

The element type cannot be incomplete; for example:

```
struct to_be_named;
to_be_named team[10];
```

The element type cannot be a type that cannot be put into an array (**void**, a function, or a reference). In general, do not use anything that cannot be used in a **sizeof(type)** expression.

C0103 class is too large

The total size of the class declared is larger than the maximum value of type **size_t**. Therefore, the class or struct will not fit in the address space of the target processor. For example, in 68K systems the maximum size is 4,294,967,295 bytes.

C0104 struct or union is too large

The total size of the struct or union declared is larger than the maximum value of type **size_t**. Therefore, the class or struct will not fit in the address space of the target processor. For example, in 68K systems the maximum size is 4,294,967,295 bytes.

C0105 invalid size for bit field

Bit fields cannot have negative sizes, nor can they have sizes that are larger than the number of bits supported for their type (**char**, **short**, **int**, and so forth).

For example, in 68K systems no bitfield can have more than 32 bits.

C0106 invalid type for a bit field

Bit fields have to be declared with integral types (**char**, **short**, **int**, **long**, and their unsigned variants).

C0120 return value type does not match the function type

The value of the expression in **return** *expression* must be the same as, or convertible to, the value declared for the function.

C0126 expression must be an lvalue

For a definition of **lvalue**, see *The Annotated C++ Reference Manual*. The context in which this diagnostic was issued required an object that was (or can be considered to be) allocated to memory, so that its address could be taken. A literal object will not suffice.

C0137 expression must be a modifiable lvalue

For a definition of **lvalue**, see *The Annotated C++ Reference Manual*. The context in which this diagnostic was issued required an object that could be modified. Lit-

eral constants are not acceptable; neither are functions nor objects with the **const** attribute.

Example:

```
#include <string.h>
...
*strcpy = *strcat;
```

You cannot copy the function pointed at by `strcat` into the space allocated for the function `strcpy`. This is, of course, meaningless anyway.

Example:

```
const int x = 1;
x = 2;
```

`x` is **const**, therefore you cannot assign it a value.

C0153 expression must have class type

The expression that received the diagnostic contains a field selection operator (“.” or “.*”). If the second operand was the name of a member function, the first operand should be an object of a class that has a member of that name.

C0154 expression must have struct or union type

The expression that received the diagnostic contains a field selection operator (“.”) and the first argument is not an object of a struct or union type.

C0155 old-fashioned assignment operator

An obsolete K&R C style assignment operator (for example, ‘=–’ instead of ‘-=’) was used.

C0156 old-fashioned initializer

An obsolete K&R C style initializer (without ‘=’ between the variable and the initial value) was used.

C0158 expression must be an lvalue or a function designator

For a definition of **lvalue**, see *The Annotated C++ Reference Manual*. This diagnostic is issued when the argument of the unary operator (&) (that is, the **address_of** operator) is not something that has an address.

Example:

```
int *x = &7;
```

C0164 name of directory for temporary files is too long ("xxxx")

The length of a temporary file (composed of a directory name specified by the user via the **MRI_68K_TMP** environment variable, plus a filename generated by the C++ compiler) exceeds the internal buffer capacity of 150 bytes.

C0170 pointer points outside of underlying object

The result of a pointer arithmetic operation (pointer +/- constant) has produced a pointer whose validity cannot be guaranteed.

Example:

```
int x[10];
int *y;

y = x+2;    /* legal */
y = x-2;    /* illegal, outside x[] */
y = x+11;   /* illegal, outside x[] */
y = x+10;   /* legal, _just past_ x[] */
```

The second and third assignments are illegal, because the pointer so constructed (*y*) does not clearly point to an `int` object. The last example is legal, because obtaining the address of the object just beyond the end of an array is legal. It is not legal to dereference that pointer, but it is legal to use it in comparisons and further arithmetic.

C0172 external/internal linkage conflict with previous declaration

The variable/function flagged by this diagnostic has had a prior definition (perhaps in a header file), and the current (re)definition is changing the linkage.

Example:

```
static int x;  
extern x;
```

Example:

```
extern int f(int);  
  
static int  
f(int) {  
    ...  
}
```

C0224 the format string requires additional arguments

When using **printf** or **scanf**, too few arguments are supplied.

C0225 the format string ends before this argument

When using **printf** or **scanf**, too many arguments are supplied.

C0242 multiple access control specifiers are not allowed**Example:**

```
class B: public  
        private A {};
```

Only one access control specifier (**public**, **private**, or **protected**) can be specified.

C0261 access control not specified ("xxxx" by default)**Example:**

```

class B:A {
    ...
};
// Access control of A is not specified.
// It is "private" by default

```

C0267 old-style parameter list (anachronism)

A K&R C nonprototype style parameter declaration was used.

Example:

```

void f(a)
int a;
{
    ...
}

```

C0271 access adjustment in a "private" section is not allowed

You cannot change the accessibility of an inherited member.

Example:

```

class A { public:
    int a;
};
class B: public A {
    private: A::a;    // illegal to change from
};                  // public to private

```

C0275 invalid access declaration -- "entity" is hidden by "entity"

It is invalid to change the access control of a hidden entity.

Example:

```

struct A {
    private: void foo(){}
};
struct B: public A {
    public: void foo(){}
};
struct C: public B {
    public A::foo;    // Incorrect. A::foo is hidden by B::foo.
};

```

C0283 qualified name is not allowed

A qualified name is used where it is not permitted by the C++ syntax.

Example:

```
struct A {};  
struct B: public A{  
    typedef A::f;  
};    // Qualified name A::f is not allowed in B.
```

C0296 invalid use of non-lvalue array

An array returned by a function is not an **lvalue**.

Example:

```
struct A {  
    char name[5];  
    A(){}  
};  
char *f() {  
    return A().name;  
}
```

C0297 expected an operator

After the **operator** keyword, an operator (+, -, ~, >>, and so forth) is required.

Example:

```
struct A {  
    operator f();  
};    // f is not a valid operator name.
```

C0298 inherited member is not allowed

A derived class cannot define a member function that is declared in one of its base classes.

Example:

```
struct A {  
    void f();  
};  
struct B: public A {};  
void B::f(){}  
// error: cannot define B::f, which is inherited from A.
```

C0308 more than one instance of *entity-kind* "*entity*" matches the argument list:

When more than one instance of a template matches the argument list, an error occurs and all matching template instances are printed after the error message.

C0309 more than one instance of constructor "*entity*" matches the argument list:

When more than one instance of a constructor matches the argument list, an error occurs and all matching constructors are printed after the error message.

C0335 linkage specification is not allowed

A linkage specification (**extern** or **static**) is not permitted in the current context.

Example:

```
struct A {  
    extern "C" A();  
};    // extern "C" is not allowed; member functions  
      // always have C++ linkage.
```

C0336 unknown external linkage specification

Only "C" and "C++" linkages are supported.

Example:

```
extern "D" void f();  
// D is not recognized.
```

C0337 linkage specification is incompatible with previous "*entity*"

A declaration contains a linkage specification that is different from the previous declaration.

Example:

```
extern int a;  
static int a;  
// The second declaration is incompatible with the first one.
```

C0338 more than one instance of *entity-kind* "entity" has "C" linkage

Functions declared with "C" linkage cannot be overloaded.

Example:

```
extern "C" void f(int);
extern "C" void f(char);
```

C0348 more than one user-defined conversion from "type" to "type" applies:

This diagnostic is followed by a list of all possible conversion functions. This happens during the resolution of overloaded functions.

C0350 more than one operator "xxxx" matches these operands:

This diagnostic is followed by a list of all possible operators. This happens during the resolution of overloaded functions.

C0366 *entity-kind* "entity" provides no initializer for:

This diagnostic is followed by a list of the related **const** and **reference** data members. A user-defined constructor of some class that has **const** or **reference** data members must have initializers for the data members listed.

C0368 *entity-kind* "entity" defines no constructor to initialize the following:

This diagnostic is followed by a list of the related **const** and **reference** data members. When a class contains **const** or **reference** data members, a user-defined constructor is required.

C0379 cast of bound function to normal function pointer (anachronism)

A cast of a pointer to a member function to a normal function pointer is illegal.

Example:

```
struct A {
    int f();
};
A *p = new A;
int (*pf)() = (int(*)())p->f;
    // Incorrect. pf is a regular function pointer.
int (A::*pmf)() = p->f;
    // pmf is a pointer to a member function in A.
```


C0395 single-argument function used for postfix "xxxx" (anachronism)

Older versions of C++ did not permit the prefix and postfix operators to be overridden independently. The definition of an overloaded operator that was used for both is now only the prefix operator. See *The Annotated C++ Reference Manual*.

Example:

```
struct A {
    operator ++();
    operator --();
};
void f() {
    A x;
    x++;    // warning 0395
    x--;    // warning 0395
}
```

C0396 access adjustment is not allowed — mixed accessibility for *entity-kind* "entity"

When a base class contains overloaded functions with different accessibility, you cannot modify the accessibility in a derived class.

Example:

```
class A {
    public: int foo();
    private: int foo(char);
};
class B: private A {
    public: A::foo;
};
// A::foo has two different access modes
```

C0397 implicitly generated assignment operator cannot copy:

This message is given when a class contains **reference** or **const** data members; an implicitly generated assignment operator cannot handle this case.

C0416 more than one constructor applies to convert from "type" to "type":

This diagnostic is followed by a list of all possible conversion functions.

C0417 more than one conversion function from "type" to "type" applies:

This diagnostic is followed by a list of all possible conversion functions.

C0418 more than one conversion function from "type" to a built-in type applies:

If class A and B have a conversion operator to **int** and class C is derived from A and B, the conversion from C to **int** is ambiguous.

This diagnostic is followed by a list of all possible conversion functions.

C0420 reference *entity-kind* "entity"

This is not a stand-alone error message; this is always issued after some other error message as a second-line explanation.

C0421 *entity-kind* "entity"

This is not a stand-alone error message; this is always issued after some other error message as a second-line explanation.

C0456 excessive recursion at instantiation of *entity-kind* "entity"

A loop in template instantiation occurred.

Example:

```
template <class X>
int foo(X n) {
    return foo (&n);
}
// Calling foo (an_int) creates infinite recursion on
// instantiation of foo(int), foo(int*), foo(int**)
// and so forth
```

C0469 tag kind of `xxxx` is incompatible with declaration of *entity-kind* "*entity*" (declared at line `xxxx`)

You cannot use the same name tag for a **struct**, **class**, **enum**, **union**, or **typedef**.

Example:

```
class A {};  
enum A {x,y};  
// A is reused.
```

C0500 extra argument of postfix "*operatorxxxx*" must be of type "*int*"

The extra argument for post-increment and post-decrement operators must be an **int**. (For more information, see *The Annotated C++ Reference Manual*.)

C0511 enumerated type is not allowed

The **enum** promotes to integer for arithmetic operations, and it cannot then be converted back to **enum**. An integer cannot be converted to an **enum** automatically. (For more information, see *The Annotated C++ Reference Manual*.)

C0520 initialization with "{...}" expected for aggregate object

An initializer for an aggregate type must be surrounded by braces { }.

C0521 pointer-to-member selection class types are incompatible ("*type*" and "*type*")

Pointer-to-member types cannot be converted to other types, for example, to an **int**.

Example:

```
class A; class B;  
int foo(int A::*p, B*x) {  
    return x->*p; // incompatible with int  
}
```

C0522 pointless friend declaration

There is no need to define a member function as a friend. Also, there is no need to define a class as its own friend.

C0525 a dependent statement may not be a declaration

In Cfront-compatibility mode, the following statement is an error:

```
if (n) int x;
```

C0546 transfer of control bypasses initialization of:

This diagnostic is followed by a list of variables. These are not initialized, since control is transferred around their declaration.

C0553 external/internal linkage conflict for *entity-kind "entity"* (declared at line xxxx)

The function declaration says that the function is static, but the template declaration says the function is external.

Example:

```
static int foo(int);  
template <class T>  
int foo(T f);
```

C0554 *entity-kind "entity"* will not be called for implicit or explicit conversions

Conversions to the same type or a reference to the same type or to a base class or a reference to a base class is not used for conversions. The only way the conversion can be used is by explicitly invoking it.

Example:

```
struct A {  
    operator A();  
} a;  
.  
.  
.  
a.operator A();    // This is the only way to invoke  
                  // the A-to-A conversion operator.
```

C0561 invalid macro definition:

This diagnostic is followed by either command line or **#pragma** input.

Examples:

```
cccfamily t.cc -D123=1
```

where *family* represents the processor family, or in the file *t.cc*:

```
#pragma option -D123=1
```

Since the identifier “123” starts with a digit, the preceding examples result in an invalid macro definition.

C0562 invalid macro undefinition:

This diagnostic is followed by either command line or **#pragma** input.

Examples:

```
cccfamily t.cc -U123=1
```

where *family* represents the processor family, or in the file *t.cc*:

```
#pragma option -U123=1
```

Since the identifier “123” starts with a digit, the preceding examples result in an invalid macro undefinition.

C0625 #pragma xxxx

These messages are issued in response to **#pragma**, **error**, **warning**, or informational directives.

Examples:

```
#pragma error ...  
#pragma warning ...  
#pragma inform ...
```


Glossary: Appendix B

Abstract Class	A class containing a general structure that can be used only as a base class for producing specific implementations of the structure in derived classes.
Anachronism	A feature currently supported by C++ for backward compatibility with the ANSI C language or earlier C++ versions. This feature might not be supported in future releases.
Base Class	A group of data items declared in a class structure. Other classes can be derived from a base class (also known as the parent class).
Call Chain	A list of the functions linked in reverse order of their calling.
catch	A keyword in C++ that indicates a section of code for handling a particular error or exception condition. Catch statements can appear only at the end of a try block. When a catch statement is executed, it means that an exception was raised and the exception type matched the argument of this particular catch statement.
Catch-All Handler	An exception handler that contains statements designed to handle all exceptions. A catch-all handler catches any exception that reaches it: <pre>catch (...) { ... }</pre>
Class	A user-defined type. struct and union are also classes in C++.
Class Members	See Data Members, Member Functions.
Class Template	A template that defines a parameterized class. For example, Stack is a class template in the following declaration: <pre>template <class T> class Stack { };</pre>

const Member Function	A member function defined with the const keyword (see also Member Functions). Whenever a const member function is invoked, only const operations are performed; const operations do not change the value of the object. const member functions can be invoked for both const and non- const objects, whereas non- const member functions are not permitted on const objects.
const object	An object defined with the const keyword. The value of this object cannot be changed; only const operations (such as reading) can be performed on this object. const objects cannot call non- const functions.
Constructor	A constructor is a special member function used to initialize or construct a class object. It has the same name as the class.
Current Exception	An exception that has been thrown but has not been completely handled. The current exception is not considered handled until the program exits the handler.
Data Members	Data defined in a given class , struct , or union are called data members or class members.
Data Template	A static data member of a class template.
delete Operator	The delete operator destroys and deallocates the object created by the new operation.
Demangling	The act of decoding a C++ mangled name.
Derived Class	If a class has one or more parent classes, the class is said to be derived from those parent classes and is sometimes referred to as a derived class.
Destructor	A destructor is a special member function that destructs (deletes) class objects that need to be destroyed.
Exception Handler	A section of code designated to handle a particular error or exception condition. When a particular exception occurs, the handler can put the run-time system into a stable state without having to terminate program execution.

Exception Object	An object created by a throw expression that is compared to the argument of catch statements in order to match an error or exception to a handler. See also Thrown Object.
Exception Specification	A list of the exceptions that a particular function can raise to the next try block.
friend Class	If class <i>X</i> is a friend of class <i>Y</i> , all the member functions of friend class <i>X</i> are friend functions of class <i>Y</i> .
friend Function	A friend function of a class is a function that is not a member of the class but is allowed to access the private/protected members of the class.
Function Signature	To enforce cross-module parameter type checking and overloaded functions, Microtec Compiler C++ encodes the argument type information into the C++ function linkage name.
Function Template	<p>A template that defines a generic function. For example, swap is a function template in the following declaration:</p> <pre>template <class T> void swap(T &x, T &y);</pre> <p>A member function of a class template is also called a function template.</p>
Global Scope	To have the data variable, function, or class available to be called from any level in the program.
Handler	See Exception Handler.
Inheritance	A derived class can inherit properties of its base class(es). The inheritable properties include data members and member functions from base class(es). A derived class can also override class members of the base class(es) or declare additional class members of the base class(es). If a class is derived from one base class, such an inheritance relation is called single inheritance. If a class is derived from more than one base class, such an inheritance relation is called multiple inheritance.

Mangling	Microtec Compiler C++ encodes type information in the linkage names it generates for function names, local variable names, and so forth. These encoded names are known as mangled names. Mangled function names are also known as function signatures.
Match	For exception handling, the pairing of an exception to the correct handler. The exception object type and the catch expression argument are compared for the match.
Member Functions	Operations defined on a given class , struct , or union are called member functions.
Name Demangling	See Demangling.
Name Mangling	See Mangling.
new Operator	The new operator creates and allocates space for an object of a given class.
Overloaded Function	Two or more functions that perform different operations can be defined with the same name. A function that uses the same function name as the other function but has different parameter types from the other function is called an overloaded function. An overloaded function can be a member function or nonmember function. All overloaded functions should have the same return type.
Overloaded Operator	C++ allows most operators to be overloaded. You can define an overloaded operator function with arguments of user-defined types such as a complex arithmetic class type. All overloaded operators should have the same return type.
Pointer to Member	Pointer to a data member or a member function in a class.
Pure Virtual Function	A virtual function declared in a base class that does nothing. The details of the implementation can be left to subsequent derivations of the class.
Raise An Exception	See Throw Expression.

Rethrow	When an exception has been raised, that exception can be thrown again from inside of an exception handler or a custom unexpected exception function. A rethrow gives an opportunity for an exception to be handled by more than one handler.
Special Template Instance	<p>A user-defined template instance. Usually this has a different implementation from the generic template. For example, if Stack<int> is to be implemented differently from other generic instances, the following syntax could be used to define a special instance.</p> <pre> class Stack<int> { /* special definition */ }; void Stack<int>::pop() { /* special definition */ }; </pre> <p>When a special template instance exists, a C++ compiler does not generate the same instance from the generic template definition.</p>
Static Data Members	A static data member's space is shared among all objects of the class. Static data members are scoped to the class. A static data member can be accessed using the scope operator.
Static Member Function	A static member function is scoped to the class. It is global to all objects of the class and, thus, does not have and does not need to be invoked with a this pointer.
Template Class	An instance of a class template. For example, class Stack<int> .
Template Data	A static data member of a template class. For example, Stack<int>::data .
Template Declaration	The declaration part of a class or function template.
Template Definition	The definition part of a class, function, member function, or static data member template.
Template Function	An instance of a function template or member function of a template class. For example, swap<int> and Stack<int>::pop() .
Template Instance	A template class, template function, or template data.

Template Instantiation	The creation of a template instance. Microtec Compiler C++ generates template instances at compile time.
this Pointer	The handle or the implicit pointer to an object of a given class. Each member function of the given class (except static member functions) has the this pointer as its first implicit argument.
throw	A keyword in C++ that indicates an error or other exception conditions exist. When a throw expression is executed, the exception handling system searches the appropriate catch block, which matches the type of the object thrown.
Throw Expression	A C++ expression that raises an exception. The throw expression specifies the object type that is used to match the exception to the appropriate exception handler.
Thrown Object	An object created by a throw expression that is compared to the argument of catch statements in order to match an error or exception to a handler. The object can contain information that describes the exceptional condition. Also called exception object.
Try Block	A block of statements that contain throw expressions or calls to functions that contain throw expressions. The try block indicates to the compiler that exceptions can occur in this section of code and that the handlers for these exceptions appear in catch blocks that follow the try block.
Type-Safe Linkage	For C++ functions, the mangled name encodes the name of the function and the types of its parameters. A function definition matches a use of the function only if the mangled names match. This ensures that the caller and the function definition agree on parameter types and members, thus avoiding common errors.
Unexpected Exception	An exception that was raised in a function that did not list the exception type in its exception specification list. The exception handling system reacts to an unexpected exception by calling the unexpected library function.

Unhandled Exception

An exception that was never matched to a handler. The exception handling system reacts to an unhandled exception by calling the **terminate** library function.

Virtual Function

A class member function whose implementation is dependent on its class type. The details of the implementation can be left to subsequent derivations of the class.

volatile Member Function

A member function defined with the **volatile** keyword (see also Member Functions). All **volatile** objects can invoke only **volatile** member functions, insuring that all operations of the **volatile** object are treated as **volatile**. **volatile** member functions will not be optimized.

volatile object

An object defined using the **volatile** keyword. A **volatile** object can change independent of the program flow. Optimization is disabled for operations on these objects.

Symbols

- __ _ _ FPU preprocessor symbol 3-33
- __pure_virtual_function_called function 5-47
- __pure_virtual_function_called function 5-47
- +delete_std option 3-58
- +lac++ option 3-59
- +lc++ option 3-59
- +lf2 option 3-59
- +lf2o option 3-59
- +lf3 option 3-59
- +lf3o option 3-59
- +tl option 3-59
- +tm option 3-60
- +tu option 3-60
- 16-bit bus 11-17, 11-20
- 16-bit displacement 3-52
- 16-bit extension on stack
 - Zp2 option 3-51
- 32-bit bus 11-18, 11-20
- 32-bit displacement
 - Ml option 3-52
 - Mlc option 3-52
 - Mld option 3-53
 - Mls option 3-53
- 32-bit extension on stack
 - Zp4 option 3-51

A

- A option 3-20
- Absolute addressing 13-23
 - Mca option 3-51
 - Mda option 3-52
- Abstract class B-1
- acc option 3-54
- acd option 3-54
- acos function 3-19

Address modes

- code references
 - absolute
 - Mca option 3-51
 - PC-relative
 - Mcp option 3-51
- const section
 - ac options 3-54
- data references
 - absolute
 - Mda option 3-52
 - PC-relative
 - Mdp option 3-52
 - register-relative
 - Md options 3-51
- initialized data section
 - ai options 3-54
- literals section
 - al options 3-54, 3-58
- zerovars section
 - az options 3-54

Addressing

- absolute 13-23
- PC-relative 13-25
- register-relative 13-23

Aggregates 11-14

- aic option 3-54
- aid option 3-54
- alc option 3-54, 3-58
- ald option 3-54, 3-58, 3-61, 3-62

Algebraic simplification 7-1

- alias command (Linker) 3-57

Aliased references

- Ob option 3-40

Alignment

- aggregates 11-14
- array 11-14
- bit fields 11-12, 11-18
- data types 11-19
- padding bytes 11-20

- structure members
 - Z options 3-31
- structures 11-14, 11-20
- trailer bytes 11-20
- unions 11-14
- Zm option 3-51
- Alignment of struct/union in parameter area 10-4
- alloca function 5-13
- Allocating
 - data space 5-68
 - data types 11-2
 - memory space 5-51
- Alternate locations
 - UNIX
 - executables 2-3
 - libraries 2-3
 - standard include files 2-3
 - temporary files 2-3
- Anachronism B-1
- ANSI-compliant mode, setting
 - A option 3-20
- Argument
 - passing 10-13
 - promotion 10-13
- Array operator synthesis optimization 7-7
- Arrays 11-4
- asc option 3-54
- ASCII character, testing for 5-32
- ASCII format
 - conversion from byte 5-62
- ASCII string
 - conversion from floating-point number 5-28
 - conversion from integer 5-33
 - conversion from long integer 5-36
 - conversion from unsigned integer 5-34
 - conversion from unsigned long integer 5-37
- asd option 3-54
- asin function 3-19
- asm pseudofunction 4-5, 13-2, 13-3
 - enabling
 - x option 3-19
- asm support 4-5
- Assembler
 - description 1-2
 - options, passing directly
 - Wa option 3-6, 3-15, 3-30
 - source file
 - (see Assembly source file)
- Assembler in-lining 4-5, 13-2
 - considerations 4-10, 13-8
- Assembly and high-level code
 - Fsm option 3-23
- Assembly code, optimizing by hand 14-1
- Assembly file
 - saving
 - H option 3-26
- Assembly language
 - example of a routine 10-9
 - interface 10-1, 10-8
- Assembly source file 14-1
 - advantages to producing 14-1
 - contents 14-3
 - expanded includes
 - Fsi option 3-23
 - generating
 - S option 3-29
 - high-level source code in comments
 - Fsm option 3-23
 - high-level source code, including as comments
 - Fsm option 3-23
 - line numbers 14-2
 - naming
 - o option 3-26
 - variable names 14-1
- atan function 3-19
- atanh function 3-19
- azc option 3-54

B

Backspace
 disable for preprocessor
 -Es option 3-25
 -Ps option 3-25
Bandwidth, data bus 11-12, 11-17
Banner
 -Vb option 3-30
Base class B-1
BIG_ENDIAN 11-1
big-endian 11-26
Bit fields 11-8
 alignment 11-12, 11-18
bld_lib script 6-6
Branch tail merging 7-11
Bus bandwidth 11-12, 11-17
Bytes
 clearing memory bytes 5-41
 conversion to ASCII format 5-62
 reading from a file 5-50
 swapping odd and even bytes 5-57
 writing to a file 5-67

C

C calling conventions 10-1
C comments, saving in preprocessor output
 -C option 3-25
C common 3-38
-C option 3-25
-c option 3-25
C system initialization file (csys.c) 13-37
C++ compiler
 description 1-1
C++ compiler package
 C++ compiler 1-1
C++ file suffix
 +z option 3-62
C++ language
 wrapper 10-16
Call stack
 definition B-1

Calling functions

 C++ from C 10-14 to 10-18
 member functions 10-14
 overloaded functions 10-14
 member functions 10-14
calloc function 13-11
catch keyword B-1
Catch-all handler
 definition B-1
char type 11-3
Character
 converting to lowercase 5-63
 converting to uppercase 5-64
 copying characters from memory 5-40
 testing for ASCII character 5-32
_CHAR_SIGNED preprocessor symbol 3-50
_CHAR_UNSIGNED preprocessor symbol
 3-50
chdir function 5-14
Checking
 extra
 -v option 3-30
 syntax only
 -y option 3-29
Class B-5
 base B-1
 definition B-1
 derived B-2
 friend B-3
 member B-1
 template B-1
Clearing memory bytes 5-41
close a pipe 5-45, 5-46
close function 5-15, 5-16
Closing a file 5-15
CLR instruction, using
 -Kc option 3-47, 3-49
Code
 position-dependent 13-18
 position-independent 13-18
Code and data section names, specifying 3-55
Code elimination, unreachable 7-2
Code hoisting 7-12

- Code optimization
 - OL option 3-42
- Code organization 12-1
 - compiler-generated sections 12-3
- Code references
 - absolute
 - Mca option 3-51
 - PC-relative
 - Mcp option 3-51
- code section 3-53, 12-2, 12-5
 - naming
 - NT option 3-56
- ColdFire
 - Multiple and accumulator unit 3-33
 - mac option 3-33
- Column number debugging information
 - Gm option 3-34
- Command file, passing to linker
 - e option 3-26
- Command line options
 - cleanup suspension
 - +delete_std option 3-58
 - specifying in file
 - d option 3-26
 - UNIX 2-1
 - Windows 2-1
- Comments, saving in preprocessor output
 - C option 3-25
- Comparing values 5-39, 5-42
- Compiler
 - command line 2-1
 - description 1-2
 - invoking 2-1
- Compiler features 1-1
- Compiler options
 - UNIX/DOS
 - e 13-22
 - G1 14-3
 - Kf 7-17
 - Kh 13-32
 - Kr 13-43
 - Ks 13-16
 - Ku 11-4
 - Mcp 13-18, 13-19
 - Mdn 13-18, 13-19
 - Mdp 13-18, 13-19
 - nOc 7-18
 - Og 7-11
 - Os 7-1
 - Ot 7-1
 - Q A-4
 - Qe A-4
 - Qi A-4
 - Qms A-4
 - Qs A-4
 - Qw A-4
 - v A-21
 - Zp2 11-6
 - Zp4 11-6
- Compiler output
 - assembly source file 14-1
 - listing 14-3
- Compiler-generated literals section 3-53
 - naming
 - NL option 3-55
 - specifying address mode
 - al options 3-54, 3-58, 3-61, 3-62
- Compiler-generated tag data section 3-53
 - Kt option 3-50
- COMPILER_HOME 2-4, 2-5, 3-21, 3-22
- Complex data types 11-4
- Concurrency control
 - and precompiled header files 9-5
- Configuration file 6-3
- Configurations, memory 13-15, 13-42
- Conflicting options, specifying 3-1
- const
 - member function B-2
 - object B-2
- const section 3-53, 12-2, 12-5
 - naming
 - NC option 3-55
 - specifying address mode
 - ac options 3-54
- Constant folding optimization 7-10

- Constant literals section
 - al options 3-54, 3-58
- Constant propagation
 - ic 3-39
- Constant variables section
 - (see const section)
- Constants
 - floating-point size
 - Kq option 3-49
- Constructor
 - definition B-2
- Copying characters from memory 5-40
- cos function 3-19
- cosh function 3-19
- creat function 5-17
 - relationship to close 5-15
 - relationship to read 5-50
 - relationship to write 5-67
- Creating a file 5-17
- Cross-checking in C++ 10-12
- csys.c 13-37
- Current Exception B-2
- cxx_edt section 3-53, 12-2
- _cxxfini function 5-18
- cxxfini function 5-18
- cxx_rtti section 3-53, 12-2

D

- D option 3-24
- d option 3-26
- Data
 - formats xxiv, 11-1
 - initialization 13-37
 - member B-2
 - position-dependent 13-18
 - system 13-35
 - template B-2
 - types 11-2
 - allocation 11-2
 - ranges 11-2
- Data browser
 - GS option 3-35
- Data bus bandwidth 11-12, 11-17

- Data references
 - absolute
 - Mda option 3-52
 - PC-relative
 - Mdp option 3-52
 - register-relative
 - Md options 3-51
- Data space
 - allocating 5-68
- Data type alignment 11-19
- Dead assignment elimination
 - id option 3-39
- Dead code elimination 7-2
- _DEBUG preprocessor symbol 3-35
- Debugging information
 - fully qualified pathnames
 - Gf option 3-34
 - generating
 - g option 3-35
 - line number information
 - Gl option 3-34
 - multiple statements on line
 - Gm option 3-34
 - preprocessor macros
 - Gd option 3-33
 - restricted
 - Gr option 3-34
 - XRAY Source Explorer
 - GS option 3-35
 - Gs option 3-34
- Default initialization 11-26
- #define directive 4-8, 13-5
- #define preprocessor directive 9-2
- Defining macros on command line
 - D option 3-24
- delete operator B-2
- Demangling B-2
- Derived classes
 - definition B-2
- Destructor
 - definition B-2
- Diagnostic messages 3-36
 - (see also Messages, diagnostic)

-
- Disable
 - backspace
 - Es option 3-25
 - Ps option 3-25
 - newline
 - Es option 3-25
 - Ps option 3-25
 - optimizations on global variables
 - Ob option 3-40
 - stack frame sharing
 - Kf option 3-47, 3-48
 - Display time stamp
 - Vt option 3-30
 - DOS
 - option, command line
 - descriptions of 3-1
 - double type 11-3
 - Duplicate function names 10-12
 - E**
 - +E option 3-35
 - E option 3-25
 - +e option 3-58
 - e option 3-26, 13-22
 - _ehs_cleanup_area function 5-19
 - _ehs_init_area function 5-20
 - _ehs_restore_from_area function 5-21
 - _ehs_save_restore_area function 5-22
 - _ehs_save_size function 5-23
 - _ehs_save_to_area function 5-24
 - Embedded systems
 - code organization 12-1
 - considerations 13-1
 - user-modified routines 13-27
 - Embedded systems and exception handling 13-48
 - enum type 11-3
 - Environment variables 2-3
 - UNIX
 - MRI_68K_BIN 2-3, 2-5
 - MRI_68K_INC 2-3, 2-6
 - MRI_68K_LIB 2-3, 2-6
 - MRI_68K_TMP 2-3, 2-6
 - Windows
 - MRI_68K_BIN 2-3, 2-5
 - MRI_68K_INC 2-3, 2-6
 - MRI_68K_LIB 2-3, 2-6
 - MRI_68K_TMP 2-3, 2-6
 - Epilogue 7-17
 - function 4-1
 - generating code for epilogue 7-17
 - Epilogue, function 10-7
 - fprintf 5-25
 - fprintf function 5-25
 - Error messages A-1
 - ID numbers
 - Qfn option 3-36
 - severity
 - Qme option 3-37
 - Qmi option 3-37
 - Qms option 3-37
 - Qmw option 3-37
 - source line numbers
 - Qfs option 3-36
 - suppression
 - Qa option 3-36
 - Qe option 3-36
 - Es option 3-25
 - Examples
 - calling C functions from C++ 10-12
 - calling C++ member functions from C 10-15
 - interrupt keyword 4-2
 - overloading function names 10-12
 - Exception handler B-2
 - catch all B-1
 - Exception handling 3-66
 - ze 3-60
 - Exception object B-3
 - Exception specification B-3
 - types
 - database of 13-49
 - EXCLUDE_AT_EXIT preprocessor symbol 6-5
 - EXCLUDE_ERRNO preprocessor symbol 6-5

- EXCLUDE_FORMAT_IO_ASSGN_SUPP

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_BRAKT_FMT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_CHAR_FMT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_DEC_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_FLOAT_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_HEX_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_h_OPT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_INT_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_L_OPT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_l_OPT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_MINUS_FLAG

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_NUMB_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_OCT_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_PLUS_FLAG

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_PNT_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_SHARP_FLAG

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_SPACE_FLAG

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_STAR_OPT

```
processor symbol 6-4
```
 - EXCLUDE_FORMAT_IO_STR_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_UN\$S_FMT

```
processor symbol 6-5
```
 - EXCLUDE_FORMAT_IO_ZERO_FLAG

```
processor symbol 6-4
```
 - EXCLUDE_INITDATA

```
processor symbol 6-5
```
 - EXCLUDE_IOB

```
processor symbol 6-5
```
 - EXCLUDE_LINE_BUFFER_DEFAULT

```
processor symbol 6-5
```
 - EXCLUDE_RAND

```
processor symbol 6-6
```
 - EXCLUDE_SIGNAL_RAISE

```
processor symbol 6-6
```
 - EXCLUDE_STRTOK

```
processor symbol 6-6
```
 - EXCLUDE_TERMINAL_SIMULATION

```
processor symbol 6-5
```
 - EXCLUDE_TIME

```
processor symbol 6-6
```
 - Executable file
 - suppressing
 - c option 3-25
 - Executables
 - UNIX
 - alternate locations 2-3
 - Executing only

```
processor
```
 - E option 3-25
 - P option 3-25
 - _exit function 5-26
 - exit function
 - relationship to _exit 5-26
 - exp function 3-19
 - Explorer, generating code for
 - GS option 3-35
 - Gs option 3-34
 - Extensions
 - filename 2-2
 - Microtec compiler
 - (see Microtec compiler extensions)
 - Extensions to C
 - x option 3-19
 - extern "C" declaration 10-12
 - External variable names 14-1
- F**
- f option 3-33
 - fabs function 3-19
 - Features of compiler 1-1
 - Fee option 3-36

- Feo option 3-36
 - File
 - closure 5-15
 - creation 5-17
 - making file inaccessible 5-66
 - opening 5-43
 - unlinking a filename 5-66
 - File descriptor, getting 5-27
 - fileno macro 5-27
 - Files
 - #include
 - (see #include files)
 - input
 - extensions 2-2
 - locations 2-2
 - linker command 3-26
 - listing
 - (see Listing files)
 - output
 - locations 2-2
 - Fli option 3-23
 - float type 11-3
 - Floating-point
 - number
 - conversion to ASCII string 5-28
 - removing unneeded support 13-32
 - Floating-point constant size
 - Kq option 3-49
 - Floating-point coprocessor instructions,
 - generating
 - f option 3-33
 - Flp option 3-23
 - Flp0 option 3-23
 - Flt option 3-23
 - Formatted output to standard error 5-25
 - _FPU preprocessor symbol 3-33
 - Frame pointer 10-5
 - free function 13-11
 - friend
 - class B-3
 - function B-3
 - functions
 - calling from C 10-14
 - Fsi option 3-23
 - Fsm option 3-23
 - ftoa function 5-28
 - Fully qualified names, generating for input files
 - Gf option 3-34
 - Function declaration
 - with interrupt keyword 11-6
 - Function in-lining 7-15
 - Function names, truncating
 - ut option 3-33
 - Function template B-3
 - Functions
 - const member B-2
 - duplicate names 10-12
 - epilogue 4-1, 10-7
 - friend B-3
 - library
 - (see Library functions)
 - member B-4
 - non-reentrant 5-8
 - overloaded B-4
 - prologue 4-1, 10-6
 - local variables in prologue 10-7
 - pure virtual B-4
 - returning from
 - Oe option 3-41
 - signature 10-12, B-3
 - static member B-5
 - tagging entry and exit points
 - Kt option 3-50
 - template B-5
 - virtual B-7
 - volatile member B-7
- ## G
- g option 3-35
 - Gd option 3-33
 - General optimizations 7-1
 - Generating floating-point processor instructions
 - f option 3-33
 - getcwd function 5-29
 - getl function 5-30

- getw function 5-31
 - Gf option 3-34
 - Gl option 3-34, 14-3
 - Global constant propagation optimization 7-4
 - Global copy propagation 7-5
 - Global optimizations 7-2 to 7-9
 - Global Scope B-3
 - Global value propagation 7-5
 - Global variable optimization
 - Ob option 3-40
 - Global variables
 - uninitialized 3-38
 - Global-flow optimizer
 - Og option 3-41
 - Gm option 3-34
 - Gr option 3-34
 - Grouping stack adjust instructions optimization 7-18
 - GS option 3-35
 - Gs option 3-34
- ## H
- H option 3-26
 - Handler B-3
 - heap section 3-53, 12-2
 - High-level and assembly code
 - Fsm option 3-23
- ## I
- I option 3-20
 - i option 3-38
 - ic option 3-39
 - id option 3-39
 - #if preprocessor directive 9-2
 - #include directive 5-4
 - use in precompiled headers 9-3
 - Include files 5-4
 - UNIX
 - alternate locations 2-3
 - #include files
 - mriext.h 5-6
 - search path, specifying
 - I option 3-20
 - J option 3-22
 - INCLUDE_BUILD_ARGV preprocessor symbol 6-6
 - INCLUDE_FAST_POW preprocessor symbol 6-6
 - includes and assembly code
 - Fsi option 3-23
 - Index simplification optimization 7-9
 - Indexing array optimization 7-18
 - Induction variable elimination 7-9
 - Informational messages
 - suppression
 - Qi option 3-36
 - Inheritance B-3
 - multiple B-3
 - single B-3
 - INITDATA linker command 13-38
 - initfini section 3-53, 12-2, 13-44
 - Initialization 13-37
 - compile-time 13-37
 - data 13-37
 - default 11-26
 - run-time 13-37
 - static constructor 13-44
 - static destructor 13-44
 - static objects 13-44
 - static variables 11-26
 - Initialize local variables
 - KI option 3-48
 - Initialized data section 3-53
 - naming
 - NI option 3-55
 - specifying address mode
 - ai options 3-54
 - initvars section 12-5, 12-6
 - In-line assembly 4-5, 13-2
 - In-line assembly code
 - enabling
 - x option 3-19
 - In-line library function expansion 7-17

In-lining
 -Oi option 3-41
In-lining assembly instructions 4-5
In-lining run-time library functions
 -Oj option 3-42
Input and output files, location
 UNIX 2-2
 Windows 2-2
Input files
 extensions 2-2
 generating fully qualified names
 -Gf option 3-34
 locations 2-2
Instruction scheduling
 -Or option 3-44
int type 11-3
Integer
 conversion to ASCII string 5-33
Internal compiler errors A-16
Interrupt
 -Kr command line option 4-1
Interrupt handlers, declaring 10-11, 13-43
interrupt keyword 3-65, 4-1, 10-11, 11-6,
 13-43
Interrupt procedures
 -Kr option 3-49
 return with RTE instruction
 -nKr option 3-49
 return with RTS instruction
 -Kr option 3-49
Intrinsic system functions
 summary 5-10
Introduction to compiler package 1-1
Invocation
 compiler 2-1
I/O buffering
 buffered 5-3
 embedded environments 5-3
 native environments 5-3
 unbuffered 5-4
 unit buffered 5-4
I/O device registers 3-41
I/O static initialization 13-48

I/O static initialization and termination 13-48
I/O termination 13-48
ioports section 12-2, 12-6
isascii function 5-32
-it option 3-39
itoa function 5-33
itostr function 5-34

J

-J option 3-22
-jH option 3-21, 9-1, 9-3, 9-4
-jHc option 9-3, 9-4
-jHd option 9-3, 9-4
-jHs option 9-3
-jHu option 9-3, 9-4
Jump optimizations 7-11 to 7-15

K

-Kc option 3-47, 3-49
Keywords
 interrupt 3-65, 11-6
 packed 3-65
 typeof 3-65
 unpacked 3-65
-Kf option 3-47, 3-48, 7-17, 10-6
-Kh option 3-48, 10-5, 13-32
-KI option 3-48
-Km option 3-49
-KP option 3-49
-Kq option 3-49
-Kr command line option 4-1
-Kr option 3-49, 10-11, 13-43
-Ks option 13-16
-Kt option 3-50
-Ku option 3-50, 11-4

L

-l option 3-24
Labels for line numbers
 -Gl option 3-34
Language specifying 3-59
Librarian description 1-3

- Libraries 13-38
 - C++ I/O levels 5-1
 - UNIX
 - alternate locations 2-3
 - UNIX level-1+ elements 5-2
 - use 5-1
 - Library customizer
 - building custom libraries 6-6
 - configuration file 6-3
 - directories 6-1
 - preprocessor symbols 6-4
 - testing custom libraries 6-7
 - Library functions
 - (see also under specific names)
 - non-reentrant 5-8
 - read 13-30
 - write 13-30
 - Line number labels, producing
 - Gl option 3-34
 - Line number, variable, and symbol information
 - g option 3-35
 - Line numbers in assembly language source file 14-2
 - #line preprocessor directive 9-2
 - LINE_BUFFER_SIZE preprocessor symbol 6-5
 - Linker
 - alias command 3-57
 - command example
 - IEEE 13-39
 - ROM-based system, IEEE 13-41
 - command file
 - passing
 - e option 3-26
 - description 1-3
 - passing default libraries 3-29
 - passing options directly
 - Wl option 3-31
 - suppressing call to 3-25
 - Linker command files
 - use 13-38
 - Listing files
 - format
 - Fl option 3-23
 - generating
 - l option 3-24
 - omitting page header
 - Flp0 option 3-23
 - page length, specifying
 - Flp option 3-23
 - title
 - Flt option 3-23
 - literals section 3-53, 12-2, 12-5
 - naming
 - NL option 3-55
 - specifying address mode
 - al options 3-54, 3-58
 - Little 11-1
 - Local optimizations 7-10
 - Ol option 3-43
 - Local variable initialization
 - KI option 3-48
 - Local variables 11-26
 - log function 3-19
 - log10 function 3-19
 - long double type 11-3
 - Long integer
 - conversion to ASCII string 5-36
 - reading from a stream 5-30
 - writing to stream 5-48
 - long type 11-3
 - Loop optimizations 7-6
 - Lower-case characters
 - converting to 5-63
 - lseek function 5-35
 - ltoa function 5-36
 - ltostr function 5-37
- ## M
- Machine-dependent optimizations 7-17 to 7-19
 - Macros
 - defining on command line
 - D option 3-24

-
- undefining
 - U option 3-24
 - _main function 5-38
 - malloc function 13-11
 - Mangling B-4
 - extern "C" and 10-17
 - function names 10-12
 - overloaded functions and 10-14
 - preventing for C function names 10-12
 - Match B-4
 - math.h include file 13-11
 - max macro 5-39
 - Mca option 3-51
 - Mcp option 3-51, 13-18, 13-19
 - Md option 3-51
 - Mda option 3-52
 - Mdn option 13-18, 13-19
 - Mdp option 3-52, 13-18, 13-19
 - Member
 - function B-4
 - calling from C 10-14
 - memcpy function 5-40
 - memclr function 5-41
 - memcpy function 3-19
 - Memory configurations 13-15, 13-42
 - Memory space
 - allocating 5-51
 - Message severity levels A-2
 - Messages, diagnostic
 - displaying
 - nQ option 3-37
 - suppressing
 - Q option 3-36
 - writing to stderr
 - Fee option 3-36
 - writing to stdout
 - Feo option 3-36
 - Messages, error A-1
 - Messenger symbols 13-38
 - Km option 3-49
 - Microprocessor
 - (see Processor, specifying)
 - Microtec compiler extensions 3-19
 - Microtec extensions
 - functions 5-6
 - eprintf 5-25
 - ftoa 5-28
 - getl 5-30
 - getw 5-31
 - isascii 5-32
 - itoa 5-33
 - itostr 5-34
 - ltoa 5-36
 - ltostr 5-37
 - memcpy 5-40
 - memclr 5-41
 - putl 5-48
 - putw 5-49
 - swab 5-57
 - toascii 5-62
 - _tolower 5-63
 - _toupper 5-64
 - zalloc 5-68
 - macros
 - BLKSIZE 5-8
 - FALSE 5-8
 - fileno 5-27
 - isascii 5-32
 - max 5-39
 - min 5-42
 - NULLPTR 5-8
 - stdaux 5-8
 - stdprn 5-8
 - toascii 5-62
 - _tolower 5-63
 - _toupper 5-64
 - TRUE 5-8
 - x option 3-19
 - min macro 5-42
 - Mixed operands 11-4
 - MI option
 - generate position-independent code and data 3-52
 - Mlc option
 - generate position-independent code 3-52

- Mld option
 - generate position-independent data 3-53
- Mls option
 - generate position-independent switch statement jump tables 3-53
- Modes
 - address
 - (see Address modes)
 - ANSI-compliant
 - A option 3-20
 - processor
 - p option 3-27
 - verbose
 - V options 3-30
- Module, naming
 - NM option 3-55
- MRI_68K_BIN 2-3, 2-5
- MRI_68K_INC 2-3, 2-6
- MRI_68K_LIB 2-3, 2-6
- MRI_68K_TMP 2-3, 2-6
- mriehs.h include file 5-5
- mriext.h include file 5-6
- _MRI_EXTENSIONS preprocessor symbol 3-19
- Multiple inheritance B-3
- Multiple jump optimization 7-14
- Multitasking environment (MTE) 13-11, 13-32
- Multi-threaded environments 13-11
- N**
 - nA option 3-20
 - Name demangling B-4
 - Name mangling B-4
 - Names, truncating
 - ut option 3-33
 - Naming convention for symbols 3-32
 - Naming modules
 - NM option 3-55
 - NC option 3-55
 - nC option 3-25
 - nc option 3-25
 - nE option 3-25
 - +ne option 3-58
 - Negative option prefix 3-1
 - new operator B-4
 - new.h include file 5-6
 - Newline
 - disable for preprocessor
 - Es option 3-25
 - Ps option 3-25
 - nf option 3-33
 - nFee option 3-36
 - nFeo option 3-36
 - nFli option 3-23
 - nFsi option 3-23
 - nFsm option 3-23
 - ng option 3-35
 - nGf option 3-33, 3-34
 - nGl option 3-34
 - nGm option 3-34
 - nGr option 3-34
 - nGS option 3-35
 - nGs option 3-34
 - nH option 3-26
 - NI option 3-55
 - ni option 3-38
 - nic option 3-39
 - nid option 3-39
 - nit option 3-39
 - nKc option 3-47, 3-49
 - nKf option 3-47, 3-48
 - nKI option 3-48
 - nKm option 3-49
 - nKP option 3-49
 - nKq option 3-49
 - nKr option 3-49
 - nKt option 3-50
 - nKu option 3-50
 - NL option 3-55
 - nl option 3-24
 - NM option 3-55
 - nO option 2-2, 3-39
 - nOb option 3-40
 - nOc option 3-41, 7-18, 10-6
 - nOe option 3-41
 - nOg option 3-41

- nOi option 3-41
 - nOj option 3-42
 - nOl option 3-43
 - Nonmember function
 - calling from C 10-14
 - Non-reentrant functions
 - (see Reentrant functions)
 - nOR option 3-43
 - nOr option 3-44
 - nP option 3-25
 - nQ option 3-37
 - nq option 3-29
 - nQo option 3-37
 - NS option 3-56
 - NT option 3-56
 - nV option 3-30
 - nv option 3-30
 - nx option 3-19
 - ny option 3-29
 - NZ option 3-57
 - nZe option 3-50, 3-51
 - nze option 3-60
 - nZle option 3-51
- O**
- O option 3-39
 - o option 3-26
 - Ob option 3-40
 - Object construction
 - tracking 13-49
 - Object files
 - naming
 - o option 3-26
 - producing only
 - c option 3-25
 - Oc option 3-41
 - Oe option 3-41
 - Og option 3-41, 7-11
 - Oi option 3-41
 - Oj option 3-42
 - OL option 3-42
 - Ol option 3-43
 - open function 5-43
 - relationship to close 5-15
 - relationship to read 5-50
 - relationship to write 5-67
 - Opening a file 5-43
 - Optimization information
 - i 3-38
 - Optimizations 7-1
 - algebraic simplification 7-1
 - array operator synthesis 7-7
 - branch tail merging 7-11
 - code for epilogue 7-17
 - code for prologue 7-17
 - constant folding 7-10
 - cross-jump 7-13
 - general 7-1
 - global 7-2 to 7-9
 - global constant propagation 7-4
 - global copy propagation 7-5
 - grouping stack adjust instructions 7-18
 - indexing arrays 7-18
 - in-line library function expansion 7-17
 - instruction scheduling 7-17
 - jump 7-11 to 7-15
 - local 7-10
 - loop 7-6
 - machine dependent 7-17 to 7-19
 - multiple jump 7-14
 - redundant code elimination 7-2
 - redundant jump elimination 7-14
 - short/long displacement 7-15
 - strength reduction 7-2
 - strength reduction and index simplification 7-9
 - Optimizing code 3-39
 - execution time
 - Ot option 3-44, 3-45
 - global-flow optimizer
 - Og option 3-41
 - in-lining
 - Oi option 3-41
 - local optimizations
 - Ol option 3-43
 - OL option 3-42

- OX option 3-45
- register coloring
 - OR option 3-43
- size
 - Os option 3-44
- Option descriptions
 - UNIX 3-1
 - Windows 3-1
- Options
 - active on output listing
 - suppression
 - Qo option 3-37
 - and precompiled header files 9-3
 - cleanup suspension
 - +delete_std option 3-58
 - command line 3-1
 - conflicting, specifying 3-1
 - descriptions 3-18
 - file, specifying
 - d option 3-26
 - negative 3-1
 - passing to assembler
 - Wa option 3-6, 3-15, 3-30
 - passing to linker
 - Wl option 3-31
 - summary 3-1
- OR option 3-43
- Or option 3-44
- Os option 3-44, 7-1
- Ot option 3-44, 3-45, 7-1
- Output
 - assembly source file 14-1
 - listing 14-3
- Output files
 - locations 2-2
 - naming
 - o option 3-26
 - specifying format
 - Fli option 3-23
- OV option 3-45
- Overloaded functions 10-14
- Overloading
 - function B-4

- names 10-12
- operators B-4
- OX option 3-45

P

- P option 3-25
- p option 3-27
- packed keyword 3-65, 4-3, 11-16
- Packed structures 11-18
 - tips 11-23
- packed type 11-22
- Packing
 - tips 11-23
- Padding
 - bytes 11-20
 - structures 11-17
- Page header, specifying for listing file
 - Flp0 option 3-23
- Parameters
 - passing 10-1
 - popping 10-5
 - setting 10-1
- Passing options
 - to assembler
 - Wa option 3-6, 3-15, 3-30
 - to linker
 - Wl option 3-31
- .pch file extension 9-4
- PC-relative address mode
 - code references
 - Mcp option 3-51
 - data references
 - Mdp option 3-52
- PC-relative addressing 13-25
- _PIC preprocessor symbol 3-51
- _PID preprocessor symbol 3-52
- _PID_REG preprocessor symbol 3-52
- Pipeline instruction scheduling 7-17
- pixinit section 3-53, 12-2, 13-47
- Pointer
 - to member B-4
- Pointers 11-3
 - types 11-3

-
- Popping stack
 - Oc option 3-41
 - Position-dependent code 13-18
 - Position-dependent data 13-18
 - Position-independent code 13-18
 - Mcp option 3-51
 - Position-independent data
 - Md options 3-51
 - #pragma asm directive 4-10, 9-4, 13-7
 - #pragma endasm directive 4-10, 9-4, 13-7
 - #pragma hdrstop directive 9-1, 9-4
 - #pragma no_pch directive 9-2, 9-4
 - #pragma option directive 9-3, 9-4
 - #pragma options 11-21
 - Precompiled header files
 - header stop point 9-1, 9-2
 - precompiled header files
 - jH option 3-21
 - Predefined symbols
 - _PIC 3-51
 - _PID 3-52
 - _PID_REG 3-52
 - Preprocessor
 - executing only
 - E option 3-25
 - P option 3-25
 - macros
 - defining on command line
 - D option 3-24
 - undefining
 - U option 3-24
 - output
 - saving comments
 - C option 3-25
 - sending to standard output
 - E option 3-25
 - symbols
 - (see under specific name)
 - Preprocessor directives
 - #define 9-2
 - #if 9-2
 - #line 9-2
 - #pragma asm 9-4
 - #pragma endasm 9-4
 - #pragma hdrstop 9-1, 9-4
 - #pragma no_pch 9-2, 9-4
 - #pragma option 9-3, 9-4
 - Preprocessor macro debug information,
 - generating
 - Gd option 3-33
 - preprocessor symbol 6-5
 - Preprocessor symbols 6-4
 - printf function
 - preprocessor symbols 6-4
 - removing unneeded I/O support 13-31
 - Processor modes
 - p option 3-27
 - Processor, specifying
 - p option 3-27
 - Program identifiers, truncating
 - ut option 3-33
 - Program termination 5-26
 - Prologue 7-17
 - function
 - generating code for prologue 7-17
 - Prologue, function 4-1, 10-6
 - Promotion 10-13
 - Ps option 3-25
 - Public
 - variable names 14-1
 - Pure virtual function B-4
 - _pure_virtual_function_called function 5-47
 - putl function 5-48
 - relationship to getl 5-30
 - putw function 5-49
 - relationship to getw 5-31
- ## Q
- Q option 3-36, A-4
 - q option 3-29
 - QA option 3-36
 - Qe option 3-36, A-4
 - Qe option (UNIX/DOS) A-4
 - Qfn option 3-36
 - Qfs option 3-36
 - Qi option 3-36, A-4

- Qme option 3-37
- Qmi option 3-37
- Qms option 3-37, A-4
- Qmw option 3-37
- Qo option 3-37
- Qs option 3-37, A-4
- Qw option 3-37, A-4

R

- Raising an exception
 - definition B-4
- read function 5-50, 13-30
- Reading bytes from a file 5-50
- realloc function 13-11
- Redirect stderr
 - +E option 3-35
- Redundant code elimination 7-2
- Redundant jump optimization 7-14
- Redundant store elimination 7-11
- Reentrant code
 - generating 13-10
- Reentrant functions 5-8, 13-10
- Register
 - reserving a register 13-32
 - reserving for special purposes 13-36
 - storage class 11-26
 - use 10-5
- register
 - keyword 11-26
- Register coloring
 - OR option 3-43
- Register-relative address mode 3-51
- Register-relative addressing 13-23
- Registers
 - I/O device 3-41
 - reserving
 - Kh option 3-48
- Reserving a register
 - Kh option 3-48
- Restricted debugging information
 - Gr option 3-34
- Rethrow B-5
- Returning a typed value 4-7, 13-5

- ROM-based systems 13-41
- Routines, user-modified 13-27
- RTE instruction
 - using for interrupt procedures
 - nKr option 3-49
- RTS instruction
 - using for interrupt procedures
 - Kr option 3-49

S

- S option 3-29
- sbrk function 5-51
- scanf function
 - preprocessor symbols 6-4
- Scheduling, instructions
 - Or option 3-44
- Search paths
 - (see also Environment variables)
 - nonstandard #include files
 - I option 3-20
 - standard #include files
 - J option 3-22
- Section names, specifying 3-55
- set_new_handler function 5-52
- set_terminate function 5-53
- Setting sign of char variables
 - Ku option 3-50
- set_unexpected function 5-54
- Severity of errors A-2
- Shared
 - program 13-33
- Short integer
 - reading from a stream 5-31
 - writing to stream 5-49
- short type 11-3
- Short/long displacement optimizations 7-15
- _simulated_input variable 13-30
- _simulated_output variable 13-30
- sin function 3-19
- Single inheritance B-3
- sinh function 3-19
- Size
 - aggregates 11-15

- data types 11-2
- unpacked structure 11-20
- Size optimization
 - Os option 3-44
- size_t type
 - memcpy function 5-40
 - memset function 5-41
 - zalloc function 5-68
- socket function 5-55
- Source file problems, detecting
 - v option 3-30
- Special Template Instance B-5
- sprintf function 13-10
- sqrt function 3-19
- sscanf function 13-10
- Stack
 - 16-bit quantities
 - Zp2 option 3-51
 - 32-bit quantities
 - Zp4 option 3-51
 - disabling frame sharing
 - Kf option 3-47, 3-48
 - frames 10-5
 - int size
 - Ze option 3-50, 3-51
 - pointer 10-5
 - popping
 - Oc option 3-41
- stack section 3-53, 12-2
- stat function 5-56
- Static
 - variables 11-26
 - names 14-2
- Static data members B-5
- Static member functions B-5
- __STDC__ preprocessor symbol 3-20, 3-63
- stderr
 - directing diagnostic messages
 - Fee option 3-36
 - redirection
 - +E option 3-35
- stdio.h include file 13-10
- stdout
 - directing diagnostic messages
 - Feo option 3-36
- Storage
 - allocation of data types 11-2
 - layout 11-1
- strcpy function 3-19
- Stream
 - reading a long integer 5-30
 - reading a short integer 5-31
 - writing a long integer 5-48
 - writing a short integer 5-49
- Strength reduction 7-2
 - optimization 7-9
- strings section 3-53, 12-2, 12-5
 - naming
 - NS option 3-56
 - specifying address mode
 - as options 3-54
- strlen function 3-19
- strtod function 13-11
- strtol function 13-11
- strtoul function 13-11
- struct type 11-4
- struct/union in parameter area, alignment of 10-4
- Structure alignment 11-20
- Structure layout
 - Zle option 3-51
- Structure members, aligning
 - Z options 3-31
- Structure padding 11-17
- Structure size
 - Zm option 3-51
- Structures 11-7
 - packing 11-23
 - size 11-20
- Suffix, C++ files
 - +z option 3-62
- Summary message
 - suppression
 - Qs option 3-37
- Summary of command line options 3-1

Superscalar instruction scheduling 7-17

Suppressing executable file

-c option 3-25

swab function 5-57

Swapping bytes in memory 5-57

switch statement

generating code for 7-10

Symbol names

convention 3-32

prepend dot

-upd option 3-32

prepend underscore

-upu option 3-32

suppress prefix

-us option 3-32

Syntax checking only

-y option 3-29

SysHost feature 13-29

syshost section 12-2, 12-7

sys_in function 5-58

sys_out function 5-59

sys_stat function 5-60

System data 13-35

System functions 5-9

chdir 5-14

close 5-15

connect 5-16

creat 5-17

relationship to close 5-15

relationship to read 5-50

relationship to write 5-67

_exit 5-26

getcwd 5-29

lseek 5-35

memclr 5-45

open 5-43

relationship to close 5-15

relationship to read 5-50

relationship to write 5-67

_popen 5-46

read 5-50, 13-30

sbrk 5-51

socket 5-55

stat 5-56

sys_in 5-58

sys_out 5-59

sys_stat 5-60

unlink 5-66

write 5-67, 13-30

T

Tagging functions

-Kt option 3-50

tags section 3-50, 3-53, 12-2, 12-7

Tail recursion optimization

-it 3-39

tan function 3-19

tanh function 3-19

Template function B-5

Templates

class B-5

data B-5

declaration B-5

definition B-5

instance B-5

instantiation B-6

special instance B-5

Temporary files

UNIX

alternate locations 2-3

terminate function 5-61

Termination

normal program 5-26

test_lib script 6-7

test_one script 6-7

This 7-13

this pointer B-6

Threads

definition 13-10

Throw B-6

Throw expression B-6

Thrown object B-6

Time optimization

-Ot option 3-44, 3-45

- Time stamp
 - compilation
 - Vt option 3-30
- Title for listing file
 - Flt option 3-23
- toascii function 5-62
- _tolower function 5-63
- tolower function 5-7
- Toolkit components 1-1
- _toupper function 5-64
- toupper function 5-7
- Trailer bytes 11-20
- Truncating identifiers
 - ut option 3-33
- Try block B-6
- Type casting 11-6
- Type conversion 11-4
- Type database 13-49
- typeof operator 3-65
- Types
 - (see also Data, types)
 - returned for functions 4-7, 13-5
- Type-safe linkage B-6
- U**
 - U option 3-24
 - Undefining preprocessor macros
 - U option 3-24
 - Unexpected exception B-6
 - unexpected function 5-65
 - Unhandled exception B-7
 - Uninitialized data section 3-53
 - Uninitialized global variables 3-38
 - Uninitialized static data section
 - naming
 - NZ option 3-57
 - union type 11-4
- UNIX
 - command line 2-1
 - compiler syntax 2-1
 - compiler use 2-1, 3-1
 - environment variables 2-3
 - file locations 2-2
 - filename defaults 2-2
 - invoking the compiler 2-1
 - option form, positive and negative 3-1
 - option, command line
 - descriptions of 3-1
 - UNIX level-1+ elements 5-2
 - unlink function 5-66
 - Unlinking a filename 5-66
 - unpacked keyword 3-65, 4-3
 - unpacked type 11-22
 - Unreachable (dead) code elimination 7-2
 - Unsigned char default
 - Ku option 3-50
 - unsigned int type 11-3
 - Unsigned integer
 - conversion to ASCII string 5-34
 - Unsigned long integer
 - conversion to ASCII string 5-37
 - unsigned types 11-3
 - upd option 3-32
 - Upper-case characters
 - converting to 5-64
 - upu option 3-32
 - us option 3-32
 - User-modified routines
 - for embedded systems 13-27
 - ut option 3-33
 - ut0 option 3-33
- V**
 - v option 3-30, A-21
 - Value comparison
 - returning the greater of two values 5-39
 - returning the lesser of two values 5-42
 - Variable initialization
 - KI option 3-48
 - Variable names, truncating
 - ut option 3-33
 - Variables
 - allocation of variables 11-25
 - global, uninitialized 3-38
 - initializations 13-37
 - local 11-26

- in the function prologue 10-7
- names 10-10
 - external 14-1
 - inside asm 4-8, 13-5
 - public 14-1
 - static 14-2
- saving initialized variables in ROM 13-37
- static 5-8, 11-26
- structure members, aligning
 - Z options 3-31
- vars section 12-2
 - specifying address mode
 - ai options 3-54
- Vb option 3-30
- Vd option 3-30
- Verbose mode, enabling and disabling
 - V options 3-30
- Vi option 3-30
- Virtual
 - function B-7
 - pure B-4
- Virtual table generation
 - +e option 3-58
- Volatile
 - member function B-7
 - object B-7
- Volatile data
 - OV option 3-45
- volatile variables, disabling optimizations
 - Ob option 3-40
- vsprintf function 13-10
- Vt option 3-30
- Vw option 3-30

W

- Wa option 3-6, 3-15, 3-30
- Warning messages A-1
 - suppression
 - Qw option 3-37
- WARNING-xxx (stub routine) called
 - ftell 5-35, 5-44, 5-66

- _WARNING_xxx_stub_used symbol
 - unresolved
 - unlink 5-66
- Weak externals 3-38
 - initialize to zero
 - X0 option 3-38
 - no value
 - Xp option 3-38
 - uninitialized globals
 - Xc option 3-38
- Windows
 - command line 2-1
 - compiler syntax 2-1
 - compiler use 2-1, 3-1
 - environment variables 2-3
 - file locations 2-2
 - filename defaults 2-2
 - invoking the compiler 2-1
 - option form, positive and negative 3-1
- Wl option 3-31
- Wrapper, C++ 10-16
- write function 5-67, 13-30
- Writing a short integer to a stream 5-49
- Writing bytes to a file 5-67

X

- x option 3-19
- X0 option 3-38
- Xc option 3-38
- Xp option 3-38
- XRAY Debugger
 - description 1-3
 - fully qualified pathnames
 - Gf option 3-34
 - generating debugging information
 - g option 3-35
 - line number information
 - Gl option 3-34
 - multiple statements on line
 - Gm option 3-34
 - preprocessor macros
 - Gd option 3-33

restricted information

-Gr option 3-34

XRAY Source Explorer

-GS option 3-35

-Gs option 3-34

Y

-y option 3-29

Z

+z option 3-62

-Za2 option 3-31

-Za4 option 3-31

zalloc function 5-68, 13-11

-ZBA option 3-20

-ZBnA option 3-20

-zc option 3-60

-Ze option 3-50, 3-51

-ze option 3-60

Zero initialization

local variables

-KI option 3-48

zerovars section 3-53, 11-26, 12-2, 12-6

naming

-NZ option 3-57

specifying address mode

-az options 3-54

-Zi option 13-8

-Zle option 3-51

-Zm option 3-51

-Zn option 3-51

-Zp2 option 3-51, 11-6

-Zp4 option 3-51, 11-6