C User's Guide

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5777-10
Revision A, December 1996

Copyright 1996 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system, licensed from Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. UNIX is a registered trademark in the United States and other countries and is exclusively licensed by X/Open Company Ltd. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

Sun, Sun Microsystems, the Sun logo, SunSoft, Solaris, OpenWindows, and Sun WorkShop are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Intel is a registered trademark of Intel Corporation. PowerPC is a trademark of International Business Machines Corporation.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Please
Recycle

Contents

Preface. xxi

1. Introduction to the C Compiler 1

Operating Environments. 1

Standards Conformance 1

Organization of the Compiler. 1

C-Related Programming Tools 3

2. `cc` Compiler Options 5

Option Syntax. 5

The `cc` Options. 6

`-#`. 6

`-###` 6

`-A`*name*`[(`*tokens*`)]` 7

`-B[static|dynamic]` 7

`-C`. 7

`-c`. 7

-D*name*[=*tokens*] 7
-d[y|n] 8
-dalign 9
-E 9
-erroff=*t* 9
-errtags=*a* 10
-fast 10
-fd 11
-flags 11
-fnonstd 11
-fns 12
-fprecision=<*p*> 12
-fround=*r* 12
-fsimple[=*n*] 12
-fsingle 13
-fstore 14
-ftrap=*t* 14
-G 14
-g 15
-H 15
-h *name* 15
-I*dir* 16
-i 16
-keeptmp 16

`-KPIC` 16

`-Kpic` 16

`-L`*dir* 17

`-l`*name* 18

`-mc` 18

`-misalign` 18

`-misalign2` 18

`-mr` 19

`-mr,`*string* 19

`-mt` 19

`-native` 19

`-nofstore` 19

`-noqueue` 20

`-O` 20

`-o` *filename* 20

`-P` 20

`-p` 20

`-Q[y|n]` 20

`-qp` 21

`-R`*dir*`[:`*dir*`]` 21

`-S` 21

`-s` 21

`-U`*name* 21

`-V` 21

-v 22

-W*c*, *arg* 22

-w 23

-X[a|c|s|t] 23

-x386 23

-x486 24

-xa 24

-xarch=*a* 24

-xautopar 27

-xcache=*c* 28

-xCC 29

-xcg[89|92] 29

-xchip=*c* 29

-xcrossfile 30

-xdepend 31

-xe 31

-xexplicitpar 31

-xF 32

-xhelp=*f* 32

-xildoff 33

-xildon 33

-xinline=[*f1*,...,*fn*] 33

-xlibmieee 34

-xlibmil 34

-xlic_lib=*l* 34

-Xlic_lib=sunperf 34

-xlicinfo.. 34

-xloopinfo 34

-xM... 35

-xM1 36

-xMerge 36

-xnolib... 36

-xnolibmil 37

-xO[1|2|3|4|5]................................ 37

-xP.. 39

-xparallel 39

-xpentium.. 40

-xpg 40

-xprofile=*p* 40

-xreduction 42

-xregs=*r* 42

-xrestrict=*f*...................................... 43

-xs.. 44

-xsafe=mem 44

-xsb 44

-xsbfast... 44

-xsfpconst 45

-xspace 45

`-xstrconst` 45

`-xtarget=`*t* 45

`-xtemp=`*dir* 51

`-xtime` 51

`-xtransition` 51

`-xunroll=`*n* 52

`-xvpara` 52

`-Y`*c*`,` *dir* 52

`-YA,` *dir* 53

`-YI,` *dir* 53

`-YP,` *dir* 53

`-YS,` *dir* 53

`-Zll` 53

`-Zlp` 53

`-Ztha` 54

Options Passed to the Linker 54

Localization of Error Messages 54

3. Sun ANSI C Compiler-Specific Information 57

Environment Variables 57

`TMPDIR` 57

`SUNPRO_SB_INIT_FILE_NAME` 57

`PARALLEL` 58

Global Behavior: Value versus `unsigned` Preserving 58

Keywords 58

`asm` Keyword 58
`_Restrict` Keyword 59
`long long` Data Type 61
Printing `long long` Data Types 62
Usual Arithmetic Conversions 62
Constants 63
Integral Constants 63
Character Constants 64
Include Files 64
Nonstandard Floating Point 65
Preprocessing Directives 66
Assertions 66
Pragmas. 68
Predefined Names 71
MP C *(SPARC)* 72
Overview 73
Explicit Parallelization and Pragmas 74
Compiler Options 82
4. `cscope`: Interactively Examining a C Program 83
The `cscope` Process 83
Basic Use 84
Step 1: Set Up the Environment 84
Step 2: Invoke the `cscope` Program 85

Step 3: Locate the Code 86

Step 4: Edit the Code 92

Command-Line Options 93

View Paths 95

`cscope` and Editor Call Stacks 96

Examples. 97

Command-Line Syntax for Editors 101

Unknown Terminal Type Error 102

SourceBrowser 102

5. `lint` Source Code Checker. 105

Overview of the `lint` Program 105

Basic and Enhanced `lint` Functionality. 106

Using `lint` 107

The `lint` Options 108

`-#`. 108

`-###` 109

`-a`. 109

`-b`. 109

`-C` ***filename***. 109

`-c`. 109

`-dirout=`***dir*** 109

`-err=warn`. 109

`-errchk=`l 109

`-errfmt=`***f*** 111

-errhdr=*h* 111

-erroff=*t* 112

-errtags=*a* 113

-F 113

-fd 113

-flagsrc=*file* 113

-h 113

-I*dir* 113

-k 114

-L*dir* 114

-l*x* 114

-m 114

-Ncheck=*c* 114

-Nlevel=*n* 115

-n 116

-o*x* 116

-p 116

-R*file* 116

-s 116

-u 117

-V 117

-v 117

-W*file* 117

-x 117

`-XCC=`***a*** 117
`-Xexplicitpar=`***a*** 117
`-Xkeeptmp=`***a*** 118
`-Xtemp=`***dir*** 118
`-Xtime=`***a*** 118
`-Xtransition=`***a*** 118
`-y` 118
`lint` Messages 119
Options to Suppress Messages 119
`lint` Message Formats 120
`lint` Directives 123
Predefined Values 123
Directives 124
`lint` Reference and Examples 128
Checks Performed by `lint` 128
`lint` Libraries 133
`lint` Filters 134
A. ANSI C Data Representations 137
Storage Allocation 137
Data Representations 138
Integer Representations 138
Floating-Point Representations 140
Exceptional Values 141
Hexadecimal Representation of Selected Numbers 143

Pointer Representation 143
Array Storage 144
Arithmetic Operations on Exceptional Values 144
Argument-Passing Mechanism 146
B. Implementation-Defined Behavior 151
Translation 151
Environment 152
Identifiers 152
Characters 153
Integers 154
Floating-Point 156
Arrays and Pointers 157
Registers 157
Structures, Unions, Enumerations, and Bit-Fields 157
Qualifiers 159
Declarators 159
Statements 159
Preprocessing Directives 159
Library Functions 164
Signals 167
Streams and Files 169
`errno` 171
Memory 176
`abort` Function 176

`exit` Function 176

`getenv` Function 176

`system` Function 177

`strerror` Function. 177

Locale Behavior 177

C. `-Xs` Differences for Sun C and ANSI C 179

D. Performance Tuning *(SPARC)*. 181

Limits. 181

`libfast.a` Library. 182

Index 183

Figures

Figure 1-1 Organization of the C Compilation System. 2

Tables

Table P-1 Summary of C Compiler Documentation and Its Location. . . xxii
Table P-2 C Man Pages and their Usage xxiv
Table P-3 C-Related Man Pages xxv
Table P-4 Typographic Conventions in This Manual xxix
Table P-5 Typographic Notations for Arguments xxix
Table P-6 Shell Prompts xxx
Table 1-1 Components of the C Compilation System 2
Table 2-1 The `-erroff` Values 9
Table 2-2 `-fast` selections across platforms 10
Table 2-3 The `-xarch` Values 25
Table 2-4 The `-xcache` Values 28
Table 2-5 The `-xchip` Values. 30
Table 2-6 The `-xregs` Values. 42
Table 2-7 The `-xtarget` Values 46
Table 2-8 The `-xtarget` Expansions 47
Table 3-1 Data Type Suffixes. 63

Table 3-2 Multiple-character Constant (ANSI) 64

Table 3-3 Multiple-character Constant (non-ANSI) 64

Table 3-4 Predefined Identifier 71

Table 4-1 `cscope` Menu Manipulation Commands 86

Table 4-2 Commands for Use After an Initial Search 88

Table 4-3 Commands for Selecting Lines to be Changed 98

Table 5-1 The `–errfmt` Values 111

Table 5-2 The `–errhdr` Values 111

Table 5-3 The `–erroff` Values 112

Table 5-4 The `–Ncheck` Values 114

Table 5-5 `lint` Options and Messages Suppressed 120

Table 5-6 `lint` Directives 125

Table A-1 Storage Allocation for Data Types 137

Table A-2 Representation of `short` 138

Table A-3 Representation of `int` and `long` 139

Table A-4 Representation of `long long` 139

Table A-5 `float` Representation 140

Table A-6 `double` Representation 140

Table A-7 `long double` Representation *(SPARC) (PowerPC)* 141

Table A-8 `long double` Representation *(Intel)* 141

Table A-9 `float` Representations 141

Table A-10 `double` Representations 142

Table A-11 `long double` Representations 142

Table A-12 Hexadecimal Representation of Selected Numbers *(SPARC) (PowerPC)* 143

Table A-13 Hexadecimal Representation of Selected Numbers *(Intel)* . . . 143

Table A-14 Automatic Array Types and Storage 144
Table A-15 Abbreviation Usage 145
Table A-16 Addition and Subtraction Results 145
Table A-17 Multiplication Results 145
Table A-18 Division Results 146
Table A-19 Comparison Results 146
Table B-1 Representations and Sets of Values of Integers 154
Table B-2 Values of Floating-Point Numbers 156
Table B-3 Padding and Alignment of Structure Members 158
Table B-4 Character Sets Tested by `isalpha`, `islower`, Etc. 165
Table B-5 Values Returned on Domain Errors 166
Table B-6 Semantics for `signal` Signals 167
Table B-7 Error Messages Generated by `perror` 171
Table B-8 Names of Months 178
Table B-9 Days and Abbreviated Days of the Week 178
Table C-1 `-Xs` Behavior 179

Preface

This manual, the *C User's Guide* describes the C 4.2 compiler. The standard language is referred to as *ANSI C*. The notation *K&R C* refers to Kernighan and Ritchie C, which is non-ANSI (or pre-ANSI) C.

Audience

This document is intended to assist software developers write programs in the C language. This book does not discuss basic concepts of C programming.

Document Organization

This book contains the following chapters:

Chapter 1, "Introduction to the C Compiler," provides an overview to C.

Chapter 2, "`cc` Compiler Options," describes the compiler optionsfor the Solaris™ operating system.

Chapter 3, "Sun ANSI C Compiler-Specific Information," describes areas specific to the Sun ANSI C compiler.

Chapter 4, "`Cscope`: Interactively Examine a C Program," describes `cscope`, a C programming tool.

Chapter 5, `Lint` Source Code Checker," describes how to use `lint`, another C programming tool.

This book also includes these appendices:

- Appendix A, "ANSI C Data Representations"
- Appendix B, "Implementation-Defined Behavior"
- Appendix C, "-Xs Differences for Sun C and ANSI C"
- Appendix D, "Performance Tuning (SPARC)"

This book concludes with an index.

C Compiler Documentation

Table P-1 summarizes the C compiler documentation provided with this release, and identifies where that documentation is located.

Table P-1 Summary of C Compiler Documentation and Its Location

Document	Online Books	Online ASCII	Online PostScript	Hard Copy
C 4.2 README File		/opt/SUNWspro /READMEs/c		
C 4.2 Quick Reference				X
C User's Guide	X			X
Error and Warning Messages file		/opt/SUNWspro /READMEs /c_lint_errors		
Incremental Link Editor (ild)	X			X
Installation and Licensing Guide	X			X
Making the Transition to ANSI C	X			
Man pages		/opt/SUNWspro /man		
"MP C" white paper			/opt/SUNWspro /READMEs /mpc.ps	
Numerical Computation Guide	X			X
Performance Profiling Tools	X			X
Quick Install for Solaris				X

Online Documentation

On-line documents are in the following formats:

- Online books
- Man pages (ASCII, formatted with `nroff`)
- Error and Warning Messages file (ASCII)
- C 4.x `README` file (ASCII)
- PostScript files

Online Books

Some of the C compiler documents are provided in online documentation viewing tools. These online documents provide the following benefits:

- Take advantage of dynamically linked headings and cross-references.
- Search for topics by using a word or phrase.

See your platform-specific installation guide and `README` file for more information on these tools.

The following C compiler documentation is provided in online books:

- *C User's Guide*—Describes the features of the C 4.x compiler.
- *Making the Transition to ANSI C*—Shows how to port your C code from previous versions of C to ANSI C.

Manual Pages

Each manual page (man page), discusses one subject, such as a user command or library function. The C man pages are in `/opt/SUNWspro/man`.

To view a man page using the `man` command:

1. Insert the name of the directory in which you installed the C compiler at the beginning of your search path.
2. Add a line to the `.cshrc` file with `setenv MANPATH=` at the start or, to the `.profile` file with `export MANPATH=` at the start.
3. Now, view a man page by executing this command:

```
% man command_name
```

where *command_name* is `cc`, for example.

Table P-2 describes the C man pages, and identifies the best uses for each man page.

Table P-2 C Man Pages and their Usage

Title	Description	Usage
cb	Formats your source code.	Makes the program more readable.
cc	Describes the C compilation system.	Provides C system information.
cflow	Produces a flow graph of the external references in C, lex, yacc, and assembly-language files.	Checks program dependencies.
cscope	Interactively examines a C program.	Searches and edits source files.
ctrace	Traces C program execution, statement by statement.	Checks program execution statements.
cxref	Generates a C program cross-reference table.	Checks program dependencies and structure.
c89	Enables compliance with XPG4.	Verifies XPG4 comopliance.
indent	Indents and formats a C program source file.	
lint	The C program checker.	Checks for code constructs that may cause the program to not compile or execute with unexpected results. Also checks program portability and cross-file consistency.

Table P-3 Identifies the man pages containing C compiler related information.

Table P-3 C-Related Man Pages

Title	Description
gprof	Includes the prof functionality, and produces a callgraph profile displaying a list of modules that call, or are called by, other modules.
ild	Contains information on the incremental ild linker.
inline	Contains information on inlining code.
m4	Preprocesses C and assembly language programs.
make	Maintains, updates, and regenerates related programs and files.
ld	Contains information on the ld linker.
lex	Describes the lexical analysis program generator.
prof	Reports time, percentage of time spent executing a program and number of calls to functions.
sccs	Provides a front end for the source code control system.
tcov	Provides a line-by-line frequency profiler that produces a copy of the source file, annotated to show which lines are used, and how often.
yacc	Parses tokens passed by a lexical analyzer.

The man man page describes the options available with the man command to browse man pages. Other tools, such as tkman, provide search features and hypertext links to the man pages listed in the "SEE ALSO" section of a man page.

To print man pages with the lp command, type:

```
% man command_name | lp
```

where *command_name* is cc, for example.

Error and Warning Messages File

The Error and Warning Messages file, located in /opt/SUNWspro/READMEs/c_lint_errors, contains C compiler error and warning messages, and the lint program messages. Many of the messages are self-explanatory. To obtain a description of these messages and code examples, search the text file for a string from the generated message, or obtain its unique tag and search on that (cc -errtags=yes).

There are several methods to locate an error message and its description in the file:

- Load the ASCII file into an editor and use the editor commands to search the file.
- Use the `grep` command in a shell to search the file.
- Use the command `cc -xhelp=errors` to view the file.

For example, if this message is received during a compilation:

`file:` ***filename*** `line:` ***n*** `empty constant expression after macro expansion`

Obtain further explanation of this message by:

1. Loading the `/opt/SUNWspro/READMEs/c_lint_errors` file into an editor.

2. Searching for a string from the message, for example, “empty constant.”

A more detailed description is displayed with the sample code generating the error message, and the message ID, or tag:

```
empty constant expression after macro expansion
A #if or #elif directive contains an expression that, after macro
expansion, consists of no tokens.
#define EMPTY
#if EMPTY
     char *mesg = “EMPTY is non-empty”;
#end if
MESSAGE ID: E_EMPTY_CONST_EXP_AFTER_EXPAND
-----------------------------------------
```

To print the Error and Warning Messages file, type:

```
% lp /opt/SUNWspro/READMEs/c_lint_errors
```

When an error occurs, the error message is preceded by a file name and line number. The line number is the line where a problem is diagnosed. Occasionally, the compiler must read the next token before it can diagnose a problem, in which case the line number in the message may be a higher line number than that of the offending line.

Note – The compiler displays many of the messages contained in this file only when used with the `cc -v` option. With this option, the compiler performs stricter semantics checking and, therefore, displays more diagnostic messages.

C 4.2 `README` *file*

The C 4.2 `README` file, located in `/opt/SUNWspro/READMEs/c`, contains important information about the compiler, such as:

- New features in this release
- Changes to features
- Software incompatibilities
- Current software bugs
- Documentation errata

You can display this file online in your editor, print it using the `lp` command, or view it using the command `cc -xhelp=readme`.

PostScript Files

The PostScript files include "white papers" such as "MP C" white paper, `MPC.ps`. These files are located in `/opt/SUNWspro/READMEs`.

To view a PostScript file online, type:

```
% imagetool filename &
```

To print a PostScript file, type:

```
% lp filename
```

Hard-Copy Documentation

The following C documents are available in hard copy:

- *C User's Guide*, describes the features accompanying the C 4.2 compiler.
- *C 4.2 Quick Reference*, summarizes the command-line options.
- The platform-specific installation guides, containing instructions for installing the C compiler and other software on Solaris; as well as information on licensing

Related Documentation

The following documents contain useful information on programming and compiling.

- *Performance Profiling Tools*—Provides information on various profiling tools, such as `prof`. Available in online books.
- *Numerical Computation Guide*—Describes floating-point software and hardware. Available in online books.
- *Linker and Libraries Guide*—Provides information on the linker, `ld`, and on linking libraries. See also the `ld(1)` man page. Part of the Solaris programming documentation.
- *Incremental Link Editor (ild)*—Provides information on the incremental linker, `ild`, which replaces the standard linker for incremental linking.

The following books are useful for information on the C language:

- *The C Language*, second edition, by Kernighan and Ritchie (Prentice-Hall, 1988)
- *C: A Reference Manual*, third edition, by Harbison and Steele (Prentice-Hall, 1991)
- *The Standard C Library*, by P. J. Plauger (Prentice-Hall, 1992)

For implementation-specific details not covered in this book, refer to the *Application Binary Interface* for your machine.

Typographic Conventions

Table P-4 describes the typographic conventions used in this book.

Table P-4 Typographic Conventions in This Manual

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Table P-5 describes the typographic notations used for arguments to the compiler and `lint` tool options.

Table P-5 Typographic Notations for Arguments

Notation	Meaning	Example
`[ ]`	Square brackets contain arguments that can be optional or required.	`-d[y|n]`
`|`	The "pipe" or "bar" symbol separates arguments, only *one* of which may be used at one time.	`-d[y|n]`
`,`	The comma separates arguments, *one or more* of which may be used at one time.	`-xinline=[`*f1,...fn*`]`
`:`	The colon, like the comma, is sometimes used to separate arguments.	`-R`*dir*`[:`*dir*`]`
`...`	The ellipsis indicates omission in a series.	`-xinline=[`*f1,...fn*`]`
`%`	The percent sign indicates the word following it has a special meaning.	`-ftrap=%all`
`` ` ``	The back quote indicates a command that is being executed.	`` `uname -s` ``

Other Documentation Conventions

Operating Environments and Platform-Specific References

The C 4.2 documentation supports the following operating environments:

- The Solaris™ 2.x operating system on the following architectures:
  - SPARC™ architectures
  - Intel architectures, where Intel refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent
  - The RISC PowerPC architecture compliant with the Common Hardware Reference Platform (CHRP) and the PowerPC Reference Platform (PReP) specifications

The C 4.2 compiler documentation supports all the above operating systems and platforms, unless otherwise specified. Anything unique to one or more platforms is identified as "*(SPARC)*," "*(Intel)*," and/or "*(PowerPC)*."

Path Names

In the C compiler documentation, the pathname to many file locations is given as `/opt/SUNWspro`. This is the default installation location. If you installed the compiler in a different directory, substitute that directory name instead.

Shell Prompts in Command Examples

Table P-6 shows the system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell, used in the command examples.

Table P-6 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Introduction to the C Compiler

This chapter provides information on the C compiler, including operating environments, standards conformance, organization of the compiler, and C-related programming tools.

Operating Environments

For an explanation of the specific operating environments supported in this release, refer to the README file.

Standards Conformance

The compiler conforms to the American National Standard for Programming Language - C, ANSI/ISO 9899-1990. It also conforms to FIPS 160. Because the compiler also supports traditional K&R C (Kernighan and Ritchie, or pre-ANSI C), it can ease your migration to ANSI C.

Organization of the Compiler

The C compilation system consists of a compiler, an assembler, and a link editor. The cc command invokes each of these components automatically unless you use command-line options to specify otherwise.

Chapter 2, "cc Compiler Options," discusses all the options available with cc.

Figure 1-1 shows the organization of the C compilation system.

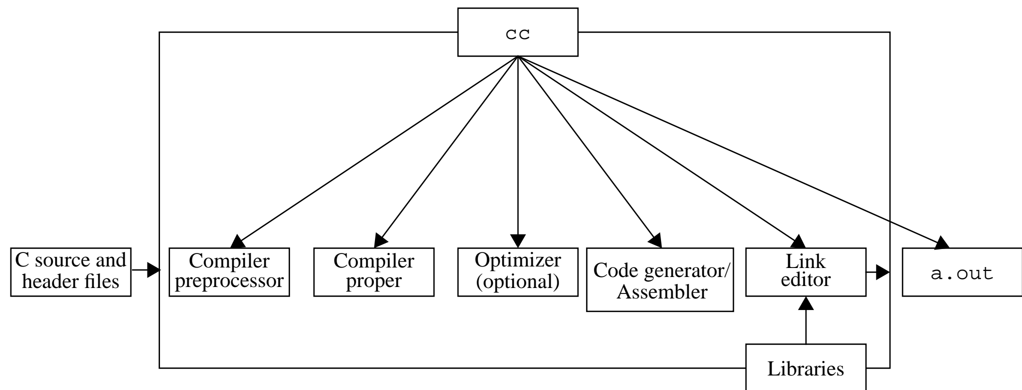

Figure 1-1 Organization of the C Compilation System

Table 1-1 summarizes the components of the compilation system.

Table 1-1 Components of the C Compilation System

Component	Description	Notes on Use
`cpp`	Preprocessor	`-Xs`
`acomp`	Compiler (preprocessor built in for non-`Xs` modes)	
`iropt`	Code optimizer	*(SPARC)* `-O`, `-xO [2-5]`, `-fast`
`cg386`	Intermediate language translator	*(Intel)* Always invoked
`cgppc`	Intermediate language translator	*(PowerPC)* Always invoked
`inline`	Inline expansion of assembly language templates	`.il` file specified
`mwinline`	Automatic inline expansion of functions	*(Intel) (PowerPC)* `-xO4`, `-xinline`
`fbe`	Assembler	

Table 1-1 Components of the C Compilation System *(Continued)*

Component	Description	Notes on Use
`cg`	Code generator, inliner, assembler	*(SPARC)*
`codegen`	Code generator	*(Intel)* *(PowerPC)*
`ld`	Linker	
`ild`	Incremental linker	*(SPARC)* `-g`, `-xildon`

The C compiler optimizer removes redundancies, optimally allocates registers, schedules instructions, and reorganizes code. Select from multiple levels of optimization to obtain the best balance between application speed and use of memory.

C-Related Programming Tools

There are a number of tools available to aid in developing, maintaining, and improving your C programs. The two most closely tied to C, `cscope` and `lint`, are described in this book. Others are described in the Solaris reference or programming documentation and/or *Performance Profiling Tools*. In addition, a man page exists for each of these tools. Refer to the preface of this book for a list of all the associated man pages.

`cc` *Compiler Options* 2

This chapter describes the C compiler options. It includes sections on option syntax, the `cc` options, and options passed to the linker.

If you are porting a K&R C program to ANSI C, make special note of the section on compatibility flags, “-X[a|c|s|t]” on page 23. Using them makes the migration to ANSI C easier. Also, Refer to the book titled *Making the Transition to ANSI C*.

Option Syntax

The syntax of the `cc` command is:

% **cc** [*options*] *filenames* [*libraries*]. . .

where:

- *options* represents one or more of the options described in “The cc Options” on page 6
- *filenames* represents one or more files used in building the executable program

  `cc` accepts a list of C source files and object files contained in the list of files specified by *filenames*. The resulting executable code is placed in `a.out`, unless the `-o` option is used. In this case, the code is placed in the file named by the `-o` option.

Use `cc` to compile and link any combination of the following:

- C source files, with a `.c` suffix
- C preprocessed source files, with a `.i` suffix
- Object-code files, with `.o` suffixes
- Assembler source files, with `.s` suffixes

After linking, `cc` places the linked files, now in executable code, into a file named `a.out`, or into the file specified by the `-o` option.

- *libraries* represents any of a number of standard or user-provided libraries containing functions, macros, and definitions of constants.

See option `-Y`*P, dir* to change the default directories used for finding libraries. *dir* is a colon-separated path list. The default library search order for `cc` is:

```
/opt/SUNWspro/SC4.2/lib

/usr/ccs/lib

/usr/lib
```

`cc` uses `getopt` to parse command-line options. Options are treated as a single letter or a single letter followed by an argument. See `getopt`(3c).

The `cc` Options

This section describes the `cc` options, arranged alphabetically. These descriptions are also available in the man page, `cc`(1). See the *C 4.x Quick Reference* or the `-flags` option for a one-line summary of these descriptions.

Options noted as being unique to one or more platforms are accepted without error and ignored on all other platforms. For an explanation of the typographic notations used with the options and arguments, refer to Table P-5 on page xxix.

-#

Turns on verbose mode, showing each component as it is invoked.

-###

Shows each component as it would be invoked, but does not actually execute it.

-A*name*[(*tokens*)]

Associates *name* as a predicate with the specified *tokens* as if by a `#assert` preprocessing directive.Preassertions:

- `system(unix)`
- `machine(sparc)` *(SPARC)*
- `machine(i386)` *(Intel)*
- `cpu(sparc)` *(SPARC)*
- `cpu(i386)` *(Intel)*
- `cpu(ppc)` *(PowerPC)*

These preassertions are not valid in `-Xc` mode.

-B[static|dynamic]

Specifies whether bindings of libraries for linking are `static` or `dynamic`, indicating whether libraries are non-shared or shared, respectively.

`-Bdynamic` causes the link editor to look for files named `lib`*x*`.so` and then for files named `lib`*x*`.a` when given the `-l`*x* option.

`-Bstatic` causes the link editor to look only for files named `lib`*x*`.a`. This option may be specified multiple times on the command line as a toggle. This option and its argument are passed to `ld`.

-C

Prevents the C preprocessor from removing comments, except those on the preprocessing directive lines.

-c

Directs `cc` to suppress linking with `ld`(1) and to produce a `.o` file for each source file. You can explicitly name a single object file using the `-o` option.

-D*name*[=*tokens*]

Associates *name* with the specified *tokens* as if by a `#define` preprocessing directive. If no `=`*tokens* is specified, the token `1` is supplied.

Predefinitions (not valid in `-Xc` mode):

- `sun`
- `unix`
- `sparc` *(SPARC)*
- `i386` *(Intel)*

The following predefinitions are valid in all modes.

`_ _sun`

`_ _unix`

`_ _SUNPRO_C=0x400`

`` _ _`uname -s`_`uname -r` `` (example: `_ _SunOS_5_4`)

`_ _sparc` *(SPARC)*

`_ _i386` *(Intel)*

`_ _BUILTIN_VA_ARG_INCR`

`_ _SVR4`

`_ _LITTLE_ENDIAN` *(PowerPC)*

`_ _ppc` *(PowerPC)*

The following is predefined in `-Xa` and `-Xt` modes only:

`_ _RESTRICT`

The compiler also predefines the object-like macro

`_ _PRAGMA_REDEFINE_EXTNAME`,

to indicate the pragma will be recognized.

-d[y|n]

`-dy` specifies dynamic linking, which is the default, in the link editor.

`-dn` specifies static linking in the link editor.

This option and its arguments are passed to `ld`(1).

-dalign

Allows compiler to generate double-word load/store instructions wherever profitable for improved performance. Assumes that all `double` and `long long` type data are double-word aligned. Do not use this option when correct alignment is not assured.

-E

Runs the source file through the preprocessor only and sends the output to `stdout`.[1] Includes the preprocessor line numbering information. See also the `-P` option.

-erroff=*t*

Suppresses `cc` warning messages. Has no effect on error messages.

t is a comma-separated list that consists of one or more of the following: *tag*, `no%`*tag*, `%all`, `%none`. Order is important; for example, `%all,no%`*tag* suppresses all warning messages except *tag*.

Table 2-1 The `-erroff` Values

Value	Meaning
tag	Suppresses the warning message specified by this *tag*. You can display the tag for a message by using the `-errtags=yes` option.
`no%`*tag*	Enables the warning message specified by this *tag*
`%all`	Suppresses all warning messages
`%none`	Enables all warning messages (default)

The default is `-erroff=%none`. Specifying `-erroff` is equivalent to specifying `-erroff=%all`.

1. The preprocessor is built directly into the compiler, except in `-Xs` mode, where `/usr/ccs/lib/cpp` is invoked.

-errtags=*a*

Displays the message tag for each error message.

a can be either `yes` or `no`. The default is `-errtags=no`. Specifying `-errtags` is equivalent to specifying `-errtags=yes`.

-fast

Selects the optimum combination of compilation options for speed. This should provide close to the maximum performance for most realistic applications. Modules compiled with `-fast` must also be linked with `-fast`.

The `-fast` option is unsuitable for programs intended to run on a different target than the compilation machine. In such cases, follow `-fast` with the appropriate xtarget option. For example:

```
cc -fast -xtarget=ultra ...
```

For C modules that depend on exception handling specified by SVID, follow `-fast` by `-nolibmil`:

```
% cc -fast -nolibmil
```

With `-xlibmil`, exceptions may not be noted by setting `errno` or calling `matherr`(3m).

The `-fast` option is unsuitable for programs that require strict conformance to the IEEE 754 Standard.

The set of options selected by `-fast` differ across platforms:

Table 2-2 `-fast` selections across platforms

Option	SPARC	Intel	PowerPC
-dalign	X	—	—
-fns	X	X	X
-fsimple=1	X	—	X
-ftrap=%none	X	X	X
-libmil	X	X	X
-native	X	X	X

Table 2-2 `-fast` selections across platforms

`-nofstore`	—	X	—
`-xO4`	X	X	X
`-fsingle`	X	X	X

`-fast` acts like a macro expansion on the command line. Therefore, the optimization level and code generation option aspects can be overridden by following `-fast` with the desired optimization level or code generation option. As far as optimization level is concerned, compiling with the `-fast -xO4` pair is like compiling with the `-xO2 -xO4` pair. The latter specification takes precedence.

In previous releases, the `-fast` macro option included `-fnonstd`; now it includes `-fns` instead.

You can usually improve performance for most programs with this option.

-fd

Reports K&R-style function definitions and declarations.

-flags

Prints a summary of each compiler option.

-fnonstd

Causes nonstandard initialization of floating-point arithmetic hardware. In addition, the `-fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into `SIGFPE` signals; if the program has no `SIGFPE` handler, it terminates with a memory dump.

By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual. (See “Nonstandard Floating Point” on page 65 for a further explanation.)

(SPARC) Synonym for `-fns -ftrap=common`.

-fns

(SPARC) Turns on the SPARC nonstandard floating-point mode.

The default is the SPARC standard floating-point mode.

If you compile one routine with `-fns`, then compile all routines of the program with the `-fns` option; otherwise, you can get unexpected results.

-fprecision=<*p*>

(*Intel*) -fprecision={single, double , extended} Initializes the rounding precision mode bits in the Floating-point Control Word to single (24 bits), double (53 bits), or extended (64 bits), respectively. The default floating-point rounding-precision mode is extended.

Note that on Intel, only the precision, not exponent range is affected by the setting of floating-point rounding precision mode.

-fround=*r*

Sets the IEEE 754 rounding mode that is established at runtime during the program initialization.

r must be one of: `nearest`, `tozero`, `negative`, `positive`.

The default is `-fround=nearest`.

The meanings are the same as those for the `ieee_flags` subroutine.

If you compile one routine with `-fround=`*r*, compile all routines of the program with the same `-fround=`*r* option; otherwise, you can get unexpected results.

-fsimple[=*n*]

Allows the optimizer to make simplifying assumptions concerning floating-point arithmetic.

If *n* is present, it must be 0, 1, or 2. The defaults are:

- With no `-fsimple`[=*n*], the compiler uses `-fsimple=0`
- With only `-fsimple`, no =*n*, the compiler uses `-fsimple=1`

`-fsimple=0`

Permits no simplifying assumptions. Preserve strict IEEE 754 conformance.

`-fsimple=1`

Allows conservative simplifications. The resulting code does not strictly conform to IEEE 754, but numeric results of most programs are unchanged.

With `-fsimple=1`, the optimizer can assume the following:

- IEEE 754 default rounding/trapping modes do not change after process initialization.
- Computations producing no visible result other than potential floating point exceptions may be deleted.
- Computations with Infinity or NaNs as operands need not propagate NaNs to their results; e.g., `x*0` may be replaced by `0`.
- Computations do not depend on sign of zero.

With `-fsimple=1`, the optimizer is *not* allowed to optimize completely without regard to roundoff or exceptions. In particular, a floating-point computation cannot be replaced by one that produces different results with rounding modes held constant at runtime. `-fast` implies `-fsimple=1`.

`-fsimple=2`

Permits aggressive floating point optimizations that may cause many programs to produce different numeric results due to changes in rounding. For example, permit the optimizer to replace all computations of `x/y` in a given loop with `x*z`, where `x/y` is guaranteed to be evaluated at least once in the loop, `z=1/y`, and the values of `y` and `z` are known to have constant values during execution of the loop.

-fsingle

(`-Xt` and `-Xs` modes only) Causes the compiler to evaluate `float` expressions as single precision rather than double precision. This option has no effect if the compiler is used in either `-Xa` or `-Xc` modes, as `float` expressions are already evaluated as single precision.

-fstore

(Intel) Causes the compiler to convert the value of a floating-point expression or function to the type on the left-hand side of an assignment, when that expression or function is assigned to a variable, or when the expression is cast to a shorter floating-point type, rather than leaving the value in a register. Due to roundoffs and truncation, the results may be different from those that are generated from the register value. This is the default mode.

To turn off this option, use the `-nofstore` option.

-ftrap=*t*

Sets the IEEE 754 trapping mode.

t is a comma-separated list that consists of one or more of the following: `%all`, `%none`, `common`, `[no%]invalid`, `[no%]overflow`, `[no%]underflow`, `[no%]division`, `[no%]inexact`.

The default is `-ftrap=%none`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left-to-right. The `common` exceptions, by definition, are invalid, division by zero, and overflow.

Example: `-ftrap=%all,no%inexact` means set all traps, except `inexact`.

The meanings are the same as for the `ieee_flags` subroutine, except that:

- `%all` turns on all the trapping modes.
- `%none`, the default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you can get unexpected results.

-G

Passes the option to the link editor to produce a shared object rather than a dynamically linked executable. This option is passed to `ld`(1), and cannot be used with the `-dn` option.

-g

Produces additional symbol table information for the debugger.

This option also uses the incremental linker; see `-xildon` and `-xildoff`.

When used with the `-O` option, a limited amount of debugging is available. The combination, `-xO4 -g`, turns off the inlining that you usually get with `-xO4`.

-H

Prints to standard output, one per line, the path name of each file included during the current compilation. The display is indented so as to show which files are included by other files.

Here, the program `sample.c` includes the files, `stdio.h` and `math.h`; `math.h` includes the file, `floatingpoint.h`, which itself includes functions that use `sys/ieeefp.h`:

```
% cc -H sample.c
    /usr/include/stdio.h
    /usr/include/math.h
         /usr/include/floatingpoint.h
              /usr/include/sys/ieeefp.h
```

-h *name*

Assigns a name to a shared dynamic library as a way to have different versions of a library. In general, the *name* after `-h` should be the same as the file name given after the `-o` option. The space between `-h` and *name* is optional.

The linker assigns the specified *name* to the library and records the name in the library file as the *intrinsic* name of the library. If there is no `-h`*name* option, then no intrinsic name is recorded in the library file.

When the runtime linker loads the library into an executable file, it copies the intrinsic name from the library file into the executable, into a list of needed shared library files. Every executable has such a list. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead.

`-I`*dir*

Adds *dir* to the list of directories that are searched for `#include` files with relative file names, that is, those not beginning with a `/` (slash).

The preprocessor first searches for `#include` files in the directory containing *sourcefile*, then in directories named with `-I` options, if any, and finally, in `/usr/include` or the directory specified by `-YI`.

`-i`

Passes the option to the linker to ignore any `LD_LIBRARY_PATH` setting.

`-keeptmp`

Retains temporary files created during compilation instead of deleting them automatically.

`-KPIC`

Produces position-independent code for use in shared libraries. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in `pc`-relative addressing mode through a procedure linkage table.

(SPARC) (PowerPC) With this option, the global offset table spans the range of 32-bit addresses in those rare cases where there are too many global data objects for `-Kpic`.

(Intel) `-KPIC` is identical to `-Kpic`.

`-Kpic`

Produces position-independent code for use in shared libraries.

(SPARC) (PowerPC) It is similar to `-KPIC`, but the size of the global offset table is limited to 8Kbytes.

There are two nominal performance costs with `-Kpic` and `-KPIC`:

- A routine compiled with either -Kpic or -KPIC executes a few extra instructions upon entry to set a register to point at a table (_GLOBAL_OFFSET_TABLE_) used for accessing a shared library's global or static variables.
- Each access to a global or static variable involves an extra indirect memory reference through _GLOBAL_OFFSET_TABLE_. If the compile is done with -KPIC, there are two additional instructions per global and static memory reference.

When considering the above costs, remember that the use of -Kpic and -KPIC can significantly reduce system memory requirements, due to the effect of library code sharing. Every page of code in a shared library compiled -Kpic or -KPIC can be shared by every process that uses the library. If a page of code in a shared library contains even a single non-pic (that is, absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether or not a .o file has been compiled with -Kpic or -KPIC is with the nm command:

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_
U _GLOBAL_OFFSET_TABLE_
```

A .o file containing position-independent code contains an unresolved external reference to _GLOBAL_OFFSET_TABLE_, as indicated by the letter U.

To determine whether to use -Kpic or -KPIC, use nm to identify the number of distinct global and static variables used or defined in the library. If the size of _GLOBAL_OFFSET_TABLE_ is under 8,192 bytes, you can use -Kpic. Otherwise, you must use -KPIC.

-L*dir*

Adds *dir* to the list of directories searched for libraries by ld(1). This option and its arguments are passed to ld.

`-l`*name*

Links with object library `lib`*name*`.so`, or `lib`*name*`.a`. The order of libraries in the command-line is important, as symbols are resolved from left to right.

This option must follow the *sourcefile* arguments.

`-mc`

Removes duplicate strings from the `.comment` section of the object file. When you use the `-mc` flag, `mcs -c` is invoked.

`-misalign`

(SPARC, PowerPC) Informs the compiler that the data in your program is not properly aligned, as in the following code:

```
char b[100];
int f(int *ar){
return  *(int *)(b +2) + *ar;
}
```

Thus, very conservative loads and stores must be used for data, one byte at a time. Using this option can cause significant degradation in the performance when you run the program. If you compile and link in separate steps, compiling with the `-misalign` option requires the option on the link step as well.

`-misalign2`

(SPARC, PowerPC) Like `-misalign`, assumes that data is not properly aligned, but that data is at least halfword-aligned. Though conservative uses of loads and stores must be used for data, the performance degradation when running a program is less than that seen for `-misalign`. If you compile and link in separate steps, compiling with the `-misalign2` option requires the option on the link step as well.

-mr

Removes all strings from the `.comment` section. When you use this flag, `mcs -d` is invoked.

-mr,*string*

Removes all strings from the `.comment` section and inserts *string* in that section of the object file. If *string* contains embedded blanks, it must be enclosed in quotation marks. A null *string* results in an empty `.comment` section. This option is passed as `-d` "*string*" to `mcs`.

-mt

Passes `-D_REENTRANT` to the preprocessor. Appends `-lthread`. If you are doing your own multithread coding, you must use this option in the compile and link steps. To obtain faster execution, this option requires a multiprocessor system. On a single-processor system, the resulting executable usually runs more slowly with this option.

-native

Ascertains which code-generation options *(SPARC)* or which processor *(Intel) (PowerPC)* are available on the machine running the compiler, and directs the compiler to generate code targeted for that machine.

This option is a synonym for `-xtarget=native`.

The `-fast` macro includes `-native` in its expansion.

-nofstore

(Intel) Does not convert the value of a floating-point expression or function to the type on the left-hand side of an assignment, when that expression or function is assigned to a variable or is cast to a shorter floating-point type; rather, it leaves the value in a register. See also "-fstore" on page 14.

`-noqueue`

Instructs the compiler not to queue this compile request if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

`-O`

Same as `-xO2`.

`-o` *filename*

Names the output file ***filename*** (as opposed to the default, `a.out`). ***filename*** cannot be the same as ***sourcefile***, since `cc` does not overwrite the source file. This option and its arguments are passed to `ld`(1).

`-P`

Runs the source file through the C preprocessor only. It then puts the output in a file with a `.i` suffix. Unlike `-E`, this option does not include preprocessor-type line number information in the output. See also the `-E` option.

`-p`

Prepares the object code to collect data for profiling with `prof`(1). This option invokes a runtime recording mechanism that produces a `mon.out` file at normal termination.

`-Q[y|n]`

Emits or does not emit identification information to the output file. `-Qy` is the default.

If `-Qy` is used, identification information about each invoked compilation tool is added to the `.comment` section of output files, which is accessible with `mcs`. This option can be useful for software administration.

`-Q`n supresses this information.

-qp

Same as -p.

-R*dir*[:*dir*]

Passes a colon-separated list of directories used to specify library search directories to the runtime linker. If present and not null, it is recorded in the output object file and passed to the runtime linker.

If both LD_RUN_PATH and the -R option are specified, the -R option takes precedence.

-S

Directs cc to produce an assembly source file but not to assemble the program.

-s

Removes all symbolic debugging information from the output object file. This option cannot be specified with -g.

Passed to ld(1).

-U*name*

Removes any initial definition of the preprocessor symbol *name*. This option is the inverse of the -D option. You can give multiple -U options.

-V

Directs cc to print the name and version ID of each pass as the compiler executes.

`-v`

Directs the compiler to perform stricter semantic checks and to enable other `lint`-like checks. For example, the code:

```
#include <stdio.h>
main(void)
{
    printf("Solipsism isn't for everybody.\n");
}
```

compiles and executes without problem. With `-v`, it still compiles; however, the compiler displays this warning:

```
"solipsism.c", line 5:  warning:  function has no return
    statement: main
```

`-v` does not give all the warnings that `lint`(1) does. Try running the above example through `lint`.

`-W`*c*`,`*arg*

Passes the argument *arg* to a specified component *c*. Each argument must be separated from the preceding only by a comma. All `-W` arguments are passed after the regular command-line arguments. A comma can be part of an argument by escaping it by an immediately preceding \ (backslash) character.

c can be one of the following:

`a`	Assembler: (`fbe`) ; (`gas`)
`c`	C code generator: (`cg`) *(SPARC)*; (`codegen`) *(Intel)(PowerPC)*
`l`	Link editor (`ld`)
`m`	mcs *(Solaris 2.x)*
`p`	Preprocessor (`cpp`)
`0`	Compiler (`acomp` and `ssbd`)
`2`	Optimizer: (`iropt`) *(SPARC)*; intermediate code translator: (`cg386`) *(Intel)*, (`cgppc`) *(PowerPC)*

-w

Suppresses compiler warning messages.

-X[a|c|s|t]

The -X (note uppercase X) options specify varying degrees of compliance to the ANSI C standard. -Xa is the default mode.

-Xa

(a = ANSI) ANSI C plus K&R C compatibility extensions, *with* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the ANSI C interpretation. *This is the default compiler mode.*

-Xc

(c = conformance) Issues errors and warnings for programs that use non-ANSI C constructs. This option is maximally conformant ANSI C, without K&R C compatibility extensions.

-Xs

(s = K&R C) Attempts to warn about all language constructs that have differing behavior between ANSI C and K&R C. The compiler language includes all features compatible with K&R C. This option invokes /usr/ccs/lib/cpp for preprocessing. __STDC__ is not defined in this mode. Differences between ANSI C and K&R C are discussed in the *C 4.2 Transition Guide*.

-Xt

(t = transition) This option uses ANSI C plus K&R C compatibility extensions *without* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, issues warnings about the conflict and uses the K&R C interpretation.

-x386

(Intel) Optimizes for the 80386 processor.

-x486

(Intel) Optimizes for the 80486 processor.

-xa

Inserts code to count how many times each basic block is executed. This option is the old style of basic block profiling for `tcov`. See `-xprofile=p` for information on the new style of profiling and the `tcov`(1) man page for more details. See also the *Profiling Tools* manual.

Invokes a runtime recording mechanism that creates a `.d` file for every `.c` file at normal termination. The `.d` file accumulates execution data for the corresponding source file. `tcov`(1) can then be run on the source file to generate statistics about the program. Since this option entails some optimization, it is incompatible with `-g`.

If set at compile-time, the `TCOVDIR` environment variable specifies the directory where the `.d` files are located. If this variable is not set, the `.d` files remain in the same directory as the `.c` files.

The `-xprofile=tcov` and the `-xa` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov`, and others with `-xa`. You cannot compile a single file with both options.

-xarch=*a*

Specifies the set of instructions the compiler may use.

a must be one of: `generic`, `v7`, `v8a`, `v8`, `v8plus`, `v8plusa, 386, pentium_pro, ppc, ppc_nofma.`

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option limits the instructions generated to those of the specified architecture, and *allows* the specified set of instructions. The option does not guarantee the specified set is used; however, under optimization, the set is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

`v7`, `v8`, and `v8a` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other and forward, but not backward. For any particular choice, the generated executable can run much more slowly on earlier architectures (to the left in the above list).

Table 2-3 The `-xarch` Values

Value	Meaning
`generic`	Gets good performance on most Intel, PowerPC and SPARC architectures, major degradation on none. This is the default. This option uses the best instruction set for good performance on most Intel, PowerPC, and SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.
`v7`	Limits instruction set to V7 architecture. This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions. This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions: The quad-precision floating-point instructions The integer `mul` and `div` instructions The `fsmuld` instruction Examples: SPARCstation 1, SPARCstation 2
`v8a`	Limits instruction set to the V8a version of the V8 architecture. By definition, V8a means the V8 architecture, but without: The quad-precision floating-point instructions The `fsmuld` instruction This option uses the best instruction set for good performance on the V8 architecture. Example: Any machine based on MicroSPARC™I chip architecture.

Table 2-3 The `-xarch` Values *(Continued)*

Value	Meaning
`v8`	Limits instruction set to V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions. Example: SPARCstation 10
`v8plus`	Limits instruction set to the V8plus version of the V9 architecture. By definition, V8plus, or V8+, means the V9 architecture, except: Without the quad-precision floating-point instructions Limited to the 32-bit subset defined by the V8+ specification Without the VIS instructions This option uses the best instruction set for good performance on the V8+ architecture. In V8+, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the i and l registers must not affect program results. Example: Any machine based on UltraSPARC™chip architecture. Use of this option also causes the `.o` file to be marked as a V8+ binary; such files will not run on a `v7` or `v8` machine.
`v8plusa`	Limits instruction set to the V8plusa version of the V9 architecture. By definition, V8plusa means the V8plus architecture, plus: The UltraSPARC-specific instructions The VIS instructions This option uses the best instruction set for good performance on the UltraSPARC architecture but limited to the 32-bit subset defined by the V8+ specification. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a Sun-specific V8+ binary; such files will not run on a `v7` or `v8` machine.

Table 2-3 The `-xarch` Values *(Continued)*

Value	Meaning
`386`	Limits instruction set to the Intel x86 architecture.
`pentium_pro`	Limits instruction set to the Intel pentium_pro architecture.
`ppc`	Limits instruction set to the PowerPC architecture.
`ppc_nofma`	Same as `ppc` but will not issue "fused multipy-add" instruction.

-xautopar

(SPARC) Turns on automatic parallelization for multiple processors. Does dependence analysis (analyze loops for inter-iteration data dependence) and loop restructuring. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is emitted.

Avoid `-xautopar` if you do your own thread management.

Parallelization options require a WorkShop license. To get faster execution, this option requires a multiple processor system. On a single-processor system, the resulting binary usually runs slower.

To determine how many processors you have, use the `psrinfo` command:

```
% psrinfo
0   on-line    since 01/12/95 10:41:54
1   on-line    since 01/12/95 10:41:54
2   on-line    since 01/12/95 10:41:54
3   on-line    since 01/12/95 10:41:54
```

To request a number of processors, set the `PARALLEL` environment variable. The default is 1.

- Do not request more processors than are available.
- If `N` is the number of processors on the machine, then for a one-user, multiprocessor system, try `PARALLEL=N-1`.

If you use `-xautopar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe C runtime library. If you use `-xautopar` and compile and link in *separate* steps, then you must also link with `-xautopar`.

-xcache=*c*

Defines the cache properties for use by the optimizer.

c must be one of the following:

- `generic (SPARC,`*`Intel`*`)`
- *s1*/*l1*/*a1*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*

The *si*/*li*/*ai* are defined as follows:

si	The size of the data cache at level *i*, in kilobytes
li	The line size of the data cache at level *i*, in bytes
ai	The associativity of the data cache at level *i*

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option specifies the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

Table 2-4 The `-xcache` Values

Value	Meaning
`generic`	Define the cache properties for good performance on most Intel, PowerPC, and SPARC architectures. This is the default value which directs the compiler to use cache properties for good performance on most Intel, PowerPC, and SPARC processors, without major performance degradation on any of them. With each new release, these best timing properties will be adjusted, if appropriate.
s1/*l1*/*a1*	Define level 1 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*	Define levels 1 and 2 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*	Define levels 1, 2, and 3 cache properties.

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 cache has:	Level 2 cache has:
16K bytes	1024K bytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

-xCC

Accepts the C++-style comments. In particular, `//` can be used to indicate the start of a comment.

-xcg[89|92]

(SPARC).

`-xcg89` is a macro for: `-xarch=v7 -xchip=old -xcache=64/32/1`.

`-xcg92` is a macro for:
`-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1`.

-xchip=*c*

Specifies the target processor for use by the optimizer.

c must be one of the following: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`, `ultra,386,486,pentium,` `pentium_pro`, `603`, `604`.

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option specifies timing properties by specifying the target processor.

Some effects are:

- The ordering of instructions, that is, scheduling

- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

Table 2-5 The `-xchip` Values

Value	Meaning
`generic`	Use timing properties for good performance on most Intel, PowerPC, and SPARC architectures. This is the default value that directs the compiler to use the best timing properties for good performance on most processors, without major performance degradation on any of them.
`old`	Uses timing properties of pre-SuperSPARC™ processors.
`super`	Uses timing properties of the SuperSPARC chip.
`super2`	Uses timing properties of the SuperSPARC II chip.
`micro`	Uses timing properties of the microSPARC chip.
`micro2`	Uses timing properties of the microSPARC II chip.
`hyper`	Uses timing properties of the hyperSPARC™ chip.
`hyper2`	Usse timing properties of the hyperSPARC II chip.
`powerup`	Uses timing properties of the Weitek® PowerUp™ chip.
`ultra`	Uses timing properties of the UltraSPARC chip.
`386`	Uses timing properties of the Intel 386 architecture.
`486`	Uses timing properties of the Intel 486 architecture
`pentium`	Uses timing properties of the Intel pentium architecture
`pentium_pro`	Uses timing properties of the Intel pentium_pro architecture
`603`	Uses timing properties of the PowerPC 603 chip.
`604`	Uses timing properties of the PowerPC 604 chip.

-xcrossfile

(SPARC) Enables optimization and inlining across source files.

Only effective with `-xO4` or `-xO5`, the compiler is allowed to analyze all the files on the command line as if they had been concatenated into a single file.

-xdepend

(SPARC) Analyzes loops for inter-iteration data dependencies and does loop restructuring. Loop restructuring includes loop interchange, loop fusion, scalar replacement, and elimination of "dead" array assignments. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued.

Dependency analysis is also included with `-xautopar` or `-xparallel`. The dependency analysis is done at compile time.

Dependency analysis may help on single-processor systems. However, if you try `-xdepend` on single-processor systems, you should not use either `-xautopar` or `-xexplicitpar`. If either of them is on, then the `-xdepend` optimization is done for multiple-processor systems.

-xe

Performs only syntax and semantic checking on the source file, but does not produce any object or executable code.

-xexplicitpar

(SPARC) Generates parallelized code based on specification of `#pragma MP` directives. You do the dependency analysis: analyze and specify loops for inter-iteration data dependencies. The software parallelizes the specified loops. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued. Avoid `-xexplicitpar` if you do your own thread management.

Parallelization options require a WorkShop license. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you indentify a loop for parallelization, and the loop has dependencies, you can get incorrect results, possibly different ones with each run, and with no warnings. Do not apply an explicit parallel pragma to a reduction loop. The explicit parallelization is done, but the reduction aspect of the loop is not done, and the results can be incorrect.

In summary, to parallelize explicitly:

- Analyze the loops to find those that are safe to parallelize.

- Insert `#pragma MP` to parallelize a loop. See the Section , “Explicit Parallelization and Pragmas” in Chapter , “Sun ANSI C Compiler-Specific Information,” for more information.
- Use the `-xexplicitpar` option.

An example of inserting a parallel pragma immediately before the loop is:

```
#pragma MP taskloop
  for (j=0; j<1000; j++){
  ...
  }
```

If you use `-xexplicitpar` and compile and link in *one* step, then linking automatically includes the microtasking library and the threads-safe C runtime library. If you use `-xexplicitpar` and compile and link in *separate* steps, then you must also *link* with `-xexplicitpar`.

-xF

Enables performance analysis of the executable using the Analyzer. (See the `analyzer`(1) man pages.) Produces code that can be reordered at the function level. Each function in the file is placed in a separate section; for example, functions `foo()` and `bar()` are placed in the sections `.text%foo` and `.text%bar`, respectively. Function ordering in the executable can be controlled by using `-xF` in conjunction with the `-M` option to `ld` (see `ld`(1)). This option also causes the assembler to generate some debugging information in the object file, necessary for data collection.

-xhelp=*f*

Displays on-line help information.

f must be one of: `flags`, `readme`, or `errors`.

`-xhelp=flags` displays a summary of the compiler options.

`-xhelp=readme` displays the `README` file.

`-xhelp=errors` displays the Error and Warning Messages file.

-xildoff

Turns off the incremental linker and forces the use of `ld`. This option is the default if you do not use the `-g` option, or you do use the `-G` option, or any source files are present on the command line. Override this default by using the `-xildon` option.

-xildon

Turns on the incremental linker and forces the use of `ild` in incremental mode. This option is the default if you use the `-g` option, and you do not use the `-G` option, and there are no source files present on the command line. Override this default by using the `-xildoff` option.

-xinline=[*f1*,...,*fn*]

Tries to inline only those named in the list *f1* to *fn* for user-written routines. The list is a comma-separated list of functions and subroutines.

If you are compiling with `-xO3`, you can use this option to increase optimization by inlining some routines. The `-xO3` option does not inline by itself.

If you are compiling with `-xO4`, using this option can decrease optimization by restricting inlining to only those routines in the list. With `-xO4`, the compiler normally tries to inline all user-written subroutines and functions. When `xinline=` is specified with an empty `rlst`, it indicates that none of the routines in the source file are to be inlined.

A routine is not inlined if any of the following conditions apply. No warning is issued.

- Optimization is less than `-xO3`.
- The routine cannot be found.
- Inlining the routine does not look practicable to the optimizer.
- Source for the routine is not in the file being compiled (however, see `-xcrossfile`).

-xlibmieee

Forces IEEE 754 style return values for math routines in exceptional cases. In such cases, no exception message is printed, you should not rely on `errno`.

-xlibmil

Inlines some library routines for faster execution. This option selects the appropriate assembly language inline templates for the floating-point option and platform for your system.

-xlic_lib=*l*

(SPARC, Intel) Links in the Sun-supplied licensed libraries specified in *l*, where *l* is a comma-separated list of libraries.

-Xlic_lib=sunperf

Links with specified Sun-supplied licensed libraries. Specifies a comma-separated list of license-controlled libraries to link with. For example:

```
cc -o pgx pgx.c -xlic_lib=sunpref
```

This option, like `-l`, should appear at the end of the command line, after source or object filenames.

-xlicinfo

Returns information about the licensing system. In particular, this option returns the name of the license server and the IDs of users who have checked out licenses. When you give this option, the compiler is not invoked, and a license is not checked out.

-xloopinfo

(SPARC) Shows which loops are parallelized and which are not. Gives a short reason for not parallelizing a loop. The `-xloopinfo` option is valid only if `-xautopar`, or `-xparallel`, or `-xexplicitpar` is specified; otherwise, the compiler issues a warning.

Parallelization options require a WorkShop license. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

-xM

Runs only the macro preprocessor on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output (see make(1) for details about makefiles and dependencies).

For example:

```
#include <unistd.h>
void main(void)
{}
```

generates this output:

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

-xM1

Collects dependencies like -xM, but excludes /usr/include files.For example:

```
more hello.c
#include<stdio.h>
main()
{
 (void)printf("hello\n");
}
cc -xM hello.c
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

Compilng with -xM1 does not report header file dependencies:

```
cc -xM1 hello.c
hello.o: hello.c
```

-xMerge

Merges data segments into text segments. Data initialized in the object file produced by this compilation is read-only and (unless linked with `ld -N`) is shared between processes.

-xnolib

Does not link any libraries by default; that is, no -l options are passed to ld. Normally, the cc driver passes -lc to ld.

When you use -xnolib, you have to pass all the -l options yourself. For example:

```
% cc test.c -xnolib -Bstatic -lm -Bdynamic -lc
```

links libm statically and the other libraries dynamically.

-xnolibmil

Does *not* inline math library routines. Use it after the `-fast` option. For example: `% cc -fast -xnolibmil....`

-xO[1|2|3|4|5]

Optimizes the object code. Specifying `-xO` is equivalent to specifying `-xO2`.

When `-O` is used with the `-g` option, a limited amount of debugging is available.

The levels (`1`, `2`, `3`, `4`, or `5`) you can use with `-xO` differ according to the platform you are using.

(SPARC)

`-xO1`
Does basic local optimization (peephole).

`-xO2`
Does basic local and global optimization. This is induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, and complex expression expansion.

The `-xO2` level does not assign global, external, or indirect references or definitions to registers. It treats these references and definitions as if they were declared `volatile`. In general, the `-xO2` level results in minimum code size.

`-xO3`
Performs like `-xO2`, but also optimizes references or definitions for external variables. Loop unrolling and software pipelining are also performed. This level does not trace the effects of pointer assignments. When compiling either device drivers, or programs that modify external variables from within signal handlers, you may need to use the `volatile` type qualifier to protect the object from optimization. In general, the `-xO3` level results in increased code size.

`-xO4`
Performs like `-xO3`, but also automatically inlines functions contained in the same file; this usually improves execution speed. This level traces the effects of pointer assignments, and usually results in increased code size.

`-xO5`
Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See `-xprofile=p`.

(Intel) (PowerPC)

`-xO1`
Preloads arguments from memory, cross-jumping (tail-merging), as well as the single pass of the default optimization.

`-xO2`
Schedules both high- and low-level instructions and performs improved spill analysis, loop memory-reference elimination, register lifetime analysis, enhanced register allocation, and elimination of global common subexpressions.

`-xO3`
Performs loop strength reduction, induction variable elimination, as well as the optimization done by level `2`.

`-xO4`
Performs loop unrolling, avoids creating stack frames when possible, and automatically inlines functions contained in the same file, as well as the optimization done by levels `2` and `3`. Note that this optimization level can cause stack traces from `adb` and `dbx` to be incorrect.

`-xO5`
Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Some of these include generating local calling convention entry points for exported functions, further optimizing spill code and adding analysis to improve instruction scheduling.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the command-line option.

If you optimize at `-xO3` or `-xO4` with very large procedures (thousands of lines of code in the same procedure), the optimizer may require a large amount of virtual memory. In such cases, machine performance may degrade.

-xP

Prints prototypes for all K&R C functions defined in this module.

```
f()
{
}

main(argc,argv)
int argc;
char *argv[];
{
}
```

produces this output:

```
int f(void);
int main(int, char **);
```

-xparallel

(SPARC) Parallelizes loops both automatically by the compiler and explicitly specified by the programmer. The `-xparallel` option is a macro, and is equivalent to specifying all three of `-xautopar`, `-xdepend`, and `-xexplicitpar`. With explicit parallelization of loops, there is a risk of producing incorrect results. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued.

Avoid `-xparallel` if you do your own thread management.

Parallelization options require a WorkShop license. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you compile and link in *one* step, `-xparallel` links with the microtasking library and the threads-safe C runtime library. If you compile and link in *separate* steps, and you compile with `-xparallel`, then link with `-xparallel`

`-xpentium`

(Intel) Optimizes for the Pentium™ processor.

`-xpg`

Prepares the object code to collect data for profiling with `gprof`(1). It invokes a runtime recording mechanism that produces a `gmon.out` file at normal termination.

`-xprofile=`***p***

Collects data for a profile or uses a profile to optimize.

(SPARC) ***p*** must be `collect`[:***name***], `use`[:***name***], or `tcov`.

(Intel) (PowerPC) ***p*** must be `tcov`.

This option causes execution frequency data to be collected and saved during execution, then the data can be used in subsequent runs to improve performance. This option is only valid when you specify a level of optimization.

`collect[:`***name***`]`

Collects and saves execution frequency data for later use by the optimizer with `-xprofile=use`. The compiler generates code to measure statement execution frequency.

The ***name*** is the name of the program that is being analyzed. This name is optional. If ***name*** is not specified, `a.out` is assumed to be the name of the executable.

At runtime a program compiled with `-xprofile=collect:`*name* will create the subdirectory *name*`.profile` to hold the runtime feedback information. Data is written to the file `feedback` in this subdirectory. If you run the program several times, the execution frequency data accumulates in the `feedback` file; that is, output from prior runs is not lost.

`use[:`*name*`]`

Uses execution frequency data to optimize strategically.

As with `collect:`*name*, the *name* is optional and may be used to specify the name of the program.

The program is optimized by using the execution frequency data previously generated and saved in the `feedback` files written by a previous execution of the program compiled with `-xprofile=collect`.

The source files and other compiler options must be exactly the same as those used for the compilation that created the compiled program that generated the `feedback` file. If compiled with `-xprofile=collect:`*name*, the same program name *name* must appear in the optimizing compilation: `-xprofile=use:`*name*.

`tcov`

Basic block coverage analysis using "new" style `tcov`.

The `-xprofile=tcov` option is the new style of basic block profiling for `tcov`. It has similar functionality to the `-xa` option, but correctly collects data for programs that have source code in header files. See `-xa` for information on the old style of profiling, the `tcov`(1) man page, and the *Performance Profiling Tools* manual for more details.

Code instrumentation is performed similarly to that of the `-xa` option, but `.d` files are no longer generated. Instead, a single file is generated, the name of which is based on the final executable. For example, if the program is run out of `/foo/bar/myprog.profile`, the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`.

The `-xprofile=tcov` and the `-xa` options are compatible in a single executable. That is, you can link a program that contains some files that have been compiled with `-xprofile=tcov`, and others with `-xa`. You cannot compile a single file with both options.

When running `tcov`, you must pass it the `-x` option to make it use the new style of data. If not, `tcov` uses the old `.d` files, if any, by default for data, and produces unexpected output.

Unlike the `-xa` option, the `TCOVDIR` environment variable has no effect at compile-time. However, its value is used at program runtime. See `tcov`(1) and the *Performance Profiling Tools* manual for more details.

-xreduction

(SPARC) Turns on reduction recognition during automatic parallelization. `-xreduction` must be specified with `-xautopar`, or `-xparallel`.

Parallelization options require a WorkShop license.

When reduction recognition is enabled, the compiler parallelizes reductions such as *dot* products, maximum and minimum finding. These reductions yield different roundoffs than obtained by unparallelized code.

-xregs=*r*

(SPARC) Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following: [`no%`]`appl`, [`no%`]`float`.

Example: `-xregs=appl,no%float`

Table 2-6 The `-xregs` Values

Value	Meaning
`appl`	Allows using the registers `g2`, `g3`, and `g4`. In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.

Table 2-6 The `-xregs` Values

Value	Meaning
`no%appl`	Does not use the `appl` registers.
`float`	Allows using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.
`no%float`	Does not use the floating-point registers. With this option, a source program cannot contain any floating-point code.

The default is `-xregs=appl,float`.

`-xrestrict=`*f*

(SPARC) Treats pointer-valued function parameters as restricted pointers. *f* is a comma-separated list that consists of one or more function parameters, `%all`, or `%none`.

If a function list is specified with this option, pointer parameters in the specified functions are treated as restricted; if `-xrestrict=%all` is specified, all pointer parameters in the entire C file are treated as restricted. Refer to Chapter 3, “Sun ANSI C Compiler-Specific Information”, “_Restrict Keyword” on page 59, for more information.

This command-line option can be used on its own, but it is best used with optimization. For example, the command:

```
%cc -xO3 -xrestrict=%all prog.c
```

treats all pointer parameters in the file `prog.c` as restricted pointers. The command:

```
%cc -xO3 -xrestrict=agc prog.c
```

treats all pointer parameters in the function `agc` in the file `prog.c` as restricted pointers.

The default is `%none`; specifying `-xrestrict` is equivalent to specifying `-xrestrict=%all`.

-xs

Disables Auto-Read for dbx. Use this option in case you cannot keep the .o files around. It passes the -s option to the assembler.

No Auto-Read is the older way of loading symbol tables. It places all symbol tables for dbx in the executable file. The linker links more slowly and dbx initializes more slowly.

Auto-Read is the newer and default way of loading symbol tables. With Auto-Read, the information is distributed in the .o files, so that dbx loads the symbol table information only if and when it is needed. Hence, the linker links faster, and dbx initializes faster.

With -xs, if you move the executables to another directory, then to use dbx, you can ignore the object (.o) files.

Without -xs, if you move the executables, you must move both the source files and the object (.o) files, or set the path with the dbx pathmap or use command.

-xsafe=mem

(SPARC) Allows the compiler to assume no memory-based traps occur.

This option grants permission to use the speculative load instruction on V9 machines.

-xsb

Generates extra symbol table information for the SourceBrowser.This option is not valid with the -Xs mode of the compiler.

-xsbfast

Creates the database for the SourceBrowser. Does not compile source into an object file. This option is not valid with the -Xs mode of the compiler.

-xsfpconst

Represents unsuffixed floating-point constants as single precision, instead of the default mode of double precision. Not valid with `-Xc`.

-xspace

Does no optimizations or parallelization of loops that increase code size.

Example: The compiler will not unroll loops or parallelize loops if it increases code size.

-xstrconst

Inserts string literals into the read-only data section of the text segment instead of the default data segment.

-xtarget=*t*

Specifies the target system for instruction set and optimization.

t must be one of: `native`, `generic`, *system-name (SPARC, Intel, ppc).*

The `-xtarget` option is a macro that permits a quick and easy specification of the `-xarch`, `-xchip`, and `-xcache` combinations that occur on real systems. The only meaning of `-xtarget` is in its expansion.

Table 2-7 The `-xtarget` Values

Value	Meaning
`native`	Gets the best performance on the host system. The compiler generates code for the best performance on the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
`generic`	Gets the best performance for generic architecture, chip, and cache. The compiler expands `-xtarget=generic` to: `-xarch=generic -xchip=generic -xcache=generic` This is the default value.
system-name	Gets the best performance for the specified system. You select a system name from Table 2-8 that lists the mnemonic encodings of the actual system name and numbers.

The performance of some programs may benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a generic specification is sufficient.

Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. See Table 2-8 for the values. For example:

`-xtarget=sun4/15` is equivalent to:
`-xarch=v8a -xchip=micro -xcache=2/16/1`

Table 2-8 The `-xtarget` Expansions

-xtarget	-xarch	-xchip	-xcache
sun4/15	v8a	micro	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/30	v8a	micro	2/16/1
sun4/40	v7	old	64/16/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/65	v7	old	64/16/1
sun4/75	v7	old	64/32/1
sun4/110	v7	old	2/16/1
sun4/150	v7	old	2/16/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/630	v7	old	64/32/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1

Table 2-8 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ssvyger	v8a	micro2	8/16/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1

Table 2-8 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss20/50	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/502	v8	super	16/32/4
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/712	v8	super	16/32/4:1024/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1

Table 2-8 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
ss1000	v8	super	16/32/4:1024/32/1
sc2000	v8	super	16/32/4:2048/64/1
cs6400	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ultra	v8	ultra	16/32/1:512/64/1
ultra2	v8	ultra2	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/200	v8	ultra	16/32/1:512/64/1
ultra2/1170	v8	ultra	16/32/1:512/64/1
ultra2/1200	v8	ultra	16/32/1:1024/64/1
ultra2/1300	v8	ultra2	16/32/1:2048/64/1
ultra2/2170	v8	ultra	16/32/1:512/64/1
ultra2/2200	v8	ultra	16/32/1:1024/64/1
ultra2/2300	v8	ultra2	16/32/1:2048/64/1
entr2	v8	ultra	16/32/1:512/64/1
entr2/1170	v8	ultra	16/32/1:512/64/1
entr2/2170	v8	ultra	16/32/1:512/64/1
entr2/1200	v8	ultra	16/32/1:512/64/1
entr2/2200	v8	ultra	16/32/1:512/64/1

Table 2-8 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
entr150	v8	ultra	16/32/1:512/64/1
entr3000	v8	ultra	16/32/1:512/64/1
entr4000	v8	ultra	16/32/1:512/64/1
entr5000	v8	ultra	16/32/1:512/64/1
entr6000	v8	ultra	16/32/1:512/64/1

For PowerPC: `-xtarget=` accepts `generic` or `native`.

For Intel: `-xtarget=` accepts:
- `generic` or `native`
- `386` (equivalent to `-386` option) or `486` (equivalent to `-486` option)
- `pentium` (equivalent to `-pentium` option) or `pentium_pro`

-xtemp=*dir*

Sets the directory for temporary files used by `cc` to *dir*. No space is allowed within this option string. Without this option, temporary files go into `/tmp`. `-xtemp` has precedence over the `TMPDIR` environment variable.

-xtime

Reports the time and resources used by each compilation component.

-xtransition

Issues warnings for the differences between K&R C and Sun ANSI C. The following warnings no longer appear unless the `-xtransition` option is used:

```
\a is ANSI C "alert" character
\x is ANSI C hex escape
bad octal digit
base type is really type tag: name
comment is replaced by "##"
comment does not concatenate tokens
```

```
declaration introduces new type in ANSI C: type tag
macro replacement within a character constant
macro replacement within a string literal
no macro replacement within a character constant
no macro replacement within a string literal
operand treated as unsigned
trigraph sequence replaced
ANSI C treats constant as unsigned: operator
semantics of operator change in ANSI C; use explicit cast
```

-xunroll=*n*

Suggests to the optimizer to unroll loops *n* times. *n* is a positive integer. When *n* is 1, it is a command, and the compiler unrolls no loops. When *n* is greater than 1, the `-xunroll=`*n* merely suggests to the compiler that it unroll loops *n* times.

-xvpara

(SPARC) Warns about loops that have `#pragma MP` directives specified when the loop may not be properly specified for parallelization. For example, when the optimizer detects data dependencies between loop iterations, it issues a warning.

Parallelization options require a WorkShop license.

Use `-xvpara` with the `-xexplicitpar` option or the `-xparallel` option and the `#pragma MP`. See the Section , “Explicit Parallelization and Pragmas” in Chapter , “Sun ANSI C Compiler-Specific Information,” for more information.

-Y*c*, *dir*

Specifies a new directory *dir* for the location of component *c*. *c* can consist of any of the characters representing components that are listed under the `-W` option.

If the location of a component is specified, then the new path name for the tool is *dir/tool*. If more than one `-Y` option is applied to any one item, then the last occurrence holds.

`-YA, dir`

Changes the default directory searched for components.

`-YI, dir`

Changes the default directory searched for `include` files.

`-YP, dir`

Changes the default directory for finding libraries files.

`-YS, dir`

Changes the default directory for startup object files.

`-Zll`

(SPARC) Creates the program database for `lock_lint`, but does not actually compile. Refer to the `lock_lint`(1) man page for more details.

`-Zlp`

(SPARC) Prepares object files for the loop profiler, `looptool`. The `looptool`(1) utility can then be run to generate loop statistics about the program. Use this option with `-xdepend`; if `-xdepend` is not explicitly or implicitly specified, turns on `-xdepend` and issues a warning. If optimization is not at `-xO3` or higher, optimization is raised to `-xO3` and a warning is issued. Generally, this option is used with one of the loop parallelization options: `-xexplicitpar`, `-xautopar`, or `-xparallel`.

Parallelization options require a WorkShop license. To get faster code, this option requires a multiprocessor system. On a single-processor system, the generated code usually runs slower.

If you compile and link in separate steps, and you compile with `-Zlp`, then be sure to *link* with `-Zlp`.

If you compile *one* subprogram with `-Zlp`, you need not compile *all* subprograms of that program with `-Zlp`. However, you get loop information only for the files compiled with `-Zlp`, and no indication that the program includes other files.

-Ztha

(SPARC) Prepares code for analysis by the Thread Analyzer, the performance analysis tool for multithreaded code. The `-Ztha` instrumentation performs two actions. It inserts calls to the profiling library at all procedure entries and exits in much the same way the `-p` does. However, instead of linking with the profiled libraries in `/usr/lib/libp`, code compiled with `-Ztha` links with the library `libtha.so`.

Options Passed to the Linker

`cc` recognizes `-a`, `-e`, `-r`, `-t`, `-u`, and `-z` and passes these options and their arguments to `ld`. `cc` passes any unrecognized options to `ld` with a warning.

Localization of Error Messages

The messages from the C compiler, and `lint`, can be localized using `gencat`. See `gencat(1)` and `catges(3C)` for more information on message catalogs.

The C compiler message text source file can be found relative to where the C compiler is installed. Using the default installation directory, the C compiler message text source files can be found in `/opt/SUNWspro/SC4.2/lib/locale/C/LC_MESSAGES`.

The C compiler message text source files are named:

```
SUNW_SPRO_SC_acomp.msg (C front end components acomp and ssbd)
SUNW_SPRO_SC_cc.msg     (cc and lint command)
```

After translating the messages, the `gencat` utility can be used to create the formatted message database catfiles. The C compiler uses these catfiles when issuing messages. In order for the C compiler to use the formatted message database catfiles, they must be:

- Named corretly
- Installed in the default location for the locale being used
- Referenced via the correct path in the environment variable `NLSPATH`

The formatted message database catfiles must be named:

```
SUNW_SPRO_SC_acomp.cat (c front end components acomp & ssbd)
SUNW_SPRO_SC_cc.cat    (cc and lint command)
```

To enable all users to use the message database catfiles, they should be installed in the proper location based on the locale. For example, if the locale is *French* (`fr`), and using the default installation of the C compiler, these files must reside in the directory: `/opt/SUNWspro/SC4.2/lib/locale/fr/LC_MESSAGES` and the environment variable `LC_MESSAGES` must be set to "`fr`" prior to invoking the `cc` command:

using `csh` shell:

```
% setenv LC_MESSAGES fr
```

Using `sh` shell:

```
$ LC_MESSAGES= fr
$ export LC_MESSAGES
```

Alternatively, they can be installed in a directory of choice and accessed by the C compiler by setting `NLSPATH` prior to invoking the C compiler. For example, if they are installed in `/usr/local/MyMessDir.NLSPATH` can be set as follows:

Using `csh` shell:

```
% setenv NLSPATH /usr/local/MyMessDir/%N.cat:$NLSPATH
```

Using `sh` shell:

```
$ NLSPATH=/usr/local/MyMessDir/%N.cat:$NLSPATH
$ export NLSPATH
```

Note – These formatted message database catfiles are shared between the `cc` and `lint` commands. For messages not translated, the default C locale translation is used.

`cflow(1)` and `cxref(1)` have their own message catalogs:

```
SUN_SPRO_SC_cflow.msg   SUNW_SPRO_SC_cflow.cat     cflow  command
SUNW_SPRO_SC_cxref.msg  SUNW_SPRO_SC_cxref.cat     cxref  command
```

Sun ANSI C Compiler-Specific Information

The Sun ANSI C compiler is compatible with the C language described in the American National Standard for Programming Language--C, ANSI/ISO 9899-1990. This chapter documents those areas specific to the Sun ANSI C compiler.

Environment Variables

TMPDIR

`cc` normally creates temporary files in the directory `/tmp`. You can specify another directory by setting the environment variable `TMPDIR` to the directory of your choice. However, if `TMPDIR` is not a valid directory, `cc` uses `/tmp`. The `-xtemp` option has precedence over the `TMPDIR` environment variable.

If you use a Bourne shell, type:

```
$ TMPDIR=dir; export TMPDIR
```

If you use a C shell, type:

```
% setenv TMPDIR dir
```

SUNPRO_SB_INIT_FILE_NAME

The absolute path name of the directory containing the `.sbinit`(5) file. This variable is used only if the `-xsb` or `-xsbfast` flag is used.

PARALLEL

(SPARC) Refer to `Environment Variable` on page 73 for details.

Global Behavior: Value versus `unsigned` Preserving

A program that depends on `unsigned` preserving arithmetic conversions behaves differently. This is considered to be the most serious change made by ANSI C.

In the first edition of K&R, *The C Programming Language* (Prentice-Hall, 1978), `unsigned` specified exactly one type; there were no `unsigned chars`, `unsigned shorts`, or `unsigned longs`, but most C compilers added these very soon thereafter.

In previous C compilers, the `unsigned` preserving rule is used for promotions: when an `unsigned` type needs to be widened, it is widened to an `unsigned` type; when an `unsigned` type mixes with a `signed` type, the result is an `unsigned` type.

The other rule, specified by ANSI C, came to be called "value preserving," in which the result type depends on the relative sizes of the operand types. When an `unsigned char` or `unsigned short` is widened, the result type is `int` if an `int` is large enough to represent all the values of the smaller type. Otherwise, the result type is `unsigned int`. The value preserving rule produces the least surprise arithmetic result for most expressions.

Only in the `-Xt` and `-Xs` modes does the compiler use the `unsigned` preserving promotions; in the other modes, `-Xc` and `-Xa`, the value preserving promotion rules are used. When the `-xtransition` option is used, the compiler warns about each expression whose behavior might depend on the promotion rules used.

Keywords

`asm` Keyword

The `_asm` keyword is a synonym for the `asm` keyword. `asm` is available under all compilation modes, although a warning is issued when it is used under the `-Xc` mode.

The `asm` statement has the form:

```
asm("string"):
```

where *string* is a valid assembly language statement.

For example:

```
main()
{
    int i;

    /* i = 10 */
    asm("mov 10,%l0");
    asm("st  %l0,[%fp-8]");

    printf("i = %d\n",i);
}
% cc foo.c
% a.out
i = 10
%
```

`asm` statements must appear within function bodies.

`_Restrict` *Keyword*

For a compiler to effectively perform parallel execution of a loop, it needs to determine if certain `lvalues` designate distinct regions of storage. Aliases are `lvalues` whose regions of storage are not distinct. Determining if two pointers to objects are aliases is a difficult and time-consuming process because it could require analysis of the entire program.

Example: the function `vsq()`

```
void vsq(int n, double * a, double * b)
{
    int i;
    for (i=0; i<n; i++) b[i] = a[i] * a[i];
}
```

The compiler can parallelize the execution of the different iterations of the loops if it knows that pointers a and b access different objects. If there is an overlap in objects accessed through pointers a and b then it would be unsafe for the compiler to execute the loops in parallel. At compile time, the compiler does not know if the objects accessed by a and b overlap by simply analyzing the function `vsq`(); the compiler may need to analyze the whole program to get this information.

Restricted pointers are used to specify pointers which designate distinct objects so that the compiler can perform pointer alias analysis. To support restricted pointers, the keyword `_Restrict` is recognized by the Sun ANSI C compiler as an extension. Below is an example of declaring function parameters of `vsq`() as restricted pointers:

```
void vsq(int n, double * _Restrict a, double * _Restrict b)
```

Pointers a and b are declared as restricted pointers, so the compiler knows that the regions of storage pointed to by a and b are distinct. With this alias information, the compiler is able to parallelize the loop.

The `_Restrict` keyword is a type qualifier, like `volatile`, and it qualifies pointer types only. `_Restrict` is recognized as a keyword only for compilation modes `-Xa` (default) and `-Xt`. For these two modes, the compiler defines the macro `__RESTRICT` to enable users write portable code with restricted pointers.

The compiler defines the macro `__RESTRICT` to enable users to write portable code with restricted pointers. For example, the following code works on the Sun ANSI C compiler in all compilation modes, and should work on other compilers which do not support restricted pointers:

```
#ifdef __RESTRICT
#define restrict _Restrict
#else
#define restrict
#endif

void vsq(int n, double * restrict a, double * restrict b)
{
    int i;
    for (i=0; i<n; i++) b[i] = a[i] * a[i];
}
```

If restricted pointers become a part of the ANSI C Standard, it is likely that “restrict” will be the keyword. Users may want to write code with restricted pointers using:

```
#define restrict _Restrict
```

as in vsq() because this way there will be minimal changes should “restrict” become a keyword in the ANSI C Standard. The Sun ANSI C compiler uses _Restrict as the keyword because it is in the implementor's name space, so there is no conflict with identifiers in the user's name space.

There are situations where a user may not want to change the source code. One can specify pointer-valued function parameters to be treated as restricted pointers with the command-line option -xrestrict; refer to “-xrestrict=f” on page 43 for details.

If a function list is specified, pointer parameters in the specified functions are treated as restricted; otherwise, all pointer parameters in the entire C file are treated as restricted. For example, -xrestrict=vsq would qualify the pointers a and b given in “Example: the function vsq()” on page 59 with the keyword _Restrict.

It is critical that _Restrict be used correctly. If pointers qualified as restricted pointers point to objects which are not distinct, loops may be incorrectly parallelized, resulting in undefined behavior. For example, assume that pointers a and b of function vsq() point to objects which overlap, such that b[i] and a[i+1] are the same object. If a and b are not declared as restricted pointers, the loops will be executed serially. If a and b are incorrectly qualified as restricted pointers, the compiler may parallelize the execution of the loops; this is not safe, because b[i+1] should only be computed after b[i] has been computed.

long long *Data Type*

The Sun ANSI C compiler includes the data types long long, and unsigned long long, which are similar to the data type long. long long can store 64 bits of information; long can store 32 bits of information. long long is not available in -Xc mode.

Printing `long long` Data Types

To print or scan `long long` data types, prefix the conversion specifier with the letters "ll." For example, to print `llvar`, a variable of `long long` data type, in signed decimal format, use:

```
printf("%lld\n", llvar);
```

Usual Arithmetic Conversions

Some binary operators convert the types of their operands to yield a common type, which is also the type of the result. These are called the usual arithmetic conversions:

- If either operand is type `long double`, the other operand is converted to `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to `double`.
- Otherwise, if either operand has type `float`, the other operand is converted to `float`.
- Otherwise, the integral promotions are performed on both operands. Then, these rules are applied:
  - If either operand has type `unsigned long long int`, the other operator is converted to `unsigned long long int`.
  - If either operand has type `long long int`, the other operator is converted to `long long int`.
  - If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
  - Otherwise, if one operand has type `long int` and the other has type `unsigned int`, both operands are converted to `unsigned long int`.
  - Otherwise, if either operand has type `long int`, the other operand is converted to `long int`.
  - Otherwise, if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
  - Otherwise, both operands have type `int`.

Constants

This section contains information related to constants that is specific to the Sun ANSI C compiler.

Integral Constants

Decimal, octal, and hexadecimal integral constants can be suffixed to indicate type, as shown in the Table 3-1.

Table 3-1 Data Type Suffixes

Suffix	Type
`u` or `U`	`unsigned`
`l` or `L`	`long`
`ll` or `LL`	`long long`[1]
`lu, LU, Lu, lU, ul, uL, Ul, or UL`	`unsigned long`
`llu, LLU, LLu, llU, ull, ULL, uLL, Ull`	`unsigned long long`[1]

1. `long long` and `unsigned long long` are not available in `-Xc` mode.

When assigning types to unsuffixed constants, the compiler uses the first of this list in which the value can be represented, depending on the size of the constant:

- `int`
- `long int`
- `unsigned long int`
- `long long int`
- `unsigned long long int`

Character Constants

A multiple-character constant that is not an escape sequence has a value derived from the numeric values of each character. For example, the constant `'123'` has a value of:

Table 3-2 Multiple-character Constant (ANSI)

0	'3'	'2'	'1'

or `0x333231`.

With the `-Xs` option and in other, non-ANSI versions of C, the value is:

Table 3-3 Multiple-character Constant (non-ANSI)

0	'1'	'2'	'3'

or `0x313233`.

Include Files

To include any of the standard header files supplied with the C compilation system, use this format:

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

The angle brackets `(<>)` cause the preprocessor to search for the header file in the standard place for header files on your system, usually the `/usr/include` directory.

The format is different for header files that you have stored in your own directories:

```
#include "header.h"
```

The quotation marks (`"  "`) cause the preprocessor to search for `header.h` first in the directory of the file containing the `#include` line.

If your header file is not in the same directory as the sourcefiles that include it, specify the path of the directory in which it is stored with the `-I` option to `cc`. Suppose, for instance, that you have included both `stdio.h` and `header.h` in the source file `mycode.c`:

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

Suppose further that `header.h` is stored in the directory `../defs`. The command:

```
% cc -I../defs mycode.c
```

directs the preprocessor to search for `header.h` first in the directory containing `mycode.c`, then in the directory `../defs`, and finally in the standard place. It also directs the preprocessor to search for `stdio.h` first in `../defs`, then in the standard place. The difference is that the current directory is searched only for header files whose names you have enclosed in quotation marks.

You can specify the `-I` option more than once on the `cc` command-line. The preprocessor searches the specified directories in the order they appear. You can specify multiple options to `cc` on the same command-line:

```
% cc -o prog -I../defs mycode.c
```

Nonstandard Floating Point

IEEE 754 floating-point default arithmetic is “nonstop.” Underflows are “gradual.” Following is a summary of explanation. See the *Numerical Computation Guide* for details.

Nonstop means that execution does not halt on occurrences like division by zero, floating-point overflow, or invalid operation exceptions. For example, consider the following, where `x` is zero and `y` is positive:

```
z = y / x;
```

By default, `z` is set to the value `+Inf`, and execution continues. With the `-fnonstd` option, however, this code causes an exit, such as a core dump.

Here is how gradual underflow works. Suppose you have the following code:

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
    x = x / 10;
```

The first time through the loop, `x` is set to `1`; the second time through, to `0.1`; the third time through, to `0.01`; and so on. Eventually, `x` reaches the lower limit of the machine's capacity to represent its value. What happens the next time the loop runs?

Let's say that the smallest number characterizable is:

`1.234567e-38`

The next time the loop runs, the number is modified by "stealing" from the mantissa and "giving" to the exponent:

`1.23456e-39`

and, subsequently,

`1.2345e-40`

and so on. This is known as "gradual underflow," which is the default behavior. In nonstandard behavior, none of this "stealing" takes place; typically, `x` is simply set to zero.

Preprocessing Directives

This section describes assertions, pragmas, and predefined names.

Assertions

A line of the form:

```
#assert predicate (token-sequence)
```

associates the *token-sequence* with the predicate in the assertion name space (separate from the space used for macro definitions). The predicate must be an identifier token.

```
#assert predicate
```

asserts that *predicate* exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicates by default (not in `-Xc` mode):

```
#assert system (unix)
#assert machine (sparc) (SPARC)
#assert machine (i386) (Intel)
#assert machine (ppc) (PowerPC)
#assert cpu (sparc) (SPARC)
#assert cpu (i386) (Intel)
#assert cpu (ppc) (PowerPC)
```

`lint` provides the following predefinition predicate by default (not in `-Xc` mode):

```
#assert lint (on)
```

Any assertion may be removed by using `#unassert`, which uses the same syntax as `assert`. Using `#unassert` with no argument deletes all assertions on the predicate; specifying an assertion deletes only that assertion.

An assertion may be tested in a `#if` statement with the following syntax:

```
#if #predicate(non-empty token-list)
```

For example, the predefined predicate `system` can be tested with the following line:

```
#if #system(unix)
```

which evaluates true.

Pragmas

Preprocessing lines of the form:

```
#pragma pp-tokens
```

specify implementation-defined actions.

The following `#pragma`s are recognized by the compilation system:

- `#pragma align` *integer* (*variable*[,*variable*])—Makes all the mentioned variables memory aligned to *integer* bytes, overriding the default. The following limitations apply:
  - *integer* must be a power of 2 between 1 and 128; valid values are: 1, 2, 4, 8, 16, 32, 64, and 128.
  - *variable* is a global or static variable; it cannot be an automatic variable.
  - If the specified alignment is smaller than the default, the default is used.
  - The pragma line must appear before the declaration of the variables which it mentions; otherwise, it is ignored.
  - Any variable that is mentioned but not declared in the text following the pragma line is ignored. For example:

```
#pragma align 64 (aninteger, astring, astruct)

int aninteger;
static char astring[256];
struct astruct{int a; char *b;};
```

- `#pragma fini` (*f1*[,*f2*...,*fn*])—Causes the implementation to call functions *f1* to *fn* (finalization functions) after it calls `main()` routine. Such functions are expected to be of type `void` and to accept no arguments, and are called either when a program terminates under program control or when the containing shared object is removed from memory. As with "initialization functions," finalization functions are executed in the order processed by the link editors.
- `#pragma init` (*f1*[,*f2*...,*fn*])—Causes the implementation to call functions *f1* to *fn* (initialization functions) before it calls `main()` routine. Such functions are expected to be of type `void` and to accept no arguments, and are called while constructing the memory image of the program at the start

of execution. In the case of initializers in a shared object, they are executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation, such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editors, both static and dynamic.

- `#pragma ident` *string*—Places *string* in the `.comment` section of the executable.
- `#pragma int_to_unsigned` *function_name*—For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, changes the function return to be of type `int`.
- *(SPARC)* `#pragma MP serial_loop`—Refer to "Serial Pragmas" on page 74 for details.
- *(SPARC)* `#pragma MP serial_loop_nested`—Refer to "Serial Pragmas" on page 74 for details.
- *(SPARC)* `#pragma MP taskloop`—Refer to "Parallel Pragmas" on page 74 for details.
- *(SPARC)* `#pragma nomemorydepend`—This pragma specifies that for any iteration of a loop, there are no memory dependences. That is, within any iteration of a loop there are no references to the same memory. This pragma will permit the compiler (pipeliner) to schedule instructions, more effectively, within a single iteration of a loop. If any memory dependences exist within any iteration of a loop, the results of executing the program are undefined. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.
- *(SPARC)* `#pragma no_side_effect` *(funcname)*—*funcname* specifies the name of a function within the current translation unit. The function must be declared prior to the pragma. The pragma must be specified prior to the function's definition. For the named function, *funcname*, the pragma declares that the function has no side effects of any kind. The compiler can use this information when doing optimizations using the function. If the function does have side effects, the results of executing a program which calls this function are undefined. The compiler takes advantage of this information at optimization level of 3 or above.

- `#pragma pack(`*n*`)`—Controls the layout of structure offsets. *n* is a number, 1, 2, or 4, that specifies the strictest alignment desired for any structure member. If *n* is omitted, members are aligned on their natural boundaries. If you are using `#pragma pack(`*n*`)`, be sure to place it after all `#includes`.
- *(SPARC)* `#pragma pipeloop(`*n*`)`—This pragma accepts a positive constant integer value, or 0, for the argument *n*. This pragma specifies that a loop is pipelinable and the minimum dependence distance of the loop-carried dependence is *n*. If the distance is 0, then the loop is effectively a Fortran-style `doall` loop and should be pipelined on the target processors. If the distance is greater than 0, then the compiler (pipeliner) will only try to pipeline *n* successive iterations. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.
- `#pragma redefine_extname old_extname new_extname`—The pragma causes every externally defined occurrence of the name "old_extname" in the object code to be "new_extname". Such that, at link time only the name "new_extname" is seen by the loader.

  If `pragma redefine_extname` is encountered after the first use of "old_extname", as a function definition, an initializer, or an expression, the effect is undefined. (Not supported in `-Xs` and `-Xc` modes.)
- `#pragma unknown_control_flow (`***name***`, [,`***name***`])`—Specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.
- *(SPARC)* `#pragma unroll (`*unroll_factor*`)`—This pragma accepts a positive constant integer value for the argument *unroll_factor*. The pragma applies to the next `for` loop within the current block. For unroll factor other than 1, this directive serves as a suggestion to the compiler that the specified loop should be unrolled by the given factor. The compiler will, when possible, use that unroll factor. When the unroll factor value is 1, this directive serves as a command which specifies to the compiler that the loop is not to be unrolled. The compiler takes advantage of this information at optimization level of 3 or above.

- `#pragma weak` *symbol1* `[=`*symbol2*`]`—Defines a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not produce an error message if it is unable to resolve a weak symbol.

```
#pragma weak symbol
```

defines *symbol* to be a weak symbol. The linker does not produce an error message if it does not find a definition for *symbol*.

```
#pragma weak symbol1 = symbol2
```

defines *symbol1* to be a weak symbol, which is an alias for the symbol *symbol2*. This form of the pragma can only be used in the same translation unit where *symbol2* is defined, either in the sourcefiles or one of its included headerfiles. Otherwise, a compilation error will result.

If your program calls but does not define *symbol1*, and *symbol1* is a weak symbol in a library being linked,the linker uses the definition from that library. However, if your program defines its own version of *symbol1*, then the program's definition is used and the weak global definition of *symbol1* in the library is not used. If the program directly calls *symbol2*, the definition from the library is used; a duplicate definition of *symbol2* causes an error.

The compiler ignores unrecognized pragmas. Using the `-v` option will give a warning on unrecognized pragmas.

Predefined Names

The following identifier is predefined as an object-like macro:

Table 3-4 Predefined Identifier

Identifier	Description

`__STDC__`	`__STDC__ 1 -Xc`
	`__STDC__ 0 -Xa, -Xt`
	`Not defined -Xs`

The compiler will issue a warning if `__STDC__` is undefined (`#undef __STDC__`). `__STDC__` is not defined in `-Xs` mode.

Predefinitions (not valid in `-Xc` mode):

- `sun`
- `unix`
- `sparc` *(SPARC)*
- `i386` *(Intel)*

The following predefinitions are valid in all modes:

- `__sun`
- `__unix`
- `__SUNPRO_C=0x400`
- `__`uname -s`_`uname -r`` (example: `__SunOS_5_4`)
- `__sparc` *(SPARC)*
- `__i386` *(Intel)*
- `__BUILTIN_VA_ARG_INCR`
- `__SVR4`
- `__LITTLE_ENDIAN` *(PowerPC)*
- `__ppc` *(PowerPC)*

The compiler also predefines the object-like macro

```
__PRAGMA_REDEFINE_EXTNAME
```

to indicate that the pragma will be recognized.

The following is predefined in `-Xa` and `-Xt` modes only:

```
__RESTRICT
```

MP C (SPARC)

SunSoft MP C is an extended ANSI C compiler that can optimize code to run on SPARC shared-memory multiprocessor machines. The process is called *parallelizing*. The compiled code can execute in parallel using the multiple processors on the system.

The SunSoft WorkShop includes the license required to use the features of MP C.

This section contains an overview and example of using MP C, and documents the environment variable, keyword, pragmas, and options used with MP C.

Refer to the *"MP C" white paper*, located in `/opt/SUNWspro/READMEs/mpc.ps`, for examples on using MP C and for further reference information.

Overview

The MP C compiler generates parallel code for those loops that it determines are safe to parallelize. Typically, these loops have iterations that are independent of each other. For such loops, it does not matter in what order the iterations are executed or if they are executed in parallel. Many, although not all, vector loops fall into this category.

Because of the way aliasing works in C, it is difficult to determine the safety of parallelization. To help the compiler, MP C offers pragmas and additional pointer qualifications to provide aliasing information known to the programmer that the compiler cannot determine.

Example of Use

The following example illustrates the use of MP C and how parallel execution can be controlled. To enable parallelization of the target program, the "`-xautopar`" option can be used as follows:

```
% cc -fast -xO4 -xautopar example.c -o example
```

This generates an executable called `example`, which can be executed normally.

Environment Variable

If multiprocessor execution is desired, the `PARALLEL` environment variable needs to be set. It specifies the number of processors available to the program:

```
% setenv PARALLEL 2
```

This will enable the execution of the program on two threads. If the target machine has multiple processors, the threads can map to independent processors.

```
% example
```

Running the program will lead to creation of two threads that will execute the parallelized portions of the program.

Keyword

The keyword `_Restrict` can be used with MP C. Refer to the section "_Restrict Keyword" on page 59 for details.

Explicit Parallelization and Pragmas

Often, there is not enough information available for the compiler to make a decision on the legality or profitability of parallelization. MP C supports pragmas that allow the programmer to effectively parallelize loops that otherwise would be too difficult or impossible for the compiler to handle.

Serial Pragmas

There are two serial pragmas, and both apply to "`for`" loops:

- `#pragma MP serial_loop`
- `#pragma MP serial_loop_nested`

The `#pragma MP serial_loop` pragma indicates to the compiler that the next `for` loop is not to be implicitly/automatically parallelized.

The `#pragma MP serial_loop_nested` pragma indicates to the compiler that the next `for` loop and any `for` loops nested within the scope of this `for` loop are not to be implicitly/automatically parallelized. The scope of the `serial_loop_nested` pragma does not extend beyond the scope of the loop to which it applies.

Parallel Pragmas

There is one parallel pragma: `#pragma MP taskloop` [*options*].

The `MP taskloop` pragma can, optionally, take one or more of the following arguments.

- `maxcpus` (*number_of_processors*)
- `private` (*list_of_private_variables*)
- `shared` (*list_of_shared_variables*)
- `readonly` (*list_of_readonly_variables*)
- `storeback` (*list_of_storeback_variables*)
- `savelast`
- `reduction` (*list_of_reduction_variables*)

- `schedtype` (*scheduling_type*)

Only one option can be specified per `MP taskloop` pragma; however, the pragmas are cumulative and apply to the next `for` loop encountered within the current block in the source code:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop shared(a,b)
#pragma MP taskloop storeback(x)
```

These options may appear multiple times prior to the `for` loop to which they apply. In case of conflicting options, the compiler will issue a warning message.

Nesting of `for` *loops*

An `MP taskloop` pragma applies to the next `for` loop within the current block. There is no nesting of parallelized `for` loops by MP C.

Eligibility for Parallelizing

An `MP taskloop` pragma suggests to the compiler that, unless otherwise disallowed, the specified `for` loop should be parallelized.

`For` loops with irregular control flow and unknown loop iteration increment are not eligible for parallelization. For example, `for` loops containing setjmp, longjmp, exit, abort, return, goto, labels, and break should not be considered as candidates for parallelization.

Of particular importance is to note that `for` loops with inter-iteration dependencies can be eligible for explicit parallelization. This means that if a MP taskloop pragma is specified for such a loop the compiler will simply honor it, unless the `for` loop is disqualified. It is the user's responsibility to make sure that such explicit parallelization will not lead to incorrect results.

If both the `serial_loop` or `serial_loop_nested` and `taskloop` pragmas are specified for a `for` loop, the last one specified will prevail.

Consider the following example:

```
#pragma MP serial_loop_nested
    for (i=0; i<100; i++) {
   # pragma MP taskloop
     for (j=0; j<1000; j++) {
```

```
                  ...
      }
    }
```

The i loop will not be parallelized but the j loop might be.

Number of Processors

`#pragma MP taskloop maxcpus` (*number_of_processors*) specifies the number of processors to be used for this loop, if possible.

The value of `maxcpus` must be a positive integer. If `maxcpus` equals 1, then the specified loop will be executed in serial. (Note that setting `maxcpus` to be 1 is equivalent to specifying the `serial_loop` pragma.) The smaller of the values of `maxcpus` or the interpreted value of the `PARALLEL` environment variable will be used. When the environment variable `PARALLEL` is not specified, it is interpreted as having the value 1.

If more than one `maxcpus` pragma is specified for a `for` loop, the last one specified will prevail.

Classifying Variables

A variable used in a loop is classified as being either a “private”, “shared”, “reduction”, or “readonly” variable. The variable will belong to only one of these classifications. A variable can only be classified as a reduction or readonly variable via an explicit pragma. See `#pragma MP taskloop reduction` and `#pragma MP taskloop readonly`. A variable can be classified as being either a “private or “shared” variable via an explicit pragma or through the following default scoping rules.

Default Scoping Rules for Private and Shared Variables

A private variable is one whose value is private to each processor processing some iterations of a `for` loop. In other words, the value assigned to a private variable in one iteration of a `for` loop is not propagated to other processors processing other iterations of that `for` loop. A shared variable, on the other hand, is a variable whose current value is accessible by all processors processing iterations of a `for` loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop. Loops being explicitly parallelized through use of `#pragma MP taskloop` directives, that contain references to shared variables, must ensure that such sharing of values does not cause any

correctness problems (such as race conditions). No synchronization is provided by the compiler on updates and accesses to shared variables in an explicitly parallelized loop.

In analyzing explicitly parallelized loops, the compiler uses the following “default scoping rules” to determine whether a variable is private or shared:

- If a variable is not explicitly classified via a pragma, the variable will default to being classified as a shared variable if it is declared as a pointer or array, and is only referenced using array syntax within the loop. Otherwise, it will be classified as a private variable.
- The loop index variable is always treated as a private variable and is always a storeback variable.

It is ***highly recommended*** that all variables used in an explicitly parallelized `for` loop be explicitly classified as one of shared, private, reduction, or readonly, to avoid the “default scoping rules.”

Since the compiler does not perform any synchronization on accesses to shared variables, extreme care must be exercised before using an `MP taskloop` pragma for a loop that contains, for example, array references. If inter-iteration data dependencies exist in such an explicitly parallelized loop, then its parallel execution may give erroneous results. The compiler may or may not be able to detect such a potential problem situation and issue a warning message. In any case, the compiler will not disable the explicit parallelization of loops with potential shared variable problems.

Private Variables

`#pragma MP taskloop private` (*list_of_private_variables*) specifies all the variables that should be treated as private variables for this loop. All other variables used in the loop that are not explicitly specified as shared, readonly, or reduction variables, will be either shared or private as defined by the default scoping rules.

A private variable is one whose value is private to each processor processing some iterations of a loop. In other words, the value assigned to a private variable by one of the processors working on iterations of a loop is not propagated to other processors processing other iterations of that loop. A private variable has no initial value at the start of each iteration of a loop and must be set to a value within the iteration of a loop prior to its first use within

that iteration. Execution of a program with a loop containing an explicitly declared private variable whose value is used prior to being set will result in undefined behavior.

Shared Variables

`#pragma MP taskloop shared` (*list_of_shared_variables*) specifies all the variables that should be treated as shared variables for this loop. All other variables used in the loop that are not explicitly specified as private, readonly, storeback or reduction variables, will be either shared or private as defined by the default scoping rules.

A shared variable is a variable whose current value is accessible by all processors processing iterations of a `for` loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop.

Read-only Variables

Read-only variables are a special class of shared variables that are not modified in any iteration of a loop. `#pragma MP taskloop readonly` (*list_of_readonly_variables*) indicates to the compiler that it may use a separate copy of that variable's value for each processor processing iterations of the loop.

Storeback Variables

`#pragma MP taskloop storeback` (*list_of_storeback_variables*) specifies all the variables to be treated as storeback variables.

A storeback variable is one whose value is computed in a loop, and this computed value is then used after the termination of the loop. The last loop iteration values of storeback variables are available for use after the termination of the loop. Such a variable is a good candidate to be declared explicitly via this directive as a storeback variable when the variable is a private variable, whether by explicitly declaring the variable private or by the default scoping rules.

Note that the storeback operation for a storeback variable occurs at the last iteration of the explicitly parallelized loop, regardless of whether or not that iteration updates the value of the storeback variable. In other words the

processor that processes the last iteration of a loop may not be the same processor that currently contains the last updated value for a storeback variable. Consider the following example:

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
 for (i=1; i <= n; i++) {
     if (...) {
         x =...
   }
}
 printf ("%d", x);
```

In the above example the value of the storeback variable *x* printed out via the `printf`() call may not be the same as that printed out by a serial version of the i loop, because in the explicitly parallelized case, the processor that processes the last iteration of the loop (when i==n), which performs the storeback operation for *x* may not be the same processor that currently contains the last updated value for *x*. The compiler will attempt to issue a warning message to alert the user of such potential problems.

In an explicitly parallelized loop, variables referenced as arrays are not treated as storeback variables. Hence it is important to include them in the list_of_storeback_variables if such storeback operation is desired (for example, if the variables referenced as arrays have been declared as private variables).

Savelast

`#pragma MP taskloop savelast` specifies that all the private variables of a loop be treated as a storeback variables. The syntax of this pragma is as follows:

```
#pragma MP taskloop savelast
```

It is often convenient to use this form, rather than list out each private variable of a loop when declaring each variable as storeback variables.

Reduction Variables

`#pragma MP taskloop reduction` (*list_of_reduction_variables*) specifies that all the variables appearing in the reduction list will be treated as reduction variables for the loop. A reduction variable is one whose partial values can be individually computed by each of the processors processing iterations of the loop, and whose final value can be computed from all its partial values. The

presence of a list of reduction variables can facilitate the compiler in identifying that the loop is a reduction loop, allowing generation of parallel reduction code for it.

Consider the following example:

```
#pragma MP taskloop reduction(x)
    for (i=0; i<n; i++) {
       x = x + a[i];
  }
```

the variable *x* is a (sum) reduction variable and the i loop is a(sum) reduction loop.

Scheduling Control

The MP C compiler supports several pragmas that can be used in conjunction with the `taskloop` pragma to control the loop scheduling strategy for a given loop. The syntax for this pragma is:

`#pragma MP taskloop schedtype (`*scheduling_type*`)`

This pragma can be used to specify the specific scheduling_type to be used to schedule the parallelized loop. Scheduling_type can be one of the following:

- `static`

In static scheduling all the iterations of the loop are uniformly distributed among all the participating processors.

Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(static)
 for (i=0; i<1000; i++) {
   ...
}
```

In the above example, each of the four processors will process 250 iterations of the loop.

- `self` [(*chunk_size*)]

In `self` scheduling, each participating processor processes a fixed number of iterations (called the “chunk size”) until all the iterations of the loop have been processed. The optional chunk_size parameter specifies the “chunk size” to be used.   Chunk_size must be a positive integer constant, or variable of integral

type. If specified as a variable chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(self(120))
 for (i=0; i<1000; i++) {
    ...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

120, 120, 120, 120, 120, 120, 120, 120, 40.

- `gss` [(*min_chunk_size*)]

In `guided self` scheduling, each participating processor processes a variable number of iterations (called the "min chunk size") until all the iterations of the loop have been processed. The optional min_chunk_size parameter specifies that each variable chunk size used must be at least min_chunk_size in size. Min_chunk_size must be a positive integer constant, or variable of integral type. If specified as a variable min_chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

Example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
 for (i=0; i<1000; i++) {
    ...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 10, 10, 10.

- `factoring` [(*min_chunk_size*)]

In `factoring` scheduling, each participating processor processes a variable number of iterations (called the "min chunk size") until all the iterations of the loop have been processed. The optional min_chunk_size parameter specifies

that each variable chunk size used must be at least min_chunk_size in size. Min_chunk_size must be a positive integer constant, or variable of integral type. If specified as a variable min_chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(factoring(10))
  for (i=0; i<1000; i++) {
     ...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10.

Compiler Options

The following compiler options can be used in MP C. Refer to Chapter 2, “cc Compiler Options” for complete descriptions of the options.

“-xautopar” on page 27
“-xdepend” on page 31
“-xexplicitpar” on page 31
“-xloopinfo” on page 34
“-xparallel” on page 39
“-xreduction” on page 42
“-xrestrict=f” on page 43
“-xvpara” on page 52
“-Zlp” on page 53.

cscope: *Interactively Examining a C Program*

cscope is an interactive program that locates specified elements of code in C, lex, or yacc source files. With cscope, you can search and edit your source files more efficiently than you could with a typical editor. That's because cscope supports function calls—when a function is being called, when it is doing the calling—as well as C language identifiers and keywords.

This chapter is a tutorial on the cscope browser, which is provided with this release.

Note – SourceBrowser, a window-oriented code browser that is more powerful than cscope, is described briefly in "SourceBrowser" on page 102. SourceBrowser is sold separately.

The cscope *Process*

When cscope is called for a set of C, lex, or yacc source files, it builds a symbol cross-reference table for the functions, function calls, macros, variables, and preprocessor symbols in those files. You can then query that table about the locations of symbols you specify. First, it presents a menu and asks you to choose the type of search you would like to have performed. You may, for instance, want cscope to find all the functions that call a specified function.

When cscope has completed this search, it prints a list. Each list entry contains the name of the file, the number of the line, and the text of the line in which cscope has found the specified code. In our case, the list also includes

the names of the functions that call the specified function. You now have the option of requesting another search or examining one of the listed lines with the editor. If you choose the latter, `cscope` invokes the editor for the file in which the line appears, with the cursor on that line. You can now view the code in context and, if you wish, edit the file as any other file. You can then return to the menu from the editor to request a new search.

Because the procedure you follow depends on the task at hand, there is no single set of instructions for using `cscope`. For an extended example of its use, review the `cscope` session described in the next section. It shows how you can locate a bug in a program without learning all the code.

Basic Use

Suppose you are given responsibility for maintaining the program `prog`. You are told that an error message, `out of storage`, sometimes appears just as the program starts up. Now you want to use `cscope` to locate the parts of the code that are generating the message. Here is how you do it.

Step 1: Set Up the Environment

`cscope` is a screen-oriented tool that can only be used on terminals listed in the Terminal Information Utilities (`terminfo`) database. Be sure you have set the `TERM` environment variable to your terminal type so that `cscope` can verify that it is listed in the `terminfo` database. If you have not done so, assign a value to `TERM` and export it to the shell as follows:

In a Bourne shell, type:

```
$ TERM=term_name; export TERM
```

In a C shell, type:

```
% setenv TERM term_name
```

You may now want to assign a value to the `EDITOR` environment variable. By default, `cscope` invokes the `vi` editor. (The examples in this chapter illustrate `vi` usage.) If you prefer not to use `vi`, set the `EDITOR` environment variable to the editor of your choice and export `EDITOR`, as follows:

In a Bourne shell, type:

```
$ EDITOR=emacs; export EDITOR
```

In a C shell, type:

```
% setenv EDITOR emacs
```

You may have to write an interface between `cscope` and your editor. For details, see “Command-Line Syntax for Editors” on page 101.

If you want to use `cscope` only for browsing (without editing), you can set the `VIEWER` environment variable to `pg` and export `VIEWER`. `cscope` will then invoke `pg` instead of `vi`.

An environment variable called `VPATH` can be set to specify directories to be searched for source files. See “View Paths” on page 95.

Step 2: Invoke the `cscope` *Program*

By default, `cscope` builds a symbol cross-reference table for all the C, `lex`, and `yacc` source files in the current directory, and for any included header files in the current directory or the standard place. So, if all the source files for the program to be browsed are in the current directory, and if its header files are there or in the standard place, invoke `cscope` without arguments:

```
% cscope
```

To browse through selected source files, invoke `cscope` with the names of those files as arguments:

```
% cscope file1.c file2.c file3.h
```

For other ways to invoke `cscope`, see “Command-Line Options” on page 93.

`cscope` builds the symbol cross-reference table the first time it is used on the source files for the program to be browsed. By default, the table is stored in the file `cscope.out` in the current directory. On a subsequent invocation, `cscope` rebuilds the cross-reference only if a source file has been modified or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files is copied from the old cross-reference, which makes rebuilding faster than the initial build, and reduces startup time for subsequent invocations.

Step 3: Locate the Code

Now let's return to the task we undertook at the beginning of this section: to identify the problem that is causing the error message `out of storage` to be printed. You have invoked `cscope`, the cross-reference table has been built. The `cscope` menu of tasks appears on the screen.

The `cscope` Menu of Tasks:

```
% cscope

cscope              Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Press the Return key to move the cursor down the screen (with wraparound at the bottom of the display), and `^p` (Control-p) to move the cursor up; or use the up (`ua`) and down (`da`) arrow keys. You can manipulate the menu and perform other tasks with the following single-key commands:

Table 4-1 `cscope` Menu Manipulation Commands

Tab	Move to the next input field.
Return	Move to the next input field.
^n	Move to the next input field.
^p	Move to the previous input field.
^y	Search with the last text typed.
^b	Move to the previous input field and search pattern.
^f	Move to the next input field and search pattern.

Table 4-1 `cscope` Menu Manipulation Commands *(Continued)*

^c	Toggle ignore/use letter case when searching. For example, a search for `FILE` matches `file` and `File` when ignoring the letter case.
^r	Rebuild cross-reference.
!	Start an interactive shell. Type `^d` to return to `cscope`.
^l	Redraw the screen.
?	Display the list of commands.
^d	Exit `cscope`.

If the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a \ (backslash) before the character.

Now move the cursor to the fifth menu item, `Find this text string`, enter the text `out of storage`, and press the Return key.

`cscope` Function: Requesting a Search for a Text String:

```
$ cscope

cscope              Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

Note – Follow the same procedure to perform any other task listed in the menu except the sixth, `Change this text string`. Because this task is slightly more complex than the others, there is a different procedure for performing it. For a description of how to change a text string, see “Examples” on page 97.

---

cscope searches for the specified text, finds one line that contains it, and reports its finding.

cscope Function: Listing Lines Containing the Text String:

```
Text string:  out of storage

  File Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

After cscope shows you the results of a successful search, you have several options. You may want to change one of the lines or examine the code surrounding it in the editor. Or, if cscope has found so many lines that a list of them does not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after cscope has found the specified text:

Table 4-2 Commands for Use After an Initial Search

1 - 9	Edit the file referenced by this line. The number you type corresponds to an item in the list of lines printed by cscope.
Space	Display the next set of matching lines.
+	Display the next set of matching lines.
^v	Display the next set of matching lines.
—	Display the previous set of matching lines.
^e	Edit the displayed files in order.
>	Append the list of lines being displayed to a file.
|	Pipe all lines to a shell command.

Again, if the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash before the character.

Now examine the code around the newly found line. Enter `1` (the number of the line in the list). The editor is invoked with the file `alloc.c` with the cursor at the beginning of line 63 of `alloc.c`.

`cscope` Function: Examining a Line of Code:

```
{
    return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
    if (p == NULL) {
        (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
        exit(1);
    }
    return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

You can see that the error message is generated when the variable `p` is `NULL`. To determine how an argument passed to `alloctest()` could have been `NULL`, you must first identify the functions that call `alloctest()`.

Exit the editor by using normal quit conventions. You are returned to the menu of tasks. Now type `alloctest` after the fourth item, `Find functions calling this function`.

`cscope` Function: Requesting a List of Functions That Call `alloctest()`:

```
Text string:  out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

`cscope` finds and lists three such functions.

`cscope` Function: `Listing` Functions That Call `alloctest()`:

```
Functions calling this function:  alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned) nelem, (unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Now you want to know which functions call `mymalloc()`. `cscope` finds ten such functions. It lists nine of them on the screen and instructs you to press the space bar to see the rest of the list.

cscope Function: Listing Functions That Call mymalloc():

```
Functions calling this function:  mymalloc

    File           Function       Line
1   alloc.c        stralloc       24      return(strcpy(mymalloc(strlen(s) + 1), s));
2   crossref.c     crossref       47      symbol = (struct symbol *) mymalloc(msymbols *
                                                      sizeof(struct symbol));
3   dir.c          makevpsrcdirs63        srcdirs = (char **) mymalloc(nsrcdirs * sizeof(char
                                                      *));
4   dir.c          addincdir      167     incdirs = (char **) mymalloc(sizeof(char *));
5   dir.c          addincdir      168     incnames = (char **) mymalloc(sizeof(char *));
6   dir.c          addsrcfile     439     p = (struct listitem *) mymalloc(sizeof(struct
                                                      listitem));
7   display.c      dispinit       87      displine = (int *) mymalloc(mdisprefs * sizeof(int
                                                      ));
8   history.c      addcmd         19      h = (struct cmd *) mymalloc(sizeof(struct cmd));
9   main.c         main           212     s = mymalloc((unsigned ) (strlen(reffile) +
                                                      strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Because you know that the error message out of storage is generated at the beginning of the program, you can guess that the problem may have occurred in the function dispinit() (display initialization).

To view dispinit(), the seventh function on the list, type 7.

`cscope` Function: Viewing `dispinit()` in the Editor:

```
void
dispinit()
{
            /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
        mdisprefs = 9;
    }
            /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

`mymalloc()` failed because it was called either with a very large number or a negative number. By examining the possible values of `FLDLINE` and `REFLINE`, you can see that there are situations in which the value of `mdisprefs` is negative, that is, in which you are trying to call `mymalloc()` with a negative number.

Step 4: Edit the Code

On a windowing terminal, you may have multiple windows of arbitrary size. The error message `out of storage` might have appeared as a result of running `prog` in a window with too few lines. In other words, that may have been one of the situations in which `mymalloc()` was called with a negative number. Now you want to be sure that when the program aborts in this situation in the future, it does so after printing the more meaningful error message `screen too small`. Edit the function `dispinit()` as follows.

cscope Function: Correcting the Problem:

```
/* initialize display parameters */
void
dispinit()
{
            /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs <= 0) {
        (void) fprintf(stderr,"\n%s:  screen too small\n", argv0);
        exit(1);
    }
    if (mdisprefs > 9)
        mdisprefs = 9;
            /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
```

You have fixed the problem we began investigating at the beginning of this section. Now if `prog` is run in a window with too few lines, it does not simply fail with the unedifying error message `out of storage`. Instead, it checks the window size and generates a more meaningful error message before exiting.

Command-Line Options

As noted, `cscope` builds a symbol cross-reference table for the C, `lex`, and source files in the current directory by default. That is,

`%` **`cscope`**

is equivalent to:

`%` **`cscope *.[chly]`**

We have also seen that you can browse through selected source files by invoking `cscope` with the names of those files as arguments:

`%` **`cscope`** ***file1*****`.c`** ***file2*****`.c`** ***file3*****`.h`**

cscope provides command-line options with greater flexibility in specifying source files to be included in the cross-reference. When you invoke cscope with the -s option and any number of directory names (separated by commas):

```
% cscope -s dir1,dir2,dir3
```

cscope builds a cross-reference for all the source files in the specified directories as well as the current directory. To browse through all of the source files whose names are listed in *file* (file names separated by spaces, tabs, or new-lines), invoke cscope with the -i option and the name of the file containing the list:

```
% cscope -i file
```

If your source files are in a directory tree, use the following commands to browse through all of them:

```
% find .  -name '*.[chly]' -print | sort > file
% cscope -i file
```

If this option is selected, however, cscope ignores any other files appearing on the command-line.

The -I option can be used for cscope in the same way as the -I option to cc. See “Include Files” on page 64.

You can specify a cross-reference file other than the default cscope.out by invoking the -f option. This is useful for keeping separate symbol cross-reference files in the same directory. You may want to do this if two programs are in the same directory, but do not share all the same files:

```
% cscope -f admin.ref admin.c common.c aux.c libs.c
% cscope -f delta.ref delta.c common.c aux.c libs.c
```

In this example, the source files for two programs, admin and delta, are in the same directory, but the programs consist of different groups of files. By specifying different symbol cross-reference files when you invoke cscope for each set of source files, the cross-reference information for the two programs is kept separate.

You can use the -p*n* option to specify that cscope display the path name, or part of the path name, of a file when it lists the results of a search. The number you give to -p stands for the last *n* elements of the path name you want to be displayed. The default is 1, the name of the file itself. So if your current directory is home/common, the command:

```
% cscope -p2
```

causes `cscope` to display `common/file1.c`, `common/file2.c`, and so forth when it lists the results of a search.

If the program you want to browse contains a large number of source files, you can use the `-b` option, so that `cscope` stops after it has built a cross-reference; `cscope` does not display a menu of tasks. When you use `cscope -b` in a pipeline with the `batch`(1) command, `cscope` builds the cross-reference in the background:

```
% echo 'cscope -b' | batch
```

Once the cross-reference is built, and as long as you have not changed a source file or the list of source files in the meantime, you need only specify:

```
% cscope
```

for the cross-reference to be copied and the menu of tasks to be displayed in the normal way. You can use this sequence of commands when you want to continue working without having to wait for `cscope` to finish its initial processing.

The `-d` option instructs `cscope` not to update the symbol cross-reference. You can use it to save time if you are sure that no such changes have been made; `cscope` does not check the source files for changes.

Note – Use the `-d` option with care. If you specify `-d` under the erroneous impression that your source files have not been changed, `cscope` refers to an outdated symbol cross-reference in responding to your queries.

Check the `cscope`(1) man page for other command-line options.

View Paths

As we have seen, `cscope` searches for source files in the current directory by default. When the environment variable `VPATH` is set, `cscope` searches for source files in directories that comprise your view path. A view path is an ordered list of directories, each of which has the same directory structure below it.

For example, suppose you are part of a software project. There is an *official* set of source files in directories below `/fs1/ofc`. Each user has a home directory (`/usr/you`). If you make changes to the software system, you may have copies of just those files you are changing in `/usr/you/src/cmd/prog1`. The official versions of the entire program can be found in the directory `/fs1/ofc/src/cmd/prog1`.

Suppose you use `cscope` to browse through the three files that comprise `prog1`, namely, `f1.c`, `f2.c`, and `f3.c`. You would set `VPATH` to `/usr/you` and `/fs1/ofc` and export it, as in:

In a Bourne shell, type:

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

In a C shell, type:

```
% setenv VPATH /usr/you:/fs1/ofc
```

You then make your current directory `/usr/you/src/cmd/prog1`, and invoke `cscope`:

```
% cscope
```

The program locates all the files in the view path. In case duplicates are found, `cscope` uses the file whose parent directory appears earlier in `VPATH`. Thus, if `f2.c` is in your directory, and all three files are in the official directory, `cscope` examines `f2.c` from your directory, and `f1.c` and `f3.c` from the official directory.

The first directory in `VPATH` must be a prefix of the directory you will be working in, usually `$HOME`. Each colon-separated directory in `VPATH` must be absolute: it should begin at `/`.

`cscope` *and Editor Call Stacks*

`cscope` and editor calls can be stacked. That is, when `cscope` puts you in the editor to view a reference to a symbol and there is another reference of interest, you can invoke `cscope` again from within the editor to view the second reference without exiting the current invocation of either `cscope` or the editor. You can then back up by exiting the most recent invocation with the appropriate `cscope` and editor commands.

Examples

This section presents examples of how `cscope` can be used to perform three tasks: changing a constant to a preprocessor symbol, adding an argument to a function, and changing the value of a variable. The first example demonstrates the procedure for changing a text string, which differs slightly from the other tasks on the `cscope` menu. That is, once you have entered the text string to be changed, `cscope` prompts you for the new text, displays the lines containing the old text, and waits for you to specify which of these lines you want it to change.

Changing a Constant to a Preprocessor Symbol

Suppose you want to change a constant, `100`, to a preprocessor symbol, `MAXSIZE`. Select the sixth menu item, `Change this text string`, and enter `\100`. The `1` must be escaped with a backslash because it has a special meaning (item 1 on the menu) to `cscope`. Now press Return. `cscope` prompts you for the new text string. Type `MAXSIZE`.

`cscope` Function: Changing a Text String:

```
cscope     Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

`cscope` displays the lines containing the specified text string, and waits for you to select those in which you want the text to be changed.

`cscope` Function: Prompting for Lines to be Changed:

```
Change "100" to "MAXSIZE"

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;      /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

You know that the constant `100` in lines 1, 2, and 3 of the list (lines 4, 26, and 8 of the listed source files) should be changed to `MAXSIZE`. You also know that `0100` in `read.c` and `100.0` in `err.c` (lines 4 and 5 of the list) should not be changed. You select the lines you want changed with the following single-key commands:

Table 4-3 Commands for Selecting Lines to be Changed

1-9	Mark or unmark the line to be changed.
*	Mark or unmark all displayed lines to be changed.
Space	Display the next set of lines.
+	Display the next set of lines.
–	Display the previous set of lines.
a	Mark all lines to be changed.
^d	Change the marked lines and exit.
Esc	Exit without changing the marked lines.

In this case, enter 1, 2, and 3. The numbers you type are not printed on the screen. Instead, cscope marks each list item you want to be changed by printing a > (greater than) symbol after its line number in the list.

cscope Function: Marking Lines to be Changed:

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

Now type ^d to change the selected lines. cscope displays the lines that have been changed and prompts you to continue.

cscope Function: Displaying Changed Lines of Text:

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

Press the RETURN key to continue:
```

When you press Return in response to this prompt, cscope redraws the screen, restoring it to its state before you selected the lines to be changed.

The next step is to add the `#define` for the new symbol `MAXSIZE`. Because the header file in which the `#define` is to appear is not among the files whose lines are displayed, you must escape to the shell by typing `!`. The shell prompt appears at the bottom of the screen. Then enter the editor and add the `#define`.

`cscope` Function: Exiting to the Shell:

```
Text string:  100

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;          /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

To resume the `cscope` session, quit the editor and type `^d` to exit the shell.

Adding an Argument to a Function

Adding an argument to a function involves two steps: editing the function itself and adding the new argument to every place in the code where the function is called.

First, edit the function by using the second menu item, `Find this global definition`. Next, find out where the function is called. Use the fourth menu item, `Find functions calling this function`, to obtain a list of all the functions that call it. With this list, you can either invoke the editor for each line found by entering the list number of the line individually, or invoke

the editor for all the lines automatically by typing ^e. Using cscope to make this kind of change ensures that none of the functions you need to edit are overlooked.

Changing the Value of a Variable

At times, you may want to see how a proposed change affects your code.

Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item, Find this C symbol, to obtain a list of references that are affected. Then use the editor to examine each one. This step helps you predict the overall effects of your proposed change. Later, you can use cscope in the same way to verify that your changes have been made.

Command-Line Syntax for Editors

cscope invokes the vi editor by default. You can override the default setting by assigning your preferred editor to the EDITOR environment variable and exporting EDITOR, as described in “Step 1: Set Up the Environment” on page 84. However, cscope expects the editor it uses to have a command-line syntax of the form:

% **editor +*linenum filename***

as does vi. If the editor you want to use does not have this command-line syntax, you must write an interface between cscope and the editor.

Suppose you want to use ed. Because ed does not allow specification of a line number on the command-line, you cannot use it to view or edit files with cscope unless you write a shell script that contains the following line:

```
/usr/bin/ed $2
```

Let’s name the shell script myedit. Now set the value of EDITOR to your shell script and export EDITOR:

In a Bourne shell, type:

$ **EDITOR=myedit; export EDITOR**

In a C shell, type:

% **setenv EDITOR myedit**

When `cscope` invokes the editor for the list item you have specified, say, line 17 in `main.c`, it invoke your shell script with the command-line:

```
% myedit +17 main.c
```

`myedit` then discards the line number (`$1`) and calls `ed` correctly with the file name (`$2`). Of course, you are not moved automatically to line 17 of the file and must execute the appropriate `ed` commands to display and edit the line.

Unknown Terminal Type Error

If you see the error message:

```
Sorry, I don't know how to deal with your "term" terminal
```

your terminal may not be listed in the Terminal Information Utilities (`terminfo`) database that is currently loaded. Make sure you have assigned the correct value to `TERM`. If the message reappears, try reloading the Terminal Information Utilities.

If this message is displayed:

```
Sorry, I need to know a more specific terminal type than "unknown"
```

set and export the `TERM` variable as described in “Step 1: Set Up the Environment” on page 84.

SourceBrowser

The SourceBrowser is an interactive tool to aid programmers in the development and maintenance of software systems, particularly large ones. Because the SourceBrowser builds a database and uses it to respond to queries, once the database it built, the size of the code you are browsing has minimal impact on SourceBrowser’s speed.

SourceBrowser can find *all* occurrences of any symbol of your choice, including those found in header files. It can be used from either a command-line or window environment.

SourceBrowser uses a “what-you-see-is-what-you-browse” paradigm. The source code you manipulate is the same source code SourceBrowser uses in its searches, hence you can edit code from within the SourceBrowser itself.

SourceBrowser is designed to be used with multiple languages. In addition to C, it can be used with FORTRAN and C++, or with *(SPARC)* Pascal.

`lint` Source Code Checker — 5

This chapter describes `lint`, the C source code checker.

Overview of the `lint` Program

`lint` is a program that checks your C code for errors that may cause your C program not to compile or to execute with unexpected results. In many cases, `lint` warns you about incorrect, error-prone, or nonstandard code that the compiler does not necessarily flag.

`lint` issues every error and warning message produced by the C compiler. It also issues `lint`–specific warnings about potential bugs and portability problems. Many messages issued by `lint` can assist you in improving your program's effectiveness, including reducing its size and required memory.

`lint` is invoked on the command line, and can be invoked with multiple options. `lint` operates in two modes:

- *Basic*, which is the default for the `lint` program
- *Enhanced*, which includes everything done by basic `lint`, plus provides additional, detailed analysis of code

Locale of `lint` error messages is the same as for `cc`, see page 54.

Basic and Enhanced `lint` Functionality

In both basic and enhanced modes, `lint` compensates for separate and independent compilation in C by flagging inconsistencies in definition and use across files, including any libraries you have used. In a large project environment especially, where the same function may be used by different programmers in hundreds of separate modules of code, `lint` can help discover bugs that otherwise might be difficult to find. A function called with one less argument than expected, for example, looks at the stack for a value the call has never pushed, with results correct in one condition, incorrect in another, depending on whatever happens to be in memory at that stack location. By identifying dependencies like this one and dependencies on machine architecture as well, `lint` can improve the reliability of code run on your machine or someone else's.

In enhanced mode, `lint` provides more detailed reporting than in basic mode. In enhanced mode, `lint`'s capabilities include:

- Structure and flow analysis of the source program
- Constant propagations and constant expression evaluations
- Analysis of control flow and data flow
- Analysis of data types usage

In enhanced mode, `lint` can detect these problems:

- Unused `#include` directives, variables, and procedures
- Memory usage after its deallocation
- Unused assignments
- Usage of a variable value before its initialization
- Deallocation of nonallocated memory
- Usage of pointers when writing in constant data segments
- Nonequivalent macro redefinitions
- Unreached code
- Conformity of the usage of value types in unions
- Implicit casts of actual arguments.

Using `lint`

Invoke the basic `lint` program as follows:

```
% lint file1.c file2.c
```

Enhanced `lint` is invoked with the `-Nlevel` or `-Ncheck` option. For example, you can invoke enhanced `lint` as follows:

```
% lint -Nlevel=3 file1.c file2.c
```

`lint` examines code in two *passes*. In the first pass, `lint` checks for error conditions within C source files; in the second pass, it checks for inconsistencies across C source files. This process is invisible to the user unless `lint` is invoked with `-c`:

```
% lint -c file1.c file2.c
```

That command directs `lint` to execute the first pass only and collect information relevant to the second—about inconsistencies in definition and use across *file1*`.c` and *file2*`.c`—in intermediate files named *file1*`.ln` and *file2*`.ln`:

```
% ls
file1.c
file1.ln
file2.c
file2.ln
```

This way, the `-c` option to `lint` is analogous to the `-c` option to `cc`, which suppresses the link editing phase of compilation. Generally speaking, `lint`'s command-line syntax closely follows `cc`'s.

When the `.ln` files are `lint`ed:

```
% lint file1.ln file2.ln
```

the second pass is executed. `lint` processes any number of `.c` or `.ln` files in their command-line order. Thus,

```
% lint file1.ln file2.ln file3.c
```

directs `lint` to check *file3*`.c` for errors internal to it and all three files for consistency.

`lint` searches directories for included header files in the same order as `cc`. You can use the `-I` option to `lint` as you would the `-I` option to `cc`. See “Include Files” on page 64

You can specify multiple options to `lint` on the same command line. Options can be concatenated unless one of the options takes an argument or if the option has more than one letter:

```
% lint -cp -Idir1 -Idir2 file1.c file2.c
```

That command directs `lint` to:

- Execute the first pass only
- Perform additional portability checks
- Search the specified directories for included header files

`lint` has many options you can use to direct `lint` to perform certain tasks and report on certain conditions.

The `lint` Options

`lint` is a static analyzer. It cannot evaluate the runtime consequences of the dependencies it detects. Certain programs, for instance, may contain hundreds of unreachable `break` statements that are of little importance, but which `lint` flags nevertheless. This is one example where the `lint` command-line options and directives—special comments embedded in the source text—come in:

- You can invoke `lint` with the `-b` option to suppress all the error messages about unreachable `break` statements.
- You can precede any unreachable statement with the comment `/* NOTREACHED */` to suppress the diagnostic for that statement.

The `lint` options are listed below alphabetically. Several `lint` options relate to suppressing `lint` diagnostic messages. These options are also listed in Table 5-5, following the alphabetized options, along with the specific messages they suppress. The options for invoking enhanced `lint` begin with `-N`.

`lint` recognizes many `cc` command-line options, including `-A`, `-D`, `-E`, `-g`, `-H`, `-O`, `-P`, `-U`, `-Xa`, `-Xc`, `-Xs`, `-Xt`, and `-Y`, although `-g` and `-O` are ignored. Unrecognized options are warned about and ignored.

-#

Turns on verbose mode, showing each component as it is invoked.

`-###`

Shows each component as it is invoked, but does not actually execute it.

`-a`

Suppresses certain messages. Refer to Table 5-5.

`-b`

Suppresses certain messages. Refer to Table 5-5.

`-C` *filename*

Creates a `.ln` file with the file name specified. These `.ln` files are the product of `lint`'s first pass only. *filename* can be a complete path name.

`-c`

Creates a `.ln` file consisting of information relevant to `lint`'s second pass for every `.c` file named on the command line. The second pass is not executed.

`-dirout=`*dir*

Specifies the directory *dir* where the `lint` output files (`.ln` files) will be placed. This option affects the `-c` option.

`-err=warn`

Treats all warnings as errors. The result is that both errors and warnings cause `lint` to exit with a failure status.

`-errchk=`*l*

Check structural arguments passed by value; Check portability to environment for which the size of long integers and pointers is 64 bits.

l is a comma-separated list of checks that consists of one or more of the following:

structarg

Check structural arguments passed by value and report the cases when formal parameter type is not known.

longptr64

Check portability to environment for which the size of long integers and pointers is 64 bits and the size of plain integers is 32 bits. Check assignments of pointer expressions and long integer expressions to plain integers, even when explicit cast is used.

%all

Perform all of `errchk`'s checks.

%none

Perform none of `errchk`'s checks. This is the default.

no%structarg

Perform none of `errchk`'s *structarg* checks.

no%longptr64

Perform none of `errchk`'s longptr64 checks.

The values may be a comma separated list, for example `-errchk=longptr64,structarg`.

The default is `-errchk=%none`. Specifying `-errchk` is equivalent to specifying `-errchk=%all`.

-errfmt=*f*

Specifies the format of lint output. *f* can be one of the following: macro, simple, src, or tab.

Table 5-1 The -errfmt Values

Value	Meaning
macro	Displays the source code, the line number, and the place of the error, with macro unfolding
simple	Displays the line number and the place number, in brackets, of the error, for one-line (simple) diagnostic messages. Similar to the -s option, but includes error-position information
src	Displays the source code, the line number, and the place of the error (no macro unfolding)
tab	Displays in tabular format. This is the default.

The default is -errfmt=tab. Specifying -errfmt is equivalent to specifying -errfmt=tab.

If more than one format is specified, the last format specified is used, and lint warns about the unused formats.

-errhdr=*h*

Enables the reporting of certain messages for header files when used with -Ncheck. *h* is a comma-separated list that consists of one or more of the following: *dir*, no%*dir*, %all, %none, %user.

Table 5-2 The -errhdr Values

Value	Meaning
dir	Checks header files used in the directory *dir*
no%*dir*	Does not check header files used in the directory *dir*

Table 5-2 The `-errhdr` Values *(Continued)*

Value	Meaning
`%all`	Checks all used header files
`%none`	Does not check header files. This is the default.
`%user`	Checks all used user header files, that is, all header files except those in `/usr/include` and its subdirectories, as well as those supplied by the compiler

The default is `-errhdr=%none`. Specifying `-errhdr` is equivalent to specifying `-errhdr=%user`.

Examples:

```
% lint -errhdr=inc1 -errhdr=../inc2
```

checks used header files in directories `inc1` and `../inc2`.

```
% lint -errhdr=%all,no%../inc
```

checks all used header files except those in the directory `../inc`.

`-erroff=`*t*

Suppresses or enables `lint` error messages.

t is a comma-separated list that consists of one or more of the following: *tag*, `no%`*tag*, `%all`, `%none`.

Table 5-3 The `-erroff` Values

Value	Meaning
tag	Suppresses the message specified by this *tag*. You can display the tag for a message by using the `-errtags=yes` option.
`no%`*tag*	Enables the message specified by this *tag*
`%all`	Suppresses all messages
`%none`	Enables all messages. This is the default.

The default is `-erroff=%none`. Specifying `-erroff` is equivalent to specifying `-erroff=%all`.

Examples:

```
% lint -erroff=%all,no%E_ENUM_NEVER_DEF,no%E_STATIC_UNUSED
```

prints only the messages “enum never defined” and “static unused”, and suppresses other messages.

```
% lint -erroff=E_ENUM_NEVER_DEF,E_STATIC_UNUSED
```

suppresses only the messages “enum never defined” and “static unused”.

`-errtags=`*a*

Displays the message tag for each error message. *a* can be either `yes` or `no`. The default is `-errtags=no`. Specifying `-errtags` is equivalent to specifying `-errtags=yes`.

Works with all `-errfmt` options.

`-F`

Prints the path names as supplied on the command line rather than only their base names when referring to the `.c` files named on the command line.

`-fd`

Reports about old-style function definitions or declarations.

`-flagsrc=`*file*

Executes `lint` with options contained in the file *file*. Multiple options can be specified in *file*, one per line.

`-h`

Suppresses certain messages. Refer to Table 5-5.

`-I`*dir*

Searches the directory *dir* for included header files.

`-k`

Alter the behavior of `/* LINTED` [*message*] `*/` directives or `NOTE(LINTED(`*message*`))` annotations. Normally, `lint` suppresses warning messages for the code following these directives. Instead of suppressing the messages, `lint` prints an additional message containing the comment inside the directive or annotation.

`-L`*dir*

Searches for a `lint` library in the directory *dir* when used with `-l`.

`-l`*x*

Accesses the `lint` library `llib-l`*x*`.ln`.

`-m`

Suppresses certain messages. Refer to Table 5-5.

`-Ncheck=`*c*

Checks header files for corresponding declarations; checks macros. *c* is a comma-separated list of checks that consists of one or more of the following: `macro`, `extern`, `%all`, `%none`, `no%macro`, `no%extern`.

Table 5-4 The `-Ncheck` Values

Value	Meaning
`macro`	Checks for consistency of macro definitions across files
`extern`	Checks for one-to-one correspondence of declarations between source files and their associated header files (for example, for `file1.c` and `file1.h`). Ensure that there are neither extraneous nor missing `extern` declarations in a header file.
`%all`	Performs all of `-Ncheck`'s checks

Table 5-4 The `-Ncheck` Values *(Continued)*

Value	Meaning
`%none`	Performs none of `-Ncheck`'s checks. This is the default.
`no%macro`	Performs none of `-Ncheck`'s `macro` checks
`no%extern`	Performs none of `-Ncheck`'s `extern` checks

The default is `-Ncheck=%none`. Specifying `-Ncheck` is equivalent to specifying `-Ncheck=%all`.

Values may be combined with a comma, for example, `-Ncheck=extern,macro.`

Example:

```
% lint -Ncheck=%all,no%macro
```

performs all checks except macro checks.

`-Nlevel=`*n*

Specifies the level of analysis for reporting problems. This option allows you to control the amount of detected errors. The higher the level, the longer the verification time. *n* is a number: 1, 2, 3, or 4.

`-Nlevel=1`

Analyzes single procedures. Reports unconditional errors that occur on some program execution paths. Does not do global data and control flow analysis.

`-Nlevel=2`

The default. Analyzes the whole program, including global data and control flow. Reports unconditional errors that occur on some program execution paths.

`-Nlevel=3`

Analyzes the whole program, including constant propagation, cases when constants are used as actual arguments, as well as the analysis performed under `-Nlevel=2`.

`-Nlevel=4`

Analyzes the whole program, and reports conditional errors that could occur when certain program execution paths are used, as well as the analysis performed under `-Nlevel=3`.

The default is `-Nevel=2`. Specifying `-Nlevel` is equivalent to specifying `-Nlevel=2`.

-n

Suppresses checks for compatibility with the default `lint` standard C library.

-o*x*

Causes `lint` to create a `lint` library with the name `llib-l`*x*`.ln`. This library is created from all the `.ln` files that `lint` used in its second pass. The `-c` option nullifies any use of the `-o` option. To produce a `llib-l`*x*`.ln` without extraneous messages, you can use the `-x` option. The `-v` option is useful if the source file(s) for the `lint` library are just external interfaces. The `lint` library produced can be used later if `lint` is invoked with `-l`*x*.

By default, you create libraries in `lint`'s basic format. If you use `lint`'s enhanced mode, the library created will be in enhanced format, and can only be used in enhanced mode.

-p

Enables certain messages relating to portability issues.

-R*file*

Write a `.ln` file to *file*, for use by `cxref`(1). This option disables the enhanced mode, if it is switched on.

-s

Converts compound messages into simple ones.

-u

Suppresses certain messages. Refer to Table 5-5. This option is suitable for running `lint` on a subset of files of a larger program.

-V

Writes the product name and releases to standard error.

-v

Suppresses certain messages. Refer to Table 5-5.

-W*file*

Write a `.ln` file to *file*, for use by `cflow`(1). This option disables the enhanced mode, if it is switched on.

-x

Suppresses certain messages. Refer to Table 5-5.

-XCC=*a*

Accepts C++-style comments. In particular, `//` can be used to indicate the start of a comment. *a* can be either `yes` or `no`. The default is `-XCC=no`. Specifying `-XCC` is equivalent to specifying `-XCC=yes`.

-Xexplicitpar=*a*

(SPARC) Directs `lint` to recognize `#pragma MP` directives. *a* can be either `yes` or `no`. The default is `-Xexplicitpar=no`. Specifying `-Xexplicitpar` is equivalent to specifying `-Xexplicitpar=yes`.

-Xkeeptmp=*a*

Keeps temporary files created during linting instead of deleting them automatically. *a* can be either yes or no. The default is -Xkeeptmp=no. Specifying -Xkeeptmp is equivalent to specifying -Xkeeptmp=yes.

-Xtemp=*dir*

Sets the directory for temporary files to *dir*. Without this option, temporary files go into /tmp.

-Xtime=*a*

Reports the execution time for each lint pass. *a* can be either yes or no. The default is -Xtime=no. Specifying -Xtime is equivalent to specifying -Xtime=yes.

-Xtransition=*a*

Issues warnings for the differences between K&R C and Sun ANSI C. *a* can be either yes or no. The default is -Xtransition=no. Specifying -Xtransition is equivalent to specifying -Xtransition=yes.

-y

Treats every .c file named on the command line as if it begins with the directive /* LINTLIBRARY */ or the annotation NOTE(LINTLIBRARY). A lint library is normally created using the /* LINTLIBRARY */ directive or the NOTE(LINTLIBRARY) annotation.

lint *Messages*

Most of lint's messages are simple, one-line statements printed for each occurrence of the problem they diagnose. Errors detected in included files are reported multiple times by the compiler, but only once by lint, no matter how many times the file is included in other source files. Compound messages are issued for inconsistencies across files and, in a few cases, for problems within them as well. A single message describes every occurrence of the problem in the file or files being checked. When use of a lint filter (see "lint Libraries" on page 133) requires that a message be printed for each occurrence, compound diagnostics can be converted to the simple type by invoking lint with the -s option.

Lint's messages are written to stderr.

The Error and Warning Messages File, located in /opt/SUNWSPRO/READMEs/c_lint_messages, contains all the C compiler error and warning messages and all the lint program's messages. Many of the messages are self-explanatory. You can obtain a description of the messages and, in many cases, code examples, by searching the text file for a string from the message that was generated. See the section, "Error and Warning Messages File" on page xxv for details on using this file.

For a discussion of localization issues, see page 54.

Options to Suppress Messages

You can use several lint options to suppress lint diagnostic messages. Messages can be suppressed with the -erroff option, followed by one or more *tags*. These mnemonic tags can be displayed with the -errtags=yes option.

Table 5-5 lists the options that suppress `lint` messages.

Table 5-5 `lint` Options and Messages Suppressed

Option	Messages Suppressed
`-a`	`assignment causes implicit narrowing conversion` `conversion to larger integral type may sign-extend incorrectly`
`-b`	`statement not reached (unreachable break and empty statements)`

`-h`	`assignment operator "=" found where equality operator`
	`"==" was expected`
	`constant operand to op: "!"`
	`fallthrough on case statements`
	`pointer cast may result in improper alignment`
	`precedence confusion possible; parenthesize`
	`statement has no consequent: if`
	`statement has no consequent: else`

`-m`	`declared global, could be static`
`-erroff=`*t*	`One or more lint messages specified by` *tag*
`-u`	`name defined but never used` `name used but not defined`
`-v`	`arguments unused in function`
`-x`	`name declared but never used or defined`

`lint` *Message Formats*

`lint` can, with certain options, show precise source file lines with pointers to the line position where the error occurred. The option enabling this feature is `-errfmt=`*f*. Under this option, `lint` provides the following information:

- Source line(s) and position(s)
- Macro unfolding
- Error-prone stack(s)

For example, the following program, `Test1.c`, contains an error.

```
1 #include <string.h>
2 static void cpv(char *s, char* v, unsigned n)
3 { int i;
4   for (i=0; i<=n; i++)
5        *v++ = *s++;
6 }
7 void main(int argc, char* argv[])
8 {
9   if (argc != 0)
10     cpv(argv[0], argc, strlen(argv[0]));
11}
```

Using `lint` on `Test1.c` with the option:

`% lint -errfmt=src Test1.c`

produces the following output:

```
      |static void cpv(char *s, char* v, unsigned n)
      |            ^  line 2, Test1.c
      |
      |   cpv(argv[0], argc, strlen(argv[0]));
      |                    ^  line 10, Test1.c
warning: improper pointer/integer combination: arg #2
      |
      |static void cpv(char *s, char* v, unsigned n)
      |                                  ^  line 2, Test1.c
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
warning: modification using a pointer produced in a questionable way
    v defined at Test1.c(2)::Test1.c(5)
      call stack:
          main()                ,Test1.c(10)
          cpv()                 ,Test1.c(5)
```

The first warning indicates two source lines that are contradictory. The second warning shows the call stack, with the control flow leading to the error.

Another program, `Test2.c`, contains a different error:

```
1 #define AA(b) AR[b+l]
2 #define B(c,d) c+AA(d)
3
4 int x=0;
5
6 int AR[10]={1,2,3,4,5,6,77,88,99,0};
7
8 main()
9  {
10  int y=-5, z=5;
11  return B(y,z);
12 }
```

Using `lint` on `Test2.c` with the option:

`% lint -errfmt=macro Test2.c`

produces the following output, showing the steps of macro substitution:

```
       | return B(y,z);
       |        ^  line 11, Test2.c
       |
       |#define B(c,d) c+AA(d)
       |                 ^  line 2, Test2.c
       |
       |#define AA(b) AR[b+l]
       |                   ^  line 1, Test2.c
error: undefined symbol: l
```

lint Directives

Predefined Values

The following predefinitions are valid in all modes:

__sun

__unix

__lint

__SUNPRO_C=0x420

__`uname -s`_`uname -r` (example: __SunOS_5_4)

__RESTRICT (-Xa and -Xt modes only)

__sparc *(SPARC)*

__i386 *(Intel)*

__BUILTIN_VA_ARG_INCR

__SVR4

__LITTLE_ENDIAN *(PowerPC)*

__ppc *(PowerPC)*

These predefinitions are not valid in -Xc mode:

sun

unix

sparc *(SPARC)*

i386 *(Intel)*

lint

Directives

lint directives in the form of /*...*/ are supported for existing annotations, but will not be supported for future annotations. Directives in the form of source code annotations, NOTE(...), are recommended for all annotations.

Specify lint directives in the form of source code annotations by including the file note.h, for example:

```
#include <note.h>
```

Lint shares the Source Code Annotations scheme with several other tools. When you install the SunSoft ANSI C Compiler, you also automatically install the file /usr/lib/note/SUNW_SPRO-lint, which contains the names of all the annotations that LockLint understands. However, the SunSoft C source code checker, lint, also checks all the files in /usr/lib/note and /opt/SUNWspro/<***current-release***>/note for all valid annotations.

You may specify a location other than /usr/lib/note by setting the environment variable NOTEPATH, as in:

```
setenv NOTEPATH $NOTEPATH:other_location
```

Table 5-6 lists the `lint` directives along with their actions.

Table 5-6 `lint` Directives

Directive	Action
`NOTE(ALIGNMENT(`*fname,n*`))` where *n*`=1, 2, 4, 8, 16, 32, 64, 128`	Makes `lint` set the following function result alignment in *n* bytes. For example, `malloc()` is defined as returning a `char`* or `void`* when in fact it really returns pointers that are word, or even doubleword, aligned. Suppresses the following message: `improper alignment`
`NOTE(ARGSUSED(`*n*`))` `/*ARGSUSED`*n*`*/`	This directive acts like the `-v` option for the next function. Suppresses the following message: `argument unused in function` for every argument but the first *n* in the function definition it precedes. Default is 0. For the `NOTE` format, *n* must be specified.
`NOTE(ARGUNUSED(`*par_name*`[,` *par_name...*`]))`	Makes `lint` not check the mentioned arguments for usage (this option acts only for the next function). Suppresses the following message: `argument unused in function` for every argument listed in `NOTE` or directive.

`NOTE(CONSTCOND)`
`/*CONSTCOND*/`
	Suppresses complaints about constant operands for the conditional expression.
	Suppresses the following messages:
	`constant in conditional context`
	`constant operands to op: "!"`
	`logical expression always false: op "&&"`
	`logical expression always true: op "||"`
	for the constructs it precedes. Also `NOTE(CONSTANTCONDITION)` or `/* CONSTANTCONDITION */`.

`NOTE(EMPTY)`
`/*EMPTY*/`
	Suppresses complaints about a `null` statement consequent on an `if` statement. This directive should be placed after the test expression, and before the semicolon. This directive is supplied to support empty `if` statements when a valid `else` statement follows. It suppresses messages on an empty `else` consequent.
	Suppresses the following messages:
	`statement has no consequent: else`
	when inserted between the `else` and semicolon;
	`statement has no consequent: if`
	when inserted between the controlling expression of the `if` and semicolon.

Table 5-6 `lint` Directives *(Continued)*

Directive	Action
NOTE(FALLTHRU) /*FALLTHRU*/	Suppresses complaints about a fall through to a `case` or `default` labelled statement. This directive should be placed immediately preceding the label. Suppresses the following message: `fallthrough on case statement` for the `case` statement it precedes. Also NOTE(FALLTHROUGH) or /* FALLTHROUGH */.

NOTE(LINTED (*msg*))
/*LINTED [*msg*]*/
	Suppresses any intra-file warning except those dealing with unused variables or functions. This directive should be placed on the line immediately preceding where the `lint` warning occurred. The `-k` option alters the way in which `lint` handles this directive. Instead of suppressing messages, `lint` prints an additional message, if any, contained in the comments. This directive is useful in conjunction with the `-s` option for post-lint filtering.
	When `-k` is not invoked, suppresses every warning pertaining to an intra-file problem, except:
	`argument unused in function`
	`declarations unused in block`
	`set but not used in function`
	`static unused`
	`variable not used in function`
	for the line of code it precedes. *msg* is ignored.

NOTE(LINTLIBRARY) /*LINTLIBRARY*/	When `-o` is invoked, writes to a library `.ln` file only definitions in the `.c` file it heads. This directive suppresses complaints about unused functions and function arguments in this file.

NOTE(NOTREACHED)
/*NOTREACHED*/
	At appropriate points, stops comments about unreachable code. This comment is typically placed just after calls to functions such as `exit`(2).
	Suppresses the following messages:
	`statement not reached`
	for the unreached statements it precedes;
	`fallthrough on case statement`
	for the `case` it precedes that cannot be reached from the preceding `case`;
	`function falls off bottom without returning value`
	for the closing curly brace it precedes at the end of the function.

Table 5-6 `lint` Directives *(Continued)*

Directive	Action

NOTE(PRINTFLIKE(*n*))
NOTE(PRINTFLIKE*(fun_name,n))*
/*PRINTFLIKE*n**/
	Treats the *n*th argument of the function definition it precedes as a [fs]printf() format string and issues the following messages:
	`malformed format strings`
	for invalid conversion specifications in that argument, and function argument type inconsistent with format;
	`too few arguments for format`
	`too many arguments for format`
	for mismatches between the remaining arguments and the conversion specifications. `lint` issues these warnings by default for errors in the calls to [fs]printf() functions provided by the standard C library.
	For the NOTE format, *n* must be specified.

NOTE(PROTOLIB(*n*)) /*PROTOLIB*n**/	When *n* is 1 and NOTE(LINTLIBRARY) or /* LINTLIBRARY */ is used, writes to a library .ln file only function prototype declarations in the .c file it heads. The default is 0, which cancels the process. For the NOTE format, *n* must be specified.
NOTE(SCANFLIKE(*n*)) NOTE(SCANLIKE*(fun_name,n))* /*SCANFLIKE*n**/	Same as NOTE(PRINTFLIKE(*n*)) or /* PRINTFLIKE*n* */, except that the *n*th argument of the function definition is treated as a [fs]scanf() format string. By default, `lint` issues warnings for errors in the calls to [fs]scanf() functions provided by the standard C library. For the NOTE format, *n* must be specified.
NOTE(VARARGS(*n*)) NOTE(VARARGS(*fun_name,n*)) /*VARARGS*n**/	Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0. The use of the ellipsis (...) terminator in the definition is suggested in new or updated code. For the function whose definition it precedes, suppresses the following message: `functions called with variable number of arguments` for calls to the function with *n* or more arguments. For the NOTE format, *n* must be specified.

5

`lint` *Reference and Examples*

This section provides reference information on `lint`, including checks performed by `lint`, `lint` libraries, and `lint` filters.

Checks Performed by `lint`

`lint`-specific diagnostics are issued for three broad categories of conditions: inconsistent use, nonportable code, and questionable constructs. In this section, we review examples of `lint`'s behavior in each of these areas, and suggest possible responses to the issues they raise.

Consistency Checks

Inconsistent use of variables, arguments, and functions is checked within files as well as across them. Generally speaking, the same checks are performed for prototype uses, declarations, and parameters as `lint` checks for old-style functions. If your program does not use function prototypes, `lint` checks the number and types of parameters in each call to a function more strictly than the compiler. `lint` also identifies mismatches of conversion specifications and arguments in `[fs]printf()` and `[fs]scanf()` control strings.

Examples:

- Within files, `lint` flags non-`void` functions that "fall off the bottom" without returning a value to the invoking function. In the past, programmers often indicated that a function was not meant to return a value by omitting the return type: `fun() {}`. That convention means nothing to the compiler, which regards `fun()` as having the return type `int`. Declare the function with the return type `void` to eliminate the problem.
- Across files, `lint` detects cases where a non-`void` function does not return a value, yet is used for its value in an expression—and the opposite problem, a function returning a value that is sometimes or always ignored in subsequent calls. When the value is *always* ignored, it may indicate an inefficiency in the function definition. When it is *sometimes* ignored, it's probably bad style (typically, not testing for error conditions). If you need not check the return values of string functions like `strcat()`, `strcpy()`, and `sprintf()`, or output functions like `printf()` and `putchar()`, cast the offending calls to `void`.

- `lint` identifies variables or functions that are declared but not used or defined; used, but not defined; or defined, but not used. When `lint` is applied to some, but not all files of a collection to be loaded together, it produces error messages about functions and variables that are:
  - Declared in those files, but defined or used elsewhere
  - Used in those files, but defined elsewhere
  - Defined in those files, but used elsewhere

  Invoke the `-x` option to suppress the first complaint, `-u` to suppress the latter two.

Portability Checks

Some nonportable code is flagged by `lint` in its default behavior, and a few more cases are diagnosed when `lint` is invoked with `-p` or `-Xc`. The latter causes `lint` to check for constructs that do not conform to the ANSI C standard. For the messages issued under `-p` and `-Xc`, see “lint Libraries” on page 133.

Examples:

- In some C language implementations, character variables that are not explicitly declared `signed` or `unsigned` are treated as signed quantities with a range typically from –128 to 127. In other implementations, they are treated as nonnegative quantities with a range typically from 0 to 255. So the test:

```
char c;
c = getchar();
if (c == EOF) ...
```

  where `EOF` has the value –1, always fails on machines where character variables take on nonnegative values. `lint` invoked with `-p` checks all comparisons that imply a *plain* `char` may have a negative value. However, declaring `c` a `signed char` in the above example eliminates the diagnostic, not the problem. That's because `getchar()` must return all possible characters and a distinct `EOF` value, so a `char` cannot store its value. We cite this example, perhaps the most common one arising from implementation-defined sign-extension, to show how a thoughtful application of `lint`'s portability option can help you discover bugs not related to portability. In any case, declare `c` as an `int`.

- A similar issue arises with bit-fields. When constant values are assigned to bit-fields, the field may be too small to hold the value. On a machine that treats bit-fields of type `int` as unsigned quantities, the values allowed for `int x:3` range from 0 to 7, whereas on machines that treat them as signed quantities, they range from –4 to 3. However, a three-bit field declared type `int` cannot hold the value 4 on the latter machines. `lint` invoked with `-p` flags all bit-field types other than `unsigned int` or `signed int`. These are the only *portable* bit-field types. The compiler supports `int`, `char`, `short`, and `long` bit-field types that may be `unsigned`, `signed`, or *plain*. It also supports the `enum` bit-field type.
- Bugs can arise when a larger-sized type is assigned to a smaller-sized type. If significant bits are truncated, accuracy is lost:

```
short s;
long l;
s = l;
```

  `lint` flags all such assignments by default; the diagnostic can be suppressed by invoking the `-a` option. Bear in mind that you may be suppressing other diagnostics when you invoke `lint` with this or any other option. Check the list in "lint Libraries" on page 133 for the options that suppress more than one diagnostic.

- A cast of a pointer to one object type to a pointer to an object type with stricter alignment requirements may not be portable. `lint` flags:

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

  because, on most machines, an `int` cannot start on an arbitrary byte boundary, whereas a `char` can. You can suppress the diagnostic by invoking `lint` with `-h`, although, again, you may be disabling other messages. Better still, eliminate the problem by using the generic pointer `void *`.

- ANSI C leaves the order of evaluation of complicated expressions undefined. That is, when function calls, nested assignment statements, or the increment and decrement operators cause side effects when a variable is

changed as a by-product of the evaluation of an expression, the order in which the side effects take place is highly machine-dependent. By default, `lint` flags any variable changed by a side effect and used elsewhere in the same expression:

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

In this example, the value of `a[1]` may be 1 if one compiler is used, 2 if another. The bitwise logical operator `&` can give rise to this diagnostic when it is mistakenly used in place of the logical operator `&&`:

```
if ((c = getchar()) != EOF & c != '0')
```

Questionable Constructs

`lint` flags a miscellany of legal constructs that may not represent what the programmer intended. Examples:

- An `unsigned` variable always has a nonnegative value. So the test:

```
unsigned x;
if (x < 0) ...
```

always fails. The test:

```
unsigned x;
if (x > 0) ...
```

is equivalent to:

```
if (x != 0) ...
```

This may not be the intended action. `lint` flags questionable comparisons of `unsigned` variables with negative constants or `0`. To compare an `unsigned` variable to the bit pattern of a negative number, cast it to `unsigned`:

```
if (u == (unsigned) -1) ...
```

Or use the `U` suffix:

```
if (u == -1U) ...
```

- `lint` flags expressions without side effects that are used in a context where side effects are expected—that is, where the expression may not represent what the programmer intends. It issues an additional warning whenever the equality operator is found where the assignment operator is expected—that is, where a side effect is expected:

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- `lint` cautions you to parenthesize expressions that mix both the logical and bitwise operators (specifically, `&`, `|`, `^`, `<<`, `>>`), where misunderstanding of operator precedence may lead to incorrect results. Because the precedence of bitwise `&`, for example, falls below logical `==`, the expression:

```
if (x & a == 0) ...
```

is evaluated as:

```
if (x & (a == 0)) ...
```

which is most likely not what you intended. Invoking `lint` with `-h` disables the diagnostic.

lint *Libraries*

You can use lint libraries to check your program for compatibility with the library functions you have called in it—the declaration of the function return type, the number and types of arguments the function expects, and so on. The standard lint libraries correspond to libraries supplied by the C compilation system, and generally are stored in a standard place on your system. By convention, lint libraries have names of the form llib-l*x*.ln.

The lint standard C library, llib-lc.ln , is appended to the lint command line by default; checks for compatibility with it can be suppressed by invoking the -n option. Other lint libraries are accessed as arguments to -l. That is:

```
% lint -lx file1.c file2.c
```

directs lint to check the usage of functions and variables in file1.c and file2.c for compatibility with the lint library llib-l*x*.ln. The library file, which consists only of definitions, is processed exactly as are ordinary source files and ordinary .ln files, except that functions and variables used inconsistently in the library file, or defined in the library file but not used in the source files, elicit no complaints.

To create your own lint library, insert the directive NOTE(LINTLIBRARY) at the head of a C source file, then invoke lint for that file with the -o option and the library name given to -l:

```
% lint -ox file1.c file2.c
```

causes only definitions in the source files headed by NOTE(LINTLIBRARY) to be written to the file llib-l*x*.ln. (Note the analogy of lint -o to cc -o.) A library can be created from a file of function prototype declarations in the same way, except that both NOTE(LINTLIBRARY) and NOTE(PROTOLIB(*n*))must be inserted at the head of the declarations file. If *n* is 1, prototype declarations are written to a library .ln file just as are old-style definitions. If *n* is 0, the default, the process is cancelled. Invoking lint with -y is another way of creating a lint library. The command line:

```
% lint -y -ox file1.c file2.c
```

causes each source file named on that line to be treated as if it begins with NOTE(LINTLIBRARY), and only its definitions to be written to llib-l*x*.ln.

By default, `lint` searches for `lint` libraries in the standard place. To direct `lint` to search for a `lint` library in a directory other than the standard place, specify the path of the directory with the `-L` option:

```
% lint -Ldir -lx file1.c file2.c
```

In enhanced mode, `lint` produces `.ln` files which store additional information than `.ln` files produced in basic mode. In enhanced mode, `lint` can read and understand all `.ln` files generated by either basic or enhanced `lint` modes. In basic mode, `lint` can read and understand `.ln` files generated only using basic `lint` mode.

By default, `lint` uses libraries from the `/usr/lib` directory. These libraries are in the basic `lint` format, that is, libraries shipped with C 3.0.1 and below. You can run a `makefile` once, and create enhanced `lint` libraries in a new format, which will enable enhanced `lint` to work more effectively. To run the `makefile` and create the new libraries, enter the command:

```
% cd /opt/SUNWspro/SC4.2/src/lintlib; make
```

where `/opt/SUNWspro/SC4.2` is the installation directory. After the `makefile` is run, `lint` will use the new libraries in enhanced mode, instead of the libraries in the `/usr/lib` directory.

The specified directory is searched before the standard place.

`lint` *Filters*

A `lint` filter is a project-specific post-processor that typically uses an `awk` script or similar program to read the output of `lint` and discard messages that your project has deemed as *not* identifying real problems—string functions, for instance, returning values that are sometimes or always ignored. `lint` filters generate customized diagnostic reports when `lint` options and directives do not provide sufficient control over output.

Two options to `lint` are particularly useful in developing a filter:

- Invoking `lint` with `-s` causes compound diagnostics to be converted into simple, one-line messages issued for each occurrence of the problem diagnosed. The easily parsed message format is suitable for analysis by an `awk` script.

- Invoking `lint` with `-k` causes certain comments you have written in the source file to be printed in output, and can be useful both in documenting project decisions and specifying the post-processor's behavior. In the latter instance, if the comment identifies an expected `lint` message, and the reported message is the same, the message can be filtered out. To use `-k`, insert on the line preceding the code you wish to comment the `NOTE(LINTED(`*msg*`))`directive, where *msg* refers to the comment to be printed when `lint` is invoked with `-k`.

  Refer to the list of directives in Table 5-6 for an explanation of what `lint` does when `-k` is *not* invoked for a file containing `NOTE(LINTED(`*msg*`))`.

ANSI C Data Representations

This appendix describes how ANSI C represents data in storage and the mechanisms for passing arguments to functions. It is intended as a guide to programmers who want to write or use modules in languages other than C and have those modules interface to C code.

Storage Allocation

Table A-1 shows the data types and how they are represented.

Table A-1 Storage Allocation for Data Types

Data Type	**Internal Representation**
`char` elements	A single 8-bit byte aligned on a byte boundary.
`short` integers	Halfword (two bytes or 16 bits), aligned on a two-byte boundary
`int` and `long`	32 bits (four bytes or one word), aligned on a four-byte boundary
`long long`[1]	*(SPARC) (PowerPC)* 64 bits (eight bytes or two words), aligned on an eight-byte boundary *(Intel)* 64 bits (eight bytes or two words), aligned on a four-byte boundary

Table A-1 Storage Allocation for Data Types *(Continued)*

Data Type	Internal Representation
`float`	32 bits (four bytes or one word), aligned on a four-byte boundary. A `float` has a `sign` bit, 8-bit exponent, and 23-bit fraction.
`double`	64 bits (eight bytes or two words), aligned on an eight-byte boundary *(SPARC)* *(PowerPC)* or aligned on a four-byte boundary *(Intel)*. A `double` element has a `sign` bit, an 11-bit exponent and a 52-bit fraction.
`long double`	*(SPARC)* 128 bits (16 bytes or four words), aligned on an eight-byte boundary. A `long double` element has a `sign` bit, a 15-bit exponent and a 112-bit fraction. *(PowerPC)* 128 bits (16 bytes or four words), aligned on a16-byte boundary. A `long double` element has a `sign` bit, a 15-bit exponent and a 112-bit fraction. *(Intel)* 96 bits (12 bytes or three words) aligned on a four-byte boundary. A `long double` element has a `sign` bit, a 16-bit exponent, and a 64-bit fraction. 16 bits are unused.

1. `long long` is not available in `-Xc` mode.

Data Representations

Bit numberings of any given data element depend on the architecture in use: SPARCstation™ machines use bit 0 as the least significant bit, with byte 0 being the most significant byte. The tables in this section describe the various representations.

Integer Representations

Integer types used in ANSI C are `short`, `int`, `long`, and `long long`:

Table A-2 Representation of `short`

Bits	Content
8 - 15	Byte 0 *(SPARC)* Byte 1 *(Intel)* *(PowerPC)*
0 - 7	Byte 1 *(SPARC)* Byte 0 *(Intel)* *(PowerPC)*

Table A-3 Representation of `int` and `long`

Bits	Content
24 - 31	Byte 0 *(SPARC)* Byte 3 *(Intel) (PowerPC)*
16 - 23	Byte 1 *(SPARC)* Byte 2 *(Intel) (PowerPC)*
8 - 15	Byte 2 *(SPARC)* Byte 1 *(Intel) (PowerPC)*
0 - 7	Byte 3 *(SPARC)* Byte 0 *(Intel) (PowerPC)*

Table A-4 Representation of `long long`[1]

Bits	Content
56 - 63	Byte 0 *(SPARC)* Byte 7 *(Intel) (PowerPC)*
48 - 55	Byte 1 *(SPARC)* Byte 6 *(Intel) (PowerPC)*
40 - 47	Byte 2 *(SPARC)* Byte 5 *(Intel) (PowerPC)*
32 - 39	Byte 3 *(SPARC)* Byte 4 *(Intel) (PowerPC)*
24 - 31	Byte 4 *(SPARC)* Byte 3 *(Intel) (PowerPC)*
16 - 23	Byte 5 *(SPARC)* Byte 2 *(Intel) (PowerPC)*
8 - 15	Byte 6 *(SPARC)* Byte 1 *(Intel) (PowerPC)*
0 - 7	Byte 7 *(SPARC)* Byte 0 *(Intel) (PowerPC)*

1. `long long` is not available in `-Xc` mode.

Floating-Point Representations

`float`, `double`, and `long double` data elements are represented according to the ANSI IEEE 754-1985 standard. The representation is:

$(-1)^s 2^{(e - bias)} \times j.f$

where:

- s = `sign`
- e = biased exponent
- j is the leading bit, determined by the value of e. In the case of `long double` *(Intel)*, the leading bit is explicit; in all other cases, it is implicit.
- f = fraction
- u means that the bit can be either 0 or 1.

The following tables show the position of the bits.

Table A-5 `float` Representation

Bits	Name
31	Sign
23 - 30	Exponent
0 - 22	Fraction

Table A-6 `double` Representation

Bits	Name
63	Sign
52 - 62	Exponent
0 - 51	Fraction

Table A-7 `long double` Representation *(SPARC) (PowerPC)*

Bits	Name
127	Sign
112 - 126	Exponent
0 - 111	Fraction

Table A-8 `long double` Representation *(Intel)*

Bits	Name
80 - 95	Unused
79	Sign
64 - 78	Exponent
63	Leading bit
0 - 62	Fraction

For further information, refer to the *Numerical Computation Guide*.

Exceptional Values

`float` and `double` numbers are said to contain a "hidden," or implied, bit, providing for one more bit of precision than would otherwise be the case. In the case of `long double`, the leading bit is implicit *(SPARC) (PowerPC)* or explicit *(Intel)*; this bit is 1 for normal numbers, and 0 for subnormal numbers.

Table A-9 `float` Representations

normal number (0<e<255):	$(-1)^{Sign} 2^{(exponent - 127)} 1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign} 2^{(-126)} 0.f$
zero (e=0, f=0):	$(-1)^{Sign} 0.0$
signaling NaN	s=u, e=255(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=255(max); f=.1uuu-uu
Infinity	s=u, e=255(max); f=.0000-00 (all zeroes)

Table A-10 `double` Representations

normal number (0<e<2047):	$(-1)^{Sign}2^{(exponent - 1023)}1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign}2^{(-1022)}0.f$
zero (e=0, f=0):	$(-1)^{Sign}0.0$
signaling NaN	s=u, e=2047(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=2047(max); f=.1uuu-uu
Infinity	s=u, e=2047(max); f=.0000-00 (all zeroes)

Table A-11 `long double` Representations

normal number (0<e<32767):	$(-1)^{Sign}2^{(exponent - 16383)}1.f$
subnormal number (e=0, f!=0):	$(-1)^{Sign}2^{(-16382)}0.f$
zero (e=0, f=0):	$(-1)^{Sign}0.0$
signaling NaN	s=u, e=32767(max); f=.0uuu-uu; at least one bit must be nonzero
quiet NaN	s=u, e=32767(max); f=.1uuu-uu
Infinity	s=u, e=32767(max); f=.0000-00 (all zeroes)

Hexadecimal Representation of Selected Numbers

The following tables show the hexadecimal representations.

Table A-12 Hexadecimal Representation of Selected Numbers *(SPARC) (PowerPC)*

Value	`float`	`double`	`long double`
+0 -0	00000000 80000000	0000000000000000 8000000000000000	00000000000000000000000000000000 80000000000000000000000000000000
+1.0 -1.0	3F800000 BF800000	3FF0000000000000 BFF0000000000000	3FFF0000000000000000000000000000 BFFF0000000000000000000000000000
+2.0 +3.0	40000000 40400000	4000000000000000 4008000000000000	40000000000000000000000000000000 40080000000000000000000000000000
+Infinity -Infinity	7F800000 FF800000	7FF0000000000000 FFF0000000000000	7FFF0000000000000000000000000000 FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF F

Table A-13 Hexadecimal Representation of Selected Numbers *(Intel)*

Value	`float`	`double`	`long double`
+0 -0	00000000 80000000	0000000000000000 0000000080000000	00000000000000000000 80000000000000000000
+1.0 -1.0	3F800000 BF800000	000000003FF00000 00000000BFF00000	3FFF8000000000000000 BFFF8000000000000000
+2.0 +3.0	40000000 40400000	0000000040000000 0000000040080000	40008000000000000000 4000C000000000000000
+Infinity -Infinity	7F800000 FF800000	000000007FF00000 00000000FFF00000	7FFF8000000000000000 FFFF8000000000000000
NaN	7FBFFFFF	FFFFFFFF7FF7FFFF	7FFFBFFFFFFFFFFFFFFF

For further information, refer to the *Numerical Computation Guide*.

Pointer Representation

A pointer in C occupies four bytes. The `NULL` value pointer is equal to zero.

Array Storage

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

C arrays are stored in row-major order; the last subscript in a multidimensional array varies the fastest.

String data types are simply arrays of `char` elements.

Table A-14 Automatic Array Types and Storage

Type	Maximum Number of Elements
`char`	268435455
`short`	134217727
`int`	67108863
`long`	67108863
`float`	67108863
`double`	33554431
`long double`	1677215 *(SPARC)* *(PowerPC)* 22369621 *(Intel)*
`long long`[1]	33554431

1. Not valid in `-Xc` mode

Static and global arrays can accommodate many more elements.

Arithmetic Operations on Exceptional Values

This section describes the results derived from applying the basic arithmetic operations to combinations of exceptional and ordinary floating-point values. The information that follows assumes that no traps or any other exception actions are taken.

The following tables explain the abbreviations:

Table A-15 Abbreviation Usage

Abbreviation	Meaning
Num	Subnormal or normal number
Inf	Infinity (positive or negative)
NaN	Not a number
Uno	Unordered

The tables that follow describe the types of values that result from arithmetic operations performed with combinations of different types of operands.

Table A-16 Addition and Subtraction Results

	Right Operand			
Left Operand	**0**	**Num**	**Inf**	**NaN**
0	0	Num	Inf	NaN
Num	Num	See Note	Inf	NaN
Inf	Inf	Inf	See Note	NaN
NaN	NaN	NaN	NaN	NaN

Note – Num + Num could be Inf, rather than Num, when the result is too large (overflow). Inf + Inf = NaN when the infinities are of opposite `sign`.

Table A-17 Multiplication Results

	Right Operand			
Left Operand	**0**	**Num**	**Inf**	**NaN**
0	0	0	NaN	NaN
Num	0	Num	Inf	NaN
Inf	NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN

Table A-18 Division Results

Left Operand	Right Operand			
	0	**Num**	**Inf**	**NaN**
0	NaN	0	0	NaN
Num	Inf	Num	0	NaN
Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN

Table A-19 Comparison Results

Left Operand	Right Operand			
	0	**+Num**	**+Inf**	**NaN**
0	=	<	<	Uno
+Num	>	The result of the comparison	<	Uno
+Inf	>	>	=	Uno
NaN	Uno	Uno	Uno	Uno

Note – NaN compared with NaN is unordered, and results in inequality. +0 compares equal to -0.

Argument-Passing Mechanism

This section describes how arguments are passed in ANSI C.

All arguments to C functions are passed by value.

Actual arguments are passed in the reverse order from which they are declared in a function declaration.

Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then placed in a register or pushed onto the stack.

(SPARC)

Functions return `integer` results in register `%o0`, `float` results in register `%f0`, and `double` results in registers `%f0` and `%f1`.

`long long`[1] integers are ***passed*** in registers with the higher word order in `%oN`, and the lower order word in `%o(N+1)`. In-register results are ***returned*** in `%i0` and `%i1`, with similar ordering.

All arguments, except `doubles` and `long doubles`, are passed as four-byte values. A `double` is passed as an eight-byte value. The first six four-byte values (`double` counts as 8) are passed in registers `%o0` through `%o5`. The rest are passed onto the stack. Structures are passed by making a copy of the structure and passing a pointer to the copy. A `long double` is passed in the same manner as a structure.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack. Registers described are as seen by the caller.

(Intel)

Functions return `integer` results in register `%eax`.

`long long` results are ***returned*** in registers `%edx` and `%eax`. Functions return `float`, `double`, and `long double` results in register `%st(0)`.

All arguments, except `structs`, `unions`, `long longs`, `doubles` and `long doubles`, are passed as four-byte values; a `long long` is passed as an eight-byte value, a `double` is passed as an eight-byte value, and a `long double` is passed as a 12-byte value.

`structs` and `unions` are copied onto the stack. The size is rounded up to a multiple of four bytes. Functions returning `structs` and `unions` are passed a hidden first argument, pointing to the location into which the returned `struct` or `union` is stored.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack, except for the extra argument for `struct` and `union` returns that is popped by the called function.

1. Not available in `-Xc` mode.

(PowerPC)

Functions return integer results in register `%r3`. `Float` and `double` results are returned in `%f1`. For `float` results, the value returned is rounded to `float` precision. `Long long` and structure or union result whose size is 8 bytes or less are returned in registers `%r3` and `%r4`, with `%r3` containing the low-addressed word of the structure as stored in memory. For other structures or unions and `long double` results, the caller passes a pointer to a storage area into which the result is copied in register `%r3` as an implicit first argument.

Arguments are passed in integer registers `%r3` through `%r10`, floating-point registers `%f1` through `%f8` and a parameter area on the stack. All arguments except `doubles`, `long doubles`, and `long long` are passed as four-byte values.

In the description of the argument-passing algorith below, gr is the number of the next integer register to be used for argument passing, fr is the number of the next floating point register, and starg is a pointer to the next word in the parameter area.

INITIALIZE:
Set fr=1, gr=3, and starg to the start of the parameter area.

SCAN:
If there are no more arguments, terminate. Otherwise, select one of the following, depending on the type of the next argument:

DOUBLE_OR_FLOAT:
If fr>8 (that is, there are no more available floating-point registers), go to OTHER. Otherwise, load the argument value into floating register fr, increment fr, and go to SCAN.

SIMPLE_ARG:
A SIMPLE_ARG is one of the following:
1. One of the simple integer types no more than 32 bits wide (`char`, `short`, `int`, `long`, `enum`), or
2. A pointer to an object of any type, or
3. A structure, union, or `long double`, is passed as a pointer to a copy of the object made by the caller.

If gr>10, go to OTHER. Otherwise, load the argument value into general register gr, set gr to gr+1, and go to SCAN. Values shorter than 32 bits are sign extended or zero extended depending on whether they are `signed` or `unsigned`.

LONG_LONG:
If gr>7, go to OTHER. If gr is even, set gr to gr+1. Load the lower-addressed word of the `long long` into gr and the higher-addressed word into gr+1, increment gr by 2 and to SCAN.

OTHER:
Arguments not otherwise handled above are passed in the parameter words of the caller's stack frame. Integer values shorter than 32 bits are (conceptually) sign or zero extended to 32 bits and considered to have 4-byte size and alignment; otherwise the size, sz, of the argument is as determined by the sizeof operator. Round starg up to a multiple of the alignment requirement of the argument and copy the argument byte-for-byte, beginning with its lowest addressed byte, into starg, ..., starg+sz-1. Set starg to starg+sz, then go to SCAN.

Implementation-Defined Behavior

The American National Standard for Programming Language--C, ANSI/ISO 9899-1990 defines the behavior of ANSI-conformant C. However, this standard leaves a number of issues as "implementation-defined," that is, as varying from compiler to compiler.

This chapter details these areas. They can be readily compared to the ANSI standard itself:

- Each issue uses the same section text as found in the ANSI standard.
- Each issue is preceded by its corresponding section number in the ANSI standard.

Translation

(2.1.1.3) Identification of diagnostics:

Error messages have the following format:

filename, `line` *line number*: *message*

Warning messages have the following format:

filename, `line` *line number*: *warning message*

Where:

- *filename* is the name of the file containing the error or warning
- *line number* is the number of the line on which the error or warning is found

- *message* is the diagnostic message

Environment

(2.1.2.2.1) Semantics of arguments to `main`:

```
int main (int argc, char *argv[])
{
....
}
```

`argc` is the number of command-line arguments with which the program is invoked with. After any shell expansion, `argc` is always equal to at least `1`, the name of the program.

`argv` is an array of pointers to the command-line arguments.

(2.1.2.3) What constitutes an interactive device:

An interactive device is one for which the system library call `isatty()` returns a nonzero value.

Identifiers

(3.1.2) The number of significant initial characters (beyond 31) in an identifier without external linkage:

The first 1,023 characters are significant. Identifiers are case-sensitive.

(3.1.2) The number of significant initial characters (beyond 6) in an identifier with external linkage:

The first 1,023 characters are significant. Identifiers are case-sensitive.

Characters

(2.2.1) The members of the source and execution character sets, except as explicitly specified in the Standard:

Both sets are identical to the ASCII character sets, plus locale-specific extensions.

(2.2.1.2) The shift states used for the encoding of multibyte characters:

There are no shift states.

(2.2.4.2.1) The number of bits in a character in the execution character set:

There are 8 bits in a character for the ASCII portion; locale-specific multiple of 8 bits for locale-specific extended portion.

(3.1.3.4) The mapping of members of the source character set (in character and string literals) to members of the execution character set:

Mapping is identical between source and execution characters.

(3.1.3.4) The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant:

It is the numerical value of the rightmost character. For example, `'\q'` equals `'q'`. A warning is emitted if such an escape sequence occurs.

(3.1.3.4) The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character:

A multiple-character constant that is not an escape sequence has a value derived from the numeric values of each character.

(3.1.3.4) The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant:

The valid locale specified by `LC_ALL`, `LC_CTYPE`, or `LANG` environment variable.

(3.2.1.1) Does a plain* `char` *have the same range of values as* `signed char` *or* `unsigned char`*:

A `char` is treated as a `signed char` *(SPARC) (Intel)* .

A `char` is treated as an `unsigned char` *(PowerPC)*.

Integers

(3.1.2.5) The representations and sets of values of the various types of integers:

Table B-1 Representations and Sets of Values of Integers

Integer	Bits	Minimum	Maximum
`char` *(SPARC) (Intel)*	8	-128	127
`char` *(PowerPC)*	8	0	255
`signed char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`	32	-2147483648	2147483647
`signed int`	32	-2147483648	2147483647
`unsigned int`	32	0	4294967295
`long`	32	-2147483648	2147483647
`signed long`	32	-2147483648	2147483647
`unsigned long`	32	0	4294967295
`long long`[1]	64	-9223372036854775808	9223372036854775807
`signed long long`[1]	64	-9223372036854775808	9223372036854775807
`unsigned long long`[1]	64	0	18446744073709551615

1. Not valid in `-Xc` mode

(3.2.1.2) The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

When an integer is converted to a shorter `signed` integer, the low order bits are copied from the longer integer to the shorter `signed` integer. The result may be negative.

When an unsigned integer is converted to a `signed` integer of equal size, the low order bits are copied from the `unsigned` integer to the `signed` integer. The result may be negative.

(3.3) The results of bitwise operations on signed integers:

The result of a bitwise operation applied to a `signed` type is the bitwise operation of the operands, including the `sign` bit. Thus, each bit in the result is set if—and only if—each of the corresponding bits in both of the operands is set.

(3.3.5) The sign of the remainder on integer division:

The result is the same sign as the dividend; thus, the remainder of `-23/4` is -3.

(3.3.7) The result of a right shift of a negative-valued signed integral type:

The result of a right shift is a `signed` right shift.

Floating-Point

(3.1.2.5) The representations and sets of values of the various types of floating-point numbers:

Table B-2 Values of Floating-Point Numbers

float	
Bits	32
Min	1.17549435E-38
Max	3.40282347E+38
Epsilon	1.19209290E-07
double	
Bits	64
Min	2.2250738585072014E-308
Max	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16
long double	
Bits	128 *(SPARC)(PowerPC)* 80 *(Intel)*
Min	3.362103143112093506262677817321752603E-4932 *(SPARC) (PowerPC)* 3.3621031431120935062627E-4932 *(Intel)*
Max	1.189731495357231765085759326628007016E+4932 *(SPARC) (PowerPC)* 1.18973149535723176502l3E4932 *(Intel)*
Epsilon	1.925929944387235853055977942584927319E-34 *(SPARC) (PowerPC)* 1.0842021724855044340075E-19 *(Intel)*

(3.2.1.3) The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value:

Numbers are rounded to the nearest value that can be represented.

(3.2.1.4) The direction of truncation or rounding when a floating- point number is converted to a narrower floating-point number:

Numbers are rounded to the nearest value that can be represented.

Arrays and Pointers

(3.3.3.4, 4.1.1) The type of integer required to hold the maximum size of an array; that is, the type of the* `sizeof` *operator,* `size_t`*:

`unsigned int` as defined in `stddef.h`.

(3.3.4) The result of casting a pointer to an integer, or vice versa:

The bit pattern does not change for pointers and values of type `int`, `long`, `unsigned int` and `unsigned long`.

(3.3.6, 4.1.1) The type of integer required to hold the difference between two pointers to members of the same array,* `ptrdiff_t`*:

`int` as defined in `stddef.h`.

Registers

(3.5.1) The extent to which objects can actually be placed in registers by use of the* `register` *storage-class specifier:

The number of effective register declarations depends on patterns of use and definition within each function and is bounded by the number of registers available for allocation. Neither the compiler nor the optimizer is required to honor register declarations.

Structures, Unions, Enumerations, and Bit-Fields

(3.3.2.3) A member of a union object is accessed using a member of a different type:

The bit pattern stored in the union member is accessed, and the value interpreted, according to the type of the member by which it is accessed.

(3.5.2.1) The padding and alignment of members of structures.

Table B-3 Padding and Alignment of Structure Members

Type	Alignment Boundary	Byte Alignment
`char`	Byte	1
`short`	Halfword	2
`int`	Word	4
`long`	Word	4
`float`	Word	4
`double`	Doubleword *(SPARC)* Word *(Intel)*	8 *(SPARC)* *(PowerPC)* 4 *(Intel)*
`long double`	Doubleword *(SPARC)* Word *(Intel)* Quadword *(PowerPC)*	8 *(SPARC)* 4 *(Intel)* 16 *(PowerPC)*
`pointer`	Word	4
`long long`[1]	Doubleword *(SPARC)* Word *(Intel)*	8 *(SPARC)* *(PowerPC)* 4 *(Intel)*

1. Not available in `-Xc` mode.

Structure members are padded internally, so that every element is aligned on the appropriate boundary.

Alignment of structures is the same as its more strictly aligned member. For example, a `struct` with only `char`s has no alignment restrictions, whereas a `struct` containing a `double` would be aligned on an 8-byte boundary.

(3.5.2.1) Whether a plain `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field:

It is treated as an `unsigned int`.

(3.5.2.1) The order of allocation of bit-fields within an `int`:

Bit-fields are allocated within a storage unit from high-order to low-order.

(3.5.2.1) Whether a bit-field can straddle a storage-unit boundary:

Bit-fields do not straddle storage-unit boundaries.

(3.5.2.2) The integer type chosen to represent the values of an enumeration type:

This is an `int`.

Qualifiers

(3.5.3) What constitutes an access to an object that has volatile-qualified type:

Each reference to the name of an object constitutes one access to the object.

Declarators

(3.5.4) The maximum number of declarators that may modify an arithmetic, structure, or union type:

No limit is imposed by the compiler.

Statements

(3.6.4.2) The maximum number of `case` ***values in a*** `switch` ***statement:***

No limit is imposed by the compiler.

Preprocessing Directives

(3.8.1) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set:

A character constant within a preprocessing directive has the same numeric value as it has within any other expression.

(3.8.1) Whether such a character constant may have a negative value:

Character constants in this context may have negative values *(SPARC) (Intel)* .

Character constants in this context may not have negative values *(PowerPC)*.

(3.8.2) The method for locating includable source files:

A file whose name is delimited by `< >` is searched for first in the directories named by the `-I` option, and then in the standard directory. The standard directory is `/usr/include`, unless the `-YI` option is used to specify a different default location.

A file whose name is delimited by quotes is searched for first in the directory of the source file that contains the `#include`, then in directories named by the `-I` option, and last in the standard directory.

If a file name enclosed in `< >` or double quotes begins with a `/` character, the file name is interpreted as a path name beginning in the root directory. The search for this file begins in the root directory only.

(3.8.2) The support of quoted names for includable source files:

Quoted file names in `include` directives are supported.

(3.8.2) The mapping of source file character sequences:

Source file characters are mapped to their corresponding ASCII values.

(3.8.6) The behavior on each recognized `#pragma` directive:

The following pragmas are supported:

`align` *integer* `(`*variable*`[ ,` *variable*`])`
: Makes the specified variables memory aligned to *integer* bytes, overriding the default. *integer* must be a power of 2, between 1 and 128. Valid values are: 1, 2, 4, 8, 16, 32, 64, 128. *variable* is a global or static variable; it cannot be a dynamic variable. If the specified alignment is smaller than the default, the default is silently used. The `pragma` line must appear before the declaration of the variables that it mentions; otherwise, it is ignored. Any *variable* that is mentioned but not declared in the text following the `pragma` line is ignored.

For example, the compilation and execution of the following program:

```
#define AL2 128
#pragma align AL2 (astruct, aint, apointer)

typedef struct {double a; long long t;} s;

int aint;
char * apointer;

main (int argc, char *argv[]) {

    static s astruct;

    printf ("align:\n");
    printf ("aint=%x mod=%x\n",&aint,((long)&aint)%AL2);
    printf ("apointer=%x mod=%x\n",&apointer,((long)&apointer)%AL2);
    printf ("astruct=%x mod=%x\n",&astruct,((long)&astruct)%AL2);
}
```

produces this output:

```
align:
aint=20900 mod=0
apointer=20980 mod=0
astruct=20880 mod=0
```

`fini (`*f1*`[,`*f2...,fn*`])`
Causes the implementation to call functions *f1* to *fn* (finalization functions) after it calls `main()` routine. Such functions are expected to be of type `void` and to accept no arguments, and are called either when a program terminates under program control or when the containing shared object is removed from memory. As with “initialization functions,” finalization functions are executed in the order processed by the link editors.

`init (`*f1*`[,`*f2...,fn*`])`
Causes the implementation to call functions *f1* to *fn* (initialization functions) before it calls `main()` routine. Such functions are expected to be of type `void` and to accept no arguments, and are called while constructing the memory image of the program at the start of execution. In the case of

initializers in a shared object, they are executed during the operation that brings the shared object into memory, either program startup or some dynamic loading operation, such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editors, both static and dynamic.

`ident` *string*
Places *string* in the `.comment` section of the executable.

`int_to_unsigned` *function name*
For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, changes the function return to be of type `int`.

`MP serial_loop`
(SPARC) Refer to "Serial Pragmas" on page 74 for details.

`MP serial_loop_nested`
(SPARC) Refer to "Serial Pragmas" on page 74 for details.

`MP taskloop`
*(SPARC)*Refer to "Parallel Pragmas" on page 74 for details.

`nomemorydepend`
(SPARC) This pragma specifies that for any iteration of a loop, there are no memory dependences. That is, within any iteration of a loop there are no references to the same memory. This pragma will permit the compiler (pipeliner) to schedule instructions, more effectively, within a single iteration of a loop. If any memory dependences exist within any iteration of a loop, the results of executing the program are undefined. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.

`no_side_effect`*(funcname)*
(SPARC) funcname specifies the name of a function within the current translation unit. The function must be declared prior to the pragma. The pragma must be specified prior to the function's definition. For the named function, *funcname*, the pragma declares that the function has no side effects of any kind. The compiler can use this information when doing optimizations using the function. If the function does have side effects, the results of executing a program which calls this function are undefined. The compiler takes advantage of this information at optimization level of 3 or above.

`pack` *(n)*
Controls the layout of structure offsets. *n* is a number, 1, 2, or 4, that specifies the strictest alignment desired for any structure member. If *n* is omitted, members are aligned on their natural boundaries. If you are using `#pragma pack`(1), be sure to place it after *all* `#include`s.

`#pragma pipeloop`(*n*)
(SPARC) This pragma accepts a positive constant integer value, or 0, for the argument *n*. This pragma specifies that a loop is pipelinable and the minimum dependence distance of the loop-carried dependence is *n*. If the distance is 0, then the loop is effectively a Fortran-style `doall` loop and should be pipelined on the target processors. If the distance is greater than 0, then the compiler (pipeliner) will only try to pipeline *n* successive iterations. The pragma applies to the next `for` loop within the current block. The compiler takes advantage of this information at optimization level of 3 or above.

`#pragma redefine_extname old_extname new_extname`
The pragma causes every externally defined occurrence of the name "old_extname" in the object code to be "new_extname". Such that, at link time only the name "new_extname" is seen by the loader.

If `pragma redefine_extname` is encountered after the first use of "old_extname", as a function definition, an initializer, or an expression, the effect is undefined. (Not supported in `-Xs` and `-Xc` modes.)

`unknown_control_flow (`***name***`, [,`***name***`])`
Specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flow graph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

`#pragma unroll` (*unroll_factor*)
(SPARC) This pragma accepts a positive constant integer value for the argument *unroll_factor*. The pragma applies to the next `for` loop within the current block. For unroll factor other than 1, this directive serves as a suggestion to the compiler that the specified loop should be unrolled by the given factor. The compiler wil, when possible, use that unroll factor. When

the unroll factor value is 1, this directive serves as a command which specifies to the compiler that the loop is not to be unrolled. The compiler takes advantage of this information at optimization level of 3 or above.

`weak` *function_name* or *function_name*1 = *function_name2*

Use `#pragma weak` to define a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not produce an error if it is unable to resolve a weak symbol.

```
#pragma weak function_name
```

defines *function_name* to be a weak symbol. The linker does not complain if it does not find a definition for *function_name*.

```
#pragma weak function_name1 = function_name2
```

defines *function_name1* to be a weak symbol, which is an alias for the symbol *function_name2*.

If your program calls but does not define *function_name1*, the linker uses the definition from the library. However, if your program defines its own version of *function_name1*, then the program definition is used and the weak global definition of *function_name1* in the library is not used. If the program directly calls *function_name2*, the definition from the library is used; a duplicate definition of *function_name2* causes an error.

(3.8.8) The definitions for `__DATE__` *and* `__TIME__` *when, respectively, the date and time of translation are not available:*

These macros are always available from the environment.

Library Functions

(4.1.5) The null pointer constant to which the macro `NULL` *expands:*

`NULL` equals 0.

(4.2) The diagnostic printed by and the termination behavior of the `assert` *function:*

The diagnostic is:

`Assertion failed:` ***statement*** `. file` ***filename*** `, line` ***number***

Where:

- *statement* is the statement which failed the assertion
- *filename* is the name of the file containing the failure
- *line number* is the number of the line on which the failure occurs

(4.3.1) The sets of characters tested for by the `isalnum`*,* `isalpha`*,* `iscntrl`*,* `islower`*,* `isprint`*, and* `isupper` *functions:*

Table B-4 Character Sets Tested by `isalpha`, `islower`, Etc.

`isalnum`	ASCII characters A-Z, a-z and 0-9
`isalpha`	ASCII characters A-Z and a-z, plus locale-specific single-byte letters
`iscntrl`	ASCII characters with value 0-31 and 127
`islower`	ASCII characters a-z
`isprint`	Locale-specific single-byte printable characters
`isupper`	ASCII characters A-Z

(4.5.1) The values returned by the mathematics functions on domain errors:

Table B-5 Values Returned on Domain Errors

Error	Math Functions	Compiler Modes	
		-Xs, -Xt	-Xa, -Xc
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

(4.5.1) Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors:

Mathematics functions, except `scalbn`, set `errno` to `ERANGE` when underflow is detected.

(4.5.6.4) Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero:

In this case, it returns the first argument with domain error.

Signals

(4.7.1.1) The set of signals for the `signal` function:

Table B-6 shows the semantics for each signal as recognized by the `signal` function:

Table B-6 Semantics for `signal` Signals

Signal	No.	Default	Event
SIGHUP	1	Exit	hangup
SIGINT	2	Exit	interrupt
SIGQUIT	3	Core	quit
SIGILL	4	Core	illegal instruction (not reset when caught)
SIGTRAP	5	Core	trace trap (not reset when caught)
SIGIOT	6	Core	IOT instruction
SIGABRT	6	Core	Used by abort
SIGEMT	7	Core	EMT instruction
SIGFPE	8	Core	floating point exception
SIGKILL	9	Exit	kill (cannot be caught or ignored)
SIGBUS	10	Core	bus error
SIGSEGV	11	Core	segmentation violation
SIGSYS	12	Core	bad argument to system call
SIGPIPE	13	Exit	write on a pipe with no one to read it
SIGALRM	14	Exit	alarm clock
SIGTERM	15	Exit	software termination signal from kill
SIGUSR1	16	Exit	user defined signal 1
SIGUSR2	17	Exit	user defined signal 2

Table B-6 Semantics for `signal` Signals *(Continued)*

Signal	No.	Default	Event
SIGCLD	18	Ignore	child status change
SIGCHLD	18	Ignore	child status change alias
SIGPWR	19	Ignore	power-fail restart
SIGWINCH	20	Ignore	window size change
SIGURG	21	Ignore	urgent socket condition
SIGPOLL	22	Exit	pollable event occurred
SIGIO	22	Exit	socket I/O possible
SIGSTOP	23	Stop	stop (cannot be caught or ignored)
SIGTSTP	24	Stop	user stop requested from tty
SIGCONT	25	Ignore	stopped process has been continued
SIGTTIN	26	Stop	background tty read attempted
SIGTTOU	27	Stop	background tty write attempted
SIGVTALRM	28	Exit	virtual timer expired
SIGPROF	29	Exit	profiling timer expired
SIGXCPU	30	Core	exceeded cpu limit
SIGXFSZ	31	Core	exceeded file size limit
SIGWAITINGT	32	Ignore	process's lwps are blocked

(4.7.1.1) The default handling and the handling at program startup for each `signal` ***recognized by the signal function:***

See above.

(4.7.1.1) If the equivalent of `signal(sig, SIG_DFL);` ***is not executed prior to the call of a signal handler, the blocking of the signal that is performed:***

The equivalent of `signal(sig,SIG_DFL)` is always executed.

(4.7.1.1) Whether the default handling is reset if the `SIGILL` ***signal is received by a handler specified to the signal function:***

Default handling is not reset in `SIGILL`.

Streams and Files

(4.9.2) Whether the last line of a text stream requires a terminating new-line character:

The last line does not need to end in a newline.

(4.9.2) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in:

All characters appear when the stream is read.

(4.9.2) The number of null characters that may be appended to data written to a binary stream:

No null characters are appended to a binary stream.

(4.9.3) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file:

The file position indicator is initially positioned at the end of the file.

(4.9.3) Whether a write on a text stream causes the associated file to be truncated beyond that point:

A write on a text stream does not cause a file to be truncated beyond that point unless a hardware device forces it to happen.

(4.9.3) The characteristics of file buffering:

Output streams, with the exception of the standard error stream (`stderr`), are by default-buffered if the output refers to a file, and line-buffered if the output refers to a terminal. The standard error output stream (`stderr`) is by default unbuffered.

A buffered output stream saves many characters, and then writes the characters as a block. An unbuffered output stream queues information for immediate writing on the destination file or terminal immediately. Line-buffered output queues each line of output until the line is complete (a newline character is requested).

(4.9.3) Whether a zero-length file actually exists:

A zero-length file does exist since it has a directory entry.

(4.9.3) The rules for composing valid file names:

A valid file name can be from 1 to 1,023 characters in length and can use all character except the characters `null` and / (slash).

(4.9.3) Whether the same file can be open multiple times:

The same file can be opened multiple times.

(4.9.4.1) The effect of the `remove` ***function on an open file:***

The file is deleted on the last call which closes the file. A program cannot open a file which has already been removed.

(4.9.4.2) The effect if a file with the new name exists prior to a call to the `rename` ***function:***

If the file exists, it is removed and the new file is written over the previously existing file.

(4.9.6.1) The output for `%p` ***conversion in the*** `fprintf` ***function:***

The output for `%p` is equivalent to `%x`.

(4.9.6.2) The input for `%p` ***conversion in the*** `fscanf` ***function:***

The input for `%p` is equivalent to `%x`.

(4.9.6.2) The interpretation of a – character that is neither the first nor the last character in the scan list for `%[` ***conversion in the*** `fscanf` ***function:***

The – character indicates an inclusive range; thus, `[0-9]` is equivalent to `[0123456789]`.

errno

(4.9.9.4) The value to which the macro `errno` *is set by the* `fgetpos` *or* `ftell` *function on failure:*

`errno` is set to `EBADF`, `ESPIPE`, or `EINVAL` on failure.

(4.9.10.4) The messages generated by the `perror` *function:*

These messages, or their translation into the language of the locale of the `LC_MESSAGE` category, are generated.

Table B-7 Error Messages Generated by `perror`

Number	Message
1	Not owner
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Arg list too long
8	Exec format error
9	Bad file number
10	No child processes
11	No more processes
12	Not enough space
13	Permission denied
14	Bad address
15	Block device required
16	Device busy
17	File exists
18	Cross-device link
19	No such device

Table B-7 Error Messages Generated by `perror` *(Continued)*

Number	Message
20	Not a directory
21	Is a directory
22	Invalid argument
23	File table overflow
24	Too many open files
25	Not a typewriter
26	Text file busy
27	File too large
28	No space left on device
29	Illegal seek
30	Read-only file system
31	Too many links
32	Broken pipe
33	Argument out of domain
34	Result too large
35	No message of desired type
36	Identifier removed
37	Channel number out of range
38	Level 2 not synchronized
39	Level 3 halted
40	Level 3 reset
41	Link number out of range
42	Protocol driver not attached
43	No CSI structure available
44	Level 2 halted
45	Deadlock situation detected/avoided
46	No record locks available

Table B-7 Error Messages Generated by `perror` *(Continued)*

Number	Message
50	Bad exchange descriptor
51	Bad request descriptor
52	Message tables full
53	Inode table overflow
54	Bad request code
55	Invalid slot
56	File locking deadlock
57	Bad font file format
60	Not a stream device
61	No data available
62	Timer expired
63	Out of stream resources
64	Machine is not on the network
65	Package not installed
66	Object is remote
67	Link has been severed
68	Advertise error
69	Srmount error
70	Communication error on send
71	Protocol error
74	Multihop attempted
77	Not a data message
78	File name too long
79	Value too large for defined data type
80	Name not unique on network
81	File descriptor in bad state
82	Remote address changed

Table B-7 Error Messages Generated by `perror` *(Continued)*

Number	Message
83	Can not access a needed shared library
84	Accessing a corrupted shared library
85	.lib section in a.out corrupted
86	Attempting to link in more shared libraries than system limit
87	Can not exec a shared library directly
88	Illegal byte sequence
89	Operation not applicable
90	Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS
93	Directory not empty
94	Too many users
95	Socket operation on non-socket
96	Destination address required
97	Message too long
98	Protocol wrong type for socket
99	Option not supported by protocol
120	Protocol not supported
121	Socket type not supported
122	Operation not supported on transport endpoint
123	Protocol family not supported
124	Address family not supported by protocol family
125	Address already in use
126	Cannot assign requested address
127	Network is down
128	Network is unreachable
129	Network dropped connection because of reset
130	Software caused connection abort

Table B-7 Error Messages Generated by `perror` *(Continued)*

Number	Message
131	Connection reset by peer
132	No buffer space available
133	Transport endpoint is already connected
134	Transport endpoint is not connected
135	Structure needs cleaning
137	Not a name file
138	Not available
139	Is a name file
140	Remote I/O error
141	Reserved for future use
142	Error 142
143	Cannot send after socket shutdown
144	Too many references: cannot splice
145	Connection timed out
146	Connection refused
147	Host is down
148	No route to host
149	Operation already in progress
150	Operation now in progress
151	Stale NFS file handle

Memory

(4.10.3) The behavior of the* `calloc`*,* `malloc`*, or* `realloc` *function if the size requested is zero:

`malloc` and `calloc` return a unique pointer if the size is zero. `realloc` frees the object pointed to if the size is zero, and the pointer is not null.

`abort` Function

(4.10.4.1) The behavior of the* `abort` *function with regard to open and temporary files:

`abort` first closes all open files, `stdio` streams, directory streams, and message catalogue descriptors, if possible, and then sends the signal `SIGABRT` to the calling process.

`exit` Function

(4.10.4.3) The status returned by the* `exit` *function if the value of the argument is other than zero,* `EXIT_SUCCESS`*, or* `EXIT_FAILURE`*:

The value returned by the argument to `exit`.

`getenv` Function

(4.10.4.4) The set of environment names and the method for altering the environment list used by the* `getenv` *function:

The set of environment names provided to a program are the same as those that were in the environment when the program was executed. Any environment variable altered during program execution does not permanently change the environment variable; that is, the environment variable has the same value upon program completion as it did before the program was executed.

`system` *Function*

(4.10.4.5) The contents and mode of execution of the string by the* `system` *function:

```
(void) execl("/sbin/sh","sh",(const char *)"-c", string, (char
*)0);
```

`strerror` *Function*

(4.11.6.2) The contents of the error message strings returned by the* `strerror` *function:

See 4.9.10.4.

Locale Behavior

(4.12.1) The local time zone and Daylight Savings Time:

The local time zone is set by the environment variable `TZ`.

(4.12.2.1) The era for the* `clock` *function

The era for the clock is represented as clock ticks with the origin at the beginning of the execution of the program.

The following characteristics of a hosted environment are locale-specific:

(2.2.1) The content of the execution character set, in addition to the required members:

Locale-specific (no extension in C locale).

(2.2.2) The direction of printing:

Printing is always left to right.

(4.1.1) The decimal-point character:

Locale-specific (“.” in C locale).

(4.3) The implementation-defined aspects of character testing and case mapping functions:

Same as 4.3.1.

(4.11.4.4) The collation sequence of the execution character set:

Locale-specific (ASCII collation in C locale).

(4.12.3.5) The formats for time and date:

Locale-specific. Formats for the C locale are shown in the tables below.

The names of the months are:

Table B-8 Names of Months

January	May	September
February	June	October
March	July	November
April	August	December

The names of the days of the week are:

Table B-9 Days and Abbreviated Days of the Week

Days		**Abbreviated Days**	
Sunday	Thursday	Sun	Thu
Monday	Friday	Mon	Fri
Tuesday	Saturday	Tue	Sat
Wednesday		Wed	

The format for time is:

`%H:%M:%S`

The format for date is:

`%m/%d/%y`

The formats for AM and PM designation are: AM PM

`-Xs` *Differences for Sun C and ANSI C*

This appendix describes the differences in compiler behavior when using the `-Xs` option. The `-Xs` option tries to emulate Sun C 1.0, and Sun C 1.1 (K&R style), but in some cases it cannot emulate the previous behavior.

Table C-1 `-Xs` Behavior

Data Type	Sun C (K&R)	Sun ANSI C (4.0)

Aggregate initialization:
`struct {`
`int a[3];`
`int b;`
`} w[] = { {1} , 2};`
	`sizeof (w) = 16`
	`w[0].a = 1, 0, 0`
	`w[0].b =2`
		`sizeof(w) = 32`
		`w[0].a = 1, 0, 0`
		`w[0].b = 0`
		`w[1].a = 2, 0, 0`
		`w[1].b = 0`

Incomplete `struct`, `union`, `enum` declaration	`struct fq {`
	`int i;`
	`struct unknown;`
	`};`
		Does not allow incomplete `struct`, `union`, and `enum` declaration.

Switch expression integral type	Allows non-integral type.	Does not allow non-integral type.
Order of precedence	Allows: `if (rcount > count += index)`	Does not allow: `if (rcount > count += index)`

`unsigned`, `short`, and `long` `typedef` declarations	Allows:
	`typedef short small`
	`unsigned small;`
		Does not allow (all modes).

Table C-1 `-Xs` Behavior *(Continued)*

Data Type	**Sun C (K&R)**	**Sun ANSI C (4.0)**

`struct` or `union` tag mismatch in nested `struct` or `union` declarations	Allows tag mismatch:
	`struct x {`
	`int i;`
	`} s1;`
	
	`/* K&R treats as a struct */`
	`{`
	`union x s2;`
	`}`
		Does not allow tag mismatch in nested `struct` or `union` declaration.

Incomplete `struct` or `union` type	Ignores an incomplete type declaration.	`struct x {`
		`int i;`
		`} s1;`
		
		`main()`
		`{`
		`struct x;`
		`struct y {`
		`struct x f1`
		`/* in K&R, f1 refers */`
		`/* to outer struct */`
		`} s2;`
		`struct x {`
		`int i;`
		`};`
		`}`

Casts as `lvalues`	Allows: `(char *) ip = &foo;`	Does not allow casts as `lvalues` (all modes).

Performance Tuning (SPARC) D

This appendix describes performance tuning on SPARC platforms.

Limits

Some parts of the C library cannot be optimized for speed, even though doing so would benefit most applications. Some examples:

- **Integer arithmetic routines**—Current SPARC V8 processors support integer multiplication and division instructions. However, if standard C library routines were to use these instructions, programs running on V7 SPARC processors would either run slowly due to kernel emulation overhead, or might break altogether. Hence, integer multiplication and division instructions cannot be used in the standard C library routines.
- **Doubleword memory access**—Block copy and move routines, such as `memmove()` and `bcopy()`, could run considerably faster if they used SPARC doubleword load and store instructions (`ldd` and `std`). Some memory-mapped devices, such as frame buffers, do not support 64-bit access; nevertheless, these devices are expected to work correctly with `memmove()` and `bcopy()`. Hence, `ldd` and `std` cannot be used in the standard C library routines.
- **Memory allocation algorithms**—The C library routines `malloc()` and `free()` are typically implemented as a compromise between speed, space, and insensitivity to coding errors in old UNIX programs. Memory allocators based on "buddy system" algorithms typically run faster than the standard library version, but tend to use more space.

`libfast.a` *Library*

The library `libfast.a` provides speed-tuned versions of standard C library functions. Because it is an optional library, it can use algorithms and data representations that may not be appropriate for the standard C library, even though they improve the performance of most applications.

Use profiling to determine whether the routines in the following checklist are important to the performance of your application, then use this checklist to decide whether `libfast.a` benefits the performance:

- *Do* use `libfast.a` if performance of integer multiplication or division is important, even if a single binary version of the application must run on both V7 and V8 SPARC platforms.

  The important routines are: `.mul`, `.div`, `.rem`, `.umul`, `.udiv`, and `.urem`.

- *Do* use `libfast.a` if performance of memory allocation is important, and the size of the most commonly allocated blocks is close to a power of two.

  The important routines are: `malloc()`, `free()`, `realloc()`.

- *Do* use `libfast.a` if performance of block move or fill routines is important.

  The important routines are: `bcopy()`, `bzero()`, `memcpy()`, `memmove()`, and `memset()`.

- *Do not* use `libfast.a` if the application requires user mode, memory-mapped access to an I/O device that does not support 64-bit memory operations.
- *Do not* use `libfast.a` if the application is multithreaded.

When linking the application, add the option `-lfast` to the `cc` command used at link time. The `cc` command links the routines in `libfast.a` ahead of their counterparts in the standard C library.

Index

A

abort function, 176
acomp (C compiler), 2
alignment of structures, 158
ANSI C, xxi
ANSI C vs. K&R C, xxi, 5, 23, 179
arithmetic conversions, 58, 62
_asm keyword, 58
asm keyword, 58
assembler, 2
#assert, 7, 66
Auto-Read, 44

B

behavior, implementation-defined, 151 to 178
binding
  static vs. dynamic, 7
bit-fields, 130, 158
bits, in execution character set, 153
bitwise
  operations on signed integers, 155
buffering, 169

C

c89, xxiv
calloc function, 176
case statements, 159
cb, xxiv
cc
  shared error message database
    catfiles, 55
cc compiler options, 5 to 54
  -#, 6
  -###, 6
  -A*name* [(*tokens*)], 7
  -B[static|dynamic], 7
  -C, 7
  -c, 7
  -d[y|n], 8
  -dalign, 9
  -D*name* [(=*tokens*)], 7
  -E, 9
  -erroff=*t*, 9
  -errtags=*a*, 10
  -fast, 10
  -fd, 11
  -flags, 11
  -fnonstd, 11
  -fns, 12
  -fprecision=*r*, 12
  -fround=*r*, 12

-fsimple[=*n*], 12
-fsingle, 13
-fstore, 14
-ftrap=*t*, 14
-G, 14
-g, 15
-H, 15
-h, 15
-I*dir*, 16
-J, 16
-keeptmp, 16
-KPIC, 16
-Kpic, 16
-L*dir*, 17
-L*name*, 18
-mc, 18
-misalign, 18
-misalign2, 18
-mr, 19
-mr, *string*, 19
-mt, 19
-native, 19
-nofstore, 19
-noqueue, 20
-O, 20
-o *filename*, 20
-P, 20
-p, 20
-Q[y|n], 20
-qp, 21
-R*dir*[:*dir*], 21
-S, 21
-s, 21
-U*name*, 21
-V, 21
-v, xxvii, 22
-w, 23
-W*c*, *arg*, 22
-X[a|c|s|t], 23
-x386, 23
-x486, 24
-xa, 24
-xarch=*a*, 24
-xautopar, 27
-xcache=*c*, 28
-xCC, 29
-xcg[89|92], 29
-xchip=*c*, 29
-xcrossfile, 30
-xdepend, 31
-xe, 31
-xexplicitpar, 31
-xF, 32
-xhelp=*f*, 32
-xildoff, 33
-xildon, 33
-xinline=[*f1,...,fn*], 33
-xlibmieee, 34
-xlibmil, 34
-xlic_lib, 34
-xlic_lib=sunperf, 34
-xlicinfo, 34
-xloopinfo, 34
-xM, 35
-xM1, 36
-xMerge, 36
-xnolib, 36
-xnolibmil, 37
-xO[1|2|3|4|5], 37
-xP, 39
-xparallel, 39
-xpentium, 40
-xpg, 40
-xprofile=*p*, 40
-xreduction, 42
-xregs=*r*, 42
-xrestrict=*f*, 43
-Xs, 179
-xs, 44
-xsafe=mem, 44
-xsb, 44
-xsbfast, 44
-xsfpconst, 45
-xspace, 45
-xstrconst, 45
-xtarget=*t*, 45
-xtemp=*dir*, 51
-xtime, 51
-xtransition, 51, 58
-xunroll=*n*, 52
-xvpara, 52
-YA, *dir*, 53

-Yc, *dir*, 52
-YI, *dir*, 53
-YP, *dir*, 53
-YS, *dir*, 53
-Zll, 53
-Zlp, 53
-Ztha, 54
cflow, xxiv
cg (code generator), 3
cg386 (intermediate language translator), 2
cgppc (intermediate language translator), 2
character
bits in set, 153
decimal point, 177
mapping set, 153
multibyte, shift status, 153
set, collation sequence, 178
single-character characater constant, 159
source and execution of set, 153
space, 169
testing of sets, 165
clock function, 177
code generator, 3
code optimization, 11, 37
code optimizer, 2
codegen (code generator), 3
compatibility options, 5, 23
compilation modes and dependencies, 71
compiler, 54
compiler flags, *See* cc compiler options
compiling a program, 5 to 6
constants, 63 to 64
conversion
integers, 155
conversions, 58, 62
cpp (C preprocessor), 2
cscope, xxiv, 83 to 102
command-line use, 85, 93 to 95
editing source files, 84, 92, 101
environment setup, 84 to 85, 102
environment variables, 95 to 96
searching source files, 83 to 84, 85, 86 to 92
See also SourceBrowser
usage examples, 84 to 93, 97 to 101
ctrace, xxiv
cxref, xxiv

D

data representation, 137 to ??
data types, 61
__DATE__, 164
date and time formats, 178
dbx tool
disable Auto-Read for, 44
initializes faster, 44
symbol table information for, 15
debugging information, removing, 21
decimal-point character, 177
declarators, 159
default
compiler behavior, 23
handling and SIGILL, 168
installation path name, xxx
locale, 153
#define, 7
diagnostics, format, 151
directives, 66
documentation for C compiler, xxii to xxviii
domain errors, math functions, 166
double load/store instructions, 9

E

edit, source files, *See* cscope
environment variables, xxiii, 27, 57, 73, 84, 85, 101, 153, 177
ERANGE, 166
errno, 166, 171
Error message
source file names, 54

Error message catalogs, 54
Error message database
shared between cc and lint commands, 55
Error messages
localization, 54
error messages, xxv, 119, 151
See also message ID (tag)
exit function, 176

F

faster linking and initializing, 44
fbe (assembler), 2
features, new in this release, *See* README file
fgetpos function, 171
files
implementation-defined behavior, 169 to 170
See also #include files; source files
temporary, 57
flags, compiler, *See* options, compiler
float expressions as single precision, 13
floating point, 156
gradual underflows, 65
nonstandard, 11
nonstop, 65
representations, 156
truncation, 156, 157
values, 156
fprintf function, 170
fscanf function, 170
ftell function, 171
function
abort, 176
calloc, 176
clock, 177
exit, 176
fgetpos, 171
fmod, 167
fprintf, 170
fscanf, 170
ftell, 171
getenv, 176
malloc, 176
perror, 171
prototypes, 128
prototypes, lint checks for, 133
realloc, 176
remove, 170
rename, 170
strerror, 177
system, 177

G

gencat utility
creating formatted message database catfiles, 54
getenv function, 176
gprof, xxv
gradual underflows, 65

H

header files
format, 64
how to include, 64 to 65
standard place, 64 to 65
with lint, 107 to 108
help, *See* documentation for C compiler

I

identifiers, 152
ild, xxv, xxviii, 3, 33
iMPact, 72
implementation-defined behavior, 151 to 178
#include files, 64 to 65, ?? to 94
incremental linker, xxv, xxviii, 3, 33
indent, xxiv
inline, xxv
inline expansion templates, 34, 37
inlining, xxv, 34
installation path name, xxx
integers, 154 to 155

interactive device, 152
iropt (code optimizer), 2
isalnum, 165
isalpha, 165
iscntrl, 165
islower, 165
isprint, 165
isupper, 165

K

K&R C vs. ANSI C, xxi, 5, 23, 179
keywords, 58 to 61

L

ld, xxv, xxviii, 3, 54
lex, xxv
libfast.a, 182
libraries
- intrinsic name, 15
- libfast.a, 182
- lint, 133 to 134
- renaming shared, 15

library bindings, 7
license, xxvii, 72
linker, xxv, xxviii, 3, 33, 44, 54
linker, *See also* incremental linker
linking
- static vs. dynamic, 8

lint, xxiv, 105 to 135
- consistency checks, 128 to 129
- filters, 134 to 135
- libraries, 133 to 134
- messages, 119
- options, 108 to 118
- portability checks, 129 to 131
- predefinitions, 67
- questionable constructs, 131 to 132
- shared error message database
  - catfiles, 55

local time zone, 177
locale
- behavior, 177
- default, 153

Localization
- Using gencat and catges, 54

Localization error messages
- lint, 54

Localization of error messages, 54
long double, 147
long int, 62
long long, 61 to 62
- arithmetic promotions, 62
- passing, 147
- representation of, 139
- returning, 147
- storage allocation, 137
- suffix, 63
- value preserving, 63

long long, 9
loops, 31

M

m4, xxv
macros
- __DATE__, 164
- __RESTRICT, 60, 72
- __TIME__, 164

main
- semantics of args, 152

make, xxv
malloc function, 176
man pages, xxiii to xxv
math functions,domain errors, 166
memory, 176
message ID (tag), xxvi, 9, 10, 112, 113, 119
messages, error, xxv, 119, 151
- *See also* message ID (tag)

messages, lint, 119
mode, compiler, 23
MP C, xxii, 72 to 82
multiprocessing, 72 to 82
mwinline, 2

N

new features in this release, *See* README file
newline, terminating, 169
NLSPATH
  setting the access path, 55
nonstop
  floating-point arithmetic, 11, 65
null characters not appended to data, 169
NULL, value of, 164

O

on-line documentation, xxiii
operating environment, xxx, 1
optimization, 11, 37, 181
optimizer, 2
optimizing performance, 11, 37, 181
options
  -xtarget=*t*, ?? to 51
options, compiler, 5 to 54
options, lint, 108 to 118

P

padding of structures, 158
PARALLEL environment variable, 27, 73
parallelization, 27, 31, 34, 39, 42, 52, 72 to 82
pass, name and version of each, 21
path name, xxx
Pentium, 51
performance,optimizing, 11, 37, 181
perror function, 171
platform, xxx
pointers, restricted, 59 to 61
portability, of code, 106, 129 to 131
#pragma _int_to_unsigned, 69, 162
#pragma _unknown_control_flow, 70, 163
#pragma align, 68, 160
#pragma fini, 68, 161
#pragma ident, 69, 162
#pragma init, 68, 161
#pragma MP serial_loop, 69, 74, 162
#pragma MP serial_loop-nested, 69, 74, 162
#pragma MP taskloop, 69, 74, 162
#pragma no_side_effect, 69, 162
#pragma nomemorydepend, 69, 162
#pragma pack, 70, 163
#pragma pipeloop, 70, 163
#pragma redefine_extname, 70, 163
#pragma unroll, 70, 163
#pragma weak, 71, 164
pragmas, 68 to 71, 160 to 164
predefinitions, 8, 72
preprocessing
  directives, 7, 64 to 65, 71, 159
  predefined names, 71
preprocessor, 35
preserving
  unsigned, 58
  value, 58
printing, 62, 177
prof, xxv
profiling
  with tcov, 24
promotions, 58

Q

qualifiers, 159

R

README file, xxvii
realloc function, 176
remove function, 170
rename function, 170
renaming shared libraries, 15
representation
  floating point, 156
  integers, 154

__RESTRICT macro, 60, 72
_Restrict keyword, 60
restricted pointers, 59 to 61
right shift, 155
rounding behavior, 65

S

sccs, xxv
search, source files, *See* cscope
shared libraries, naming, 15
shared object, 14
signal, 167 to 168
signed, 58, 154
Source Code Control System, *See* sccs
source files
  checking with lint, 105 to 135
  editing, *See* cscope
  indenting, xxiv
  locating, 160
  searching, *See* cscope
SourceBrowser, 102
space characters, 169
standards conformance, xxiv, 1, 57
streams, 169
strerror function, 177
string literals in text segment, 45
structure
  alignment, 158
  padding, 158
SUNPRO_SB_INIT_FILE_NAME
  environment variable, 57
symbol table for dbx, 44
symbolic debugging information,
  removing, 21
system function, 177

T

tcov, xxv
  new style with -xprofile, 41
tcov tool, 24
text
  segment and string literals, 45
  stream, 169
__TIME__, 164
time and date formats, 178
/tmp, 57
TMPDIR environment variable, 57
translation behavior, 151
type conversions, 58
typographic conventions, xxix

U

underflow, gradual, 11
unsigned, 58, 154

V

value
  floating point, 156
  integers, 154
  preserving, 58
volatile, 159

W

warning messages, xxv, 119, 151
  See also message ID (tag)
write on text stream, 169

Y

yacc, xxv

Z

zero-length file, 170
zero-size memory allocation, 176

Copyright 1996 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100, U.S.A. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Des parties de ce produit pourront être dérivées du système UNIX® licencié par Novell, Inc. et du système Berkeley 4.3 BSD licencié par l'Université de Californie. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Solaris, et SunSoft sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc. Intel sont enregistrées de Intel Corporation. PowerPC sont des marques déposées International Business Machines Corporation.

Les interfaces d'utilisation graphique OPEN LOOK® et Sun™ ont été développées par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant aussi les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE “EN L'ETAT” SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.