



CenterLine-C Programmer's Guide

Version 1.1a



CenterLine Software, Inc.
10 Fawcett Street
Cambridge, Massachusetts 02138





CenterLine Software, Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should in all cases consult CenterLine to determine whether any such changes have been made.

This Manual contains proprietary information that is the sole property of CenterLine. This Manual is furnished to authorized users of CenterLine-C solely to facilitate the use of CenterLine-C as specified in written agreements.

No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means without prior explicit written permission from CenterLine Software.

The software programs described in this document are copyrighted and are confidential information and proprietary products of CenterLine Software.

CenterLine and ViewCenter are registered trademarks of CenterLine Software, Inc. CodeCenter, ObjectCenter, ResourceCenter, and TestCenter are trademarks of CenterLine Software, Inc.

Motif is a registered trademark of The Open Software Foundation, Inc.

Object Interface Library (OI) is a trademark of ParcPlace Systems, Inc.

Sun, Sun-2, Sun-3, Sun-4, Solaris 2, Sun386i, SunCD, SunInstall, SunOS, NFS, SunView, ToolTalk, and OpenWindows are trademarks of Sun Microsystems, Inc.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc. licensed exclusively to Sun Microsystems, Inc.

DeltaSeries, DeltaWINDOWS, and SYSTEM V/88 are trademarks of Motorola, Inc. in the USA. Motorola is a registered trademark of Motorola, Inc. in the USA and in other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Co, Ltd. OPEN LOOK is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc. X Window System and X11 are trademarks of the Massachusetts Institute of Technology.

Postscript is a registered trademark of Adobe Systems Incorporated.

© 1992-1995 CenterLine Software, Inc.

All rights reserved.

Printed in the United States of America.





About this manual

What this manual is about

This manual is the programmer's guide for the CenterLine-C™ compiler.

The Programmer's Guide tells how to invoke CenterLine-C, choose switches, specify pragmas, turn compiler toggles on and off, and optimize performance. It also describes C data types, system specifics, listings and cross references, inlining, loop unrolling, inline assembly code, and diagnostic messages.

What you should know before starting

We designed this book for readers who are familiar with the C programming language and the UNIX® operating system.

For more information

The *Release Bulletin* accompanying your CenterLine product contains information generated too late to be included in the other documentation.

Platform-specific information, including run-time organization, is described in the appendixes. UNIX manual pages are available for the ANSI C library distributed on the SunOS 4.x platform.

Documentation conventions

Unless otherwise noted in the text, we use the following symbolic conventions:

literal names	Bold words or characters in command descriptions represent words or values that you must use literally.
<i>user-supplied values</i>	Italic words or characters in command descriptions represent values that you must supply. Italic words in text also indicate emphasis or the first use of a new term.
<code>sample user input</code>	In interactive examples, information that you must enter appears in this typeface .
output/source code	Information that the system displays appears in <i>this typeface</i> .
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.





Contents

About this manual iii

Chapter 1 Introduction 1

Invoking the compiler 3

Predefined macros 4

Using the ANSI C library 6

Chapter 2 Command-line switches 7

Command-line switches 9

Chapter 3 Pragas and toggles 15

Compiler pragmas 17

Compiler toggles 19

Chapter 4 Optimizing program performance 25

Specifying an optimization level 27

Debugging optimized code 28

Execution speed versus code size 28

Optimizations at each level 30

Loop unrolling 38

Chapter 5 Inlining routines 41

What is inlining? 43

Invoking and activating the inliner 44

Switches for inlining 45

How routines are selected for inlining 48

Debugging inlined routines 48



Contents

Chapter 6 Listings and cross references 49

Generating listings and cross references 51

Pragmas Page, Skip, Title 51

Listing format 52

Cross-reference features 53

Using the `clccxref` command 54

Cross-reference pragmas 56

Cross-reference format 58

Distinction of file names 60

Chapter 7 System specifics 61

Identifiers 63

Data types 63

Preprocessing directives 65

Debugger support 65

Chapter 8 Storage mapping 67

Size and alignment of data types 69

Arrays 70

Bit fields 70

Storage classes 72

Chapter 9 Inline assembly code 73

`_ASM` and direct register assignment for variables 75

Restrictions on using `_ASM` 76



Chapter 10 Diagnostic messages 77

File I/O errors 79

System errors 79

User errors and warnings 80

Warning levels 81

Error and warning messages 82

Appendix A HP run-time organization 89

Stack-frame layout 91

Register usage 93

How function arguments are passed 94

How function values are returned 95

Prologue/epilogue code 96

Appendix B Sun run-time organization 99

Stack-frame layout 101

Register usage 102

How function arguments are passed 104

How function values are returned 104

Prologue/epilogue code 105

Assembly-language communication 107

Index 109



List of Tables

List of Tables

Table 1	Definitions of Environment Macros	5
Table 2	CenterLine-C Command-Line Switches	9
Table 3	Pragmas	18
Table 4	Compiler Toggles	20
Table 5	Optimization Benefits	29
Table 6	Optimization Levels	30
Table 7	Inlining switches	34
Table 8	Switches for Inlining	45
Table 9	clccxref Command-Line Switches	55
Table 10	Integer Ranges	64
Table 11	Size of Floating-Point Data Type	65
Table 12	C Data Types: Size and Alignment	69
Table 13	Bit-Field Widths	71
Table 14	HP Register Usage and Naming Conventions	93
Table 15	HP Calling Convention	94
Table 16	HP Registers Used for Return Types	95

List of Figures

Figure 1	HP Memory Stack Frame Organization	92
Figure 2	Sun Memory Stack Frame Organization	101
Figure 3	Sun Integer and Floating-Point Registers	102
Figure 4	Sun Circular-Model Register Windows	103





Chapter 1 Introduction

CenterLine-C is an optimizing C compiler, compliant with the ANSI C standard and designed to achieve small code size and high-speed code execution. This chapter covers the following topics:

- *Invoking the compiler*
- *Predefined macros*
- *Using the ANSI C library provided on the SunOS 4.x platform*







Invoking the compiler

The `clcc` command invokes the CenterLine-C compiler, which translates C programs into executable load modules, relocatable binary object modules, or assembly language source files. The syntax of the command is:

`clcc [switches]... files...`

Any number of switches and one or more files may be specified. Each switch specified in the command applies to all the specified files for which it makes sense, except as noted below.

File names

Several types of file names are allowed. A file name ending with `.c` is taken to be a C source module. It is compiled and an object module is produced with the same name as the source except with `.o` substituted for `.c`. The `.o` file is normally deleted when a single-module C program is compiled and linked.

A file name ending with `.s` is taken to be an assembly source module and is assembled, producing a `.o` file. Any other file specification is assumed to be an object module or archive library to be passed to the linker.

All `.o` files are placed in the current working directory.

A file name ending with `.i` is taken to be a C source file that has already been processed by a macro processor.

In general, the linker is invoked if no compilation errors were detected and the `-c` switch was not specified. The resultant load module is named `a.out` unless specified otherwise with the `-o` switch (described below). Any argument beginning with a dash (`-`) is taken as a switch specification.

Example

The following command compiles the program in file `sort.c`, links it, and generates a load module named `sort`:

```
clcc -o sort sort.c
```





Introduction

Setting the CLCC_CORE environment variable

You can save some compilation time by setting the environment variable **CLCC_CORE** to the following value:

```
.../CenterLine/clcc/arch-os
```

In the preceding statement, ... is the parent of the CenterLine-C directory, **/CenterLine/clcc** is the CenterLine-C directory name, and *arch-os* is the name of your platform.

Invoking the C macro preprocessor

The CenterLine-C compiler has an integrated inboard macro preprocessor that conforms to the ANSI C Standard. ANSI refers to the American National Standards Institute, which is responsible for the Standard C language and library definition. We provide the preprocessor because the outboard C macro preprocessor supplied with most UNIX operating systems prior to SVR4 did not conform to the Standard in some ways.

Because many C programs written for UNIX operating systems depend on minor idiosyncrasies of the outboard C preprocessor, we provide the **-cpp** switch to enable you to specify another preprocessor. By default, **-cpp** specifies **/lib/cpp**, but you can specify a different preprocessor. See “Command-line switches” on page 9 for a description of **-cpp**.

If you compile with the **-traditional** switch, the default preprocessor is **/lib/cpp**.

Predefined macros

The compiler predefines certain macros. You can use these macros in programs to enable the programs to recognize the environments in which they are running. The definition of the macros in this table depends on the switch you set. For example, if you set the **-ansi** switch, the compiler defines **__STDC__** as 1.

NOTE The columns in the table are arranged from left to right in order of increasing ANSI C compliance required by the switches.

You can see which macros are defined by compiling with the **-v** switch.



Table 1 Definitions of Environment Macros

Macro Name	Switches			
	-traditional	-Xt	-Xa	-ansi
<code>__CLCC__</code>	1	1	1	1
<code>__unix__</code>	1	1	1	1
<code>__hppa^a</code>	1	1	1	1
<code>__hpux^a</code>	1	1	1	1
<code>__hpux_^a</code>	1	1	1	1
<code>__hp9000s700^a</code>	1	1	1	1
<code>__hp9000s800^a</code>	1	1	1	1
<code>__HPUX^a</code>	1	1	1	1
<code>__parisc_^a</code>	1	1	1	1
<code>__PARISC^a</code>	1	1	1	1
<code>__sparc_^b</code>	1	1	1	1
<code>__sol_^c</code>	1	1	1	1
<code>__sun_^d</code>	1	1	1	1
<code>__SPARC^b</code>	1	1	1	1
<code>__SOL^c</code>	1	1	1	1
<code>__SUN^d</code>	1	1	1	1
<code>sun^b</code>	1	1	1	undefined
<code>sparc^b</code>	1	1	1	undefined
<code>unix</code>	1	1	1	undefined
<code>__HIGHC__</code>	undefined	1	1	undefined
<code>__STDC__</code>	undefined	0	0	1

a. Defined on HP-UX platforms only

b. Defined on Sun platforms only

c. Defined on Solaris 2.x platforms only

d. Defined on SunOS 4.x platforms only



Using the ANSI C library

Prior to the introduction of SVR4, many libraries provided with UNIX did not conform to the ANSI C standard. Therefore on the SunOS 4.x platform CenterLine supplies a library and a set of header files that do conform to the ANSI C standard. When you link a program, you can specify which of these libraries you want to use with the **-traditional**, **-Xt**, **-Xa**, or **-ansi** compiler switches.

When is the library used?

In **-traditional** mode, the driver program **clcc** uses the system-provided libraries. In other modes, **clcc** uses the CenterLine-supplied ANSI-C-conforming library and header files to compile and link programs. The ANSI library is searched prior to the standard UNIX C library (**/lib/libc.a**); therefore, most of the ANSI-defined functions present in the UNIX C library are not used.

Referring to non-ANSI functions

If your application references non-ANSI functions (other than system calls) that reside in the UNIX library, for example **getcwd()**, problems may occur if you use the CenterLine-supplied library. Some functions may reference other functions or global variables in the UNIX library that may, in turn, attempt to include more modules from the UNIX library.

This can have the following consequences:

- Multiply defined symbols can occur during linking (for example, **_iob**).
- Incompatible functions can be linked together; for example, the function **fwrite()** from the UNIX library could be linked with **fopen()** from the ANSI library. This may cause problems at run time because internal data structures used for input/output between the two libraries are not necessarily identical. This kind of problem could manifest itself as a segmentation fault from within a library function at run time, or it could exhibit the more subtle behavior of output not being written to a file.

If your program does not depend on functions found only in the UNIX library, you might want to use the **-traditional** switch. This switch specifies that the ANSI library and header files are not to be used.

No POSIX macros defined

The CenterLine-C header files do not define **_POSIX_PATH_MAX** and other POSIX macros. If you need to use a POSIX macro, use the **-traditional** switch or define the macro on the command line or in a separate POSIX header file.





Chapter 2 Command-line switches

This chapter describes the command-line switches that the CenterLine-C compiler recognizes. Switches that are specific to a single platform are called out in the first column of the table.





Command-line switches

Each command-line switch is described in Table 2. The switches are also described in the `clcc` manual page. To view the manual page, issue the `man` command:

```
% man clcc
```

Any switch that is not recognized by `clcc` is assumed to be a linker switch and is passed on to the linker.

NOTE If an argument to a switch passed on to the linker contains embedded spaces, you must enclose the entire argument in quotes.

Table 2 CenterLine-C Command-Line Switches

Switch	Description
<code>-anno</code>	Used with <code>-S</code> to cause assembler file produced to be annotated with the C source code.
<code>-ansi</code>	Direct the compiler to accept only programs that conform to the ANSI C standard. On SunOS 4, the <code>clcc/inc</code> directory is searched before <code>/usr/include</code> , and the ANSI library is used for linking. ^a
<code>-Bbinding</code>	Specify whether bindings of libraries for linking are static or dynamic, indicating whether libraries are non-shared or shared respectively.
<code>-c</code>	Suppress linking and produce a <code>.o</code> file for each source file. Forces an object file to be produced even if only one module is compiled.
<code>-cpp[=name]</code>	Cause a preprocessor other than the <code>clcc</code> preprocessor to be used. If <code>name</code> is not present <code>/lib/cpp</code> is used; otherwise, <code>name</code> is used for the preprocessor name. By default, <code>/lib/cpp</code> is used in <code>-traditional</code> mode.
<code>-Dname[=def]</code>	Define <code>name</code> to the preprocessor. Equivalent to the <code>#define</code> directive. If no <code>def</code> is given, <code>name</code> is defined to be 1.
<code>-dryrun</code>	Show, but do not execute, the compilation commands.



Command-line switches

Table 2 CenterLine-C Command-Line Switches (Continued)

Switch	Description
-E	Specifies that the C macro preprocessor is to be invoked and no compilation done. Preprocessor output is sent to standard output.
-fdouble	Specifies that all floating-point computations are to be done in no less than double precision. Operands of type float are widened appropriately.
-fsingle	Specifies that single-precision arithmetic is to be used in floating-point computations involving only float expressions. This switch is on by default.
-g	Place debugging information in compiled code and provide the linker with information needed to link the file for debugging. Unless the -O switch (on page 12) is used to enforce a certain optimization level, -g turns off many of the optimizations the compiler otherwise performs. (For more information, see "Optimizing program performance" on page 25.) On the HP platform -g also links /usr/lib/end.o .
-Hcppopt=<i>string</i>	<p>Passes cpp switch <i>string</i> to the outboard C preprocessor. Use -Hcppopt to specify preprocessor switches that conflict with compiler switches. You can specify multiple switches by specifying -Hcppopt multiple times, or by separating switches and arguments with commas. You must specify a switch and its corresponding argument separately. The switches are passed in the order given.</p> <p>For example, you can use either of the following to pass -flag1 -flag2 arg to the preprocessor:</p> <pre>clcc files...-Hcppopt=-flag1 -Hcppopt=-flag2 -Hcppopt=arg</pre> <p>or</p> <pre>clcc files...-Hcppopt=-flag1,-flag2,arg</pre>
-Hix	Specifies which functions are inlined. See "Switches for inlining" on page 45 for a description of each of the switches.



**Table 2** CenterLine-C Command-Line Switches (Continued)

Switch	Description
-Hldopt=string	<p>Passes linker switch <i>string</i> to the linker. Use -Hldopt to specify linker switches that conflict with compiler switches. You can specify multiple switches by specifying -Hldopt multiple times, or by separating switches and arguments with commas. You must specify a switch and its corresponding argument separately. The switches are passed in the order given.</p> <p>For example, you can use either of the following to pass -flag1 -flag2 arg to the linker:</p> <pre>clcc files...-Hldopt=-flag1 -Hldopt=-flag2 -Hldopt=arg</pre> <p>or</p> <pre>clcc files...-Hldopt=-flag1,-flag2,arg</pre>
-Hlines=n	Causes a page eject to occur after every <i>n</i> lines written to standard output. The default is 60. This is used in conjunction with the -Hlist switch. If <i>n</i> is zero, no page ejects are emitted.
-Hlist	Causes the compiler to generate a source listing on standard output.
-Hoff=toggle_name -Hon=toggle_name	Turns compiler toggles on or off. See the “Compiler toggles” section on page 19.
-Hsys_as <i>HP only</i>	Causes the compiler to use the system assembler. By default, clcc used the GNU assembler, gas , on the HP platform.
-Hunroll=fff,sss	Cause the compiler to replace a loop with the code that makes up the body of the loop (see “Loop unrolling” on page 38 for notes on this switch). <i>fff</i> is the maximum times a loop is unrolled, and <i>sss</i> is the maximum size of loops to unroll. Loop unrolling reduces loop overhead. You can use this switch multiple times with different loop parameters.
-Ipathname	Add <i>pathname</i> to the list of directories in which to search for #include files with relative filenames (not beginning with slash). The preprocessor first searches in the directory containing the source file, then in the directories named with -I switches (if any), and finally in the system include directories.
-llibrary	Link with object library <i>library</i> .





Command-line switches

Table 2 CenterLine-C Command-Line Switches (Continued)

Switch	Description
-Ldirectory	Specifies the name of a directory to search for a library specified with switch -library . This switch is simply passed to the linker.
-M	Runs only the macro preprocessor, requesting that it generates make file dependencies for the files being compiled and sends them to standard output. See the “Compiler toggles” section on page 19.
-MM	This is similar to -M , but entries for files included with the form #include "file" are not included in the dependencies.
-o file	Name the output file <i>file</i> .
-O[n]	Sets the optimization level to <i>n</i> , where <i>n</i> is a number from 1 to 7. If <i>n</i> is not specified, then a level of 2 is used. If the -O switch is not specified, then the optimization level is 1.
-Os	Optimize for smaller code size.
-p	Prepare the object code to collect data for profiling with prof(1) .
-P	Run the source file through the preprocessor and put the output in a file with a .i suffix.
-pg	Prepare the object code to collect data for profiling with gprof(1) .
-pic	Produce position-independent code. When this switch is used, each reference to a global symbol is generated as a pointer reference in the global offset table. Function calls are generated in pc-relative addressing mode through a procedure linkage table. When you use the -pic switch, the memory allocated to static variables cannot exceed 4K. When you use the -PIC switch, there is no limit on the memory consumed by static variables.
-PIC	Like -pic but allows the global offset table to span the range of 32-bit addresses in those cases where there are too many global data objects for -pic .
-proto	Causes the compiler to write to standard output a new prototype-style function header for each function definition. This aids in the conversion of C programs to the ANSI prototype syntax.



Table 2 CenterLine-C Command-Line Switches (Continued)

Switch	Description
-Qpath <i>dir</i>	Insert directory <i>dir</i> into the compilation search path. The path will be searched for alternative versions of the compilation programs, such as as and ld .
-R <i>SunOS 4.x only</i>	Merge the data segment with text segment for as(1) . Data initialized in the object file produced by this compilation is a read-only, and (unless linked with ld -N) is shared between processes. Ignored when -g is used. This switch just gets passed to as(1) .
-R <i>dir_list</i> <i>Solaris 2.x only</i>	Specifies which directories to search for libraries, where <i>dir_list</i> is a list of directories separated by colons. This switch is passed to the run-time linker. The -R switch overrides the LD_RUN_PATH environment variable if both are specified.
-S	Do not assemble the program but produce an assembly source file.
-temp=directory	Set directory for temporary files to be <i>directory</i> .
-traditional	Compile programs in old-style (K&R) C. This is the default compilation mode unless the configuration file has been modified for site-wide changes. This mode does not cause the clcc/inc directory to be searched.
-Uname	Remove any initial definition of the preprocessor symbol <i>name</i> .
-v	Verbose mode. Causes the name of each subprocess to be printed as it begins to execute.
-w	Do not print warning messages.
-w[n]	Suppress warning messages at level <i>n</i> and higher. The higher the level, the less important the messages are considered to be. 0 <= n <= 4 . -w0 is the same as -w .
-wide	Specifies multi-module inlining when you specify -wide with -Hiw . See the "What is inlining?" section on page 43 for more information. -wide causes the compiler to operate on all files being compiled as a collection rather than on each file individually. This is known as wide compilation. The switch also reduces compile time on systems lacking enough disk cache to store the compiler executables in memory.



Command-line switches

Table 2 CenterLine-C Command-Line Switches (Continued)

Switch	Description
-Xa	This causes a mixed mode of operation, favoring ANSI when a construct is encountered with an interpretation different for ANSI and pre-ANSI mode. The ANSI preprocessor is used unless -cpp is specified.
-Xt	Transition mode which supports ANSI and pre-ANSI features, but favors pre-ANSI semantics when a construct with different interpretation is encountered. The ANSI preprocessor is used unless -cpp is specified.
-Yl <i>dir</i> <i>Solaris 2.x only</i>	Insert directory <i>dir</i> into the compilation search path. The path will be searched for alternative versions of the compilation programs, such as as and ld . This switch is a synonym for -Qpath .

a. The **-ansi** switch requires strict ANSI C compliance. Some system header files contain constructs that are not ANSI-compliant and that will not compile with the **-ansi** switch. For example, the ANSI standard allows comments following **#else** and **#endif** directives, but it does not allow tokens following these directives. These files can be compiled with the **-Xa** or **-Xt** switch.





Chapter 3 Pragmaѕ and toggles

The CenterLine-C compiler provides pragmas to control inclusion and listing of source text, optimizations, warning levels, and toggle settings. Toggles are switches that turn compiler controls on and off.

This chapter lists the pragmas and toggles available.





Compiler pragmas

The CenterLine-C compiler provides pragmas (the term comes from Ada) that direct compiler operations. Pragmas control the inclusion and listing of source text, the production of object code files, the generation of optional additional program and debugging information, and so on. Each pragma has a default value.

How to specify pragmas

Compiler pragmas take one of the following general forms in your source:

#pragma *Pragma_name*

#pragma *Pragma_name*(*Pragma_parameters*)

or

pragma *Pragma_name*;

pragma *Pragma_name*(*Pragma_parameters*);

where *Pragma_parameters* is a list of constant expressions separated by commas. The number and types of the expressions depend on the particular *Pragma_name*. A pragma can appear anywhere a statement or declaration can appear. *Pragma_names* are not case sensitive.

NOTE The first syntax style above, **#pragma**, is the ANSI C standard syntax. The second form already existed in CenterLine-C prior to the Standard, and has the advantage that it can appear in macros. It must be terminated with a semicolon, and it can only appear in a statement or declaration context.

Table 3 shows the available pragmas.

Pragmas and toggles

Table 3 Pragmas

Domain Name	Pragma	Purpose
Toggles	On, Off, Pop	Turns On or Off, or reinstates a prior status of a compiler switch or “toggle”.
Optimizations	Loop_factor	Replaces a loop with the code that makes up the loop body, replicated some number of loop iterations.
Listings	Skip	Causes blank lines to be inserted into the listing. This pragma takes effect only when the List toggle is On.
	Page	Causes page ejects to be inserted into the listing. This pragma takes effect only when the List toggle is On.
	Title	Causes a title to appear at the top of each subsequent page. This pragma takes effect only when the List toggle is On.
Diagnostics	Warning_level	Determines which level of warning the compiler issues.

For information on optimization, see the “Optimizing program performance” section on page 25. For more information about listings, see the “Listings and cross references” section on page 49. For more information on diagnostics, see the “File I/O errors” section on page 79.

Compiler toggles

This section describes each of the compiler toggles. These are switches you can turn on or off to affect compilation in various ways. A toggle can be set on the command line with switch **-Hon** or **-Hoff**; the syntax is

```
clcc -Hon=Toggle_name [...]
```

or:

```
clcc -Hoff=Toggle_name [...]
```

Toggles can also be turned on or off with pragmas placed in source code or in a profile (see the “Compiler pragmas” section on page 17). The syntax is

```
#pragma On(Toggle_name)
```

```
#pragma Off(Toggle_name)
```

```
#pragma Pop(Toggle_name)
```

On turns the toggle on; **Off** turns it off; and **Pop** reinstates it to a prior value. Toggles operate in a stack-like fashion, where each **On** or **Off** is a “push” of on or off, and a **Pop** “pops” the stack. The stack for each toggle is at least 16 elements deep, but no diagnostic is given if the stack overflows or underflows. Here’s an example.

```
#pragma On (List) /* Turns on source listing. */
#pragma Off(List) /* Turns off source listing. */
#pragma Pop(List) /* Back to off for listing. */
#pragma Pop(List) /* Back to on for listing. */
```

The names, default values, and meanings of the compiler toggles are described in Table 4. Any of the defaults can be altered by making appropriate changes in the driver configuration file. Toggle names are *not* case sensitive.

Pragmas and toggles

Table 4 Compiler Toggles

Toggle Name	Default Value	Effect of Toggle
Back_substitute_epilog	Off unless optimization level ≥ 2	When On, the compiler replaces an unconditional jump to the epilogue of a function with the epilogue sequence.
Behaved	Off unless optimization level ≥ 4	<p>When On, this toggle specifies that the program is “well behaved”, and the optimizer can be less conservative in generating code for pointer-based objects. A well behaved program is one that follows these rules:</p> <ul style="list-style-type: none"> • The address of a union member is never assigned to a pointer. • A value of a pointer type is never cast to an incompatible pointer type. <p>With these assumptions, the compiler may be able to generate substantially better code in referencing pointer-based variables.</p> <p>The compiler issues an appropriate warning if either of the above assumptions is violated in such a way as to affect assumptions made by the optimizer. You must decide whether the warnings can be safely ignored or whether the program should be compiled at a lower optimization level.</p> <p>Note: The compiler may not catch all instances of misbehaved code. For example, a pointer to char may be passed to an undeclared external function expecting a pointer to int. Therefore, it is possible for a program to compile at optimization level 4 without warnings (and run incorrectly), but run correctly when compiled at a lower optimization level.</p>
Cleanup_spills	Off unless optimization level ≥ 3	When On, the compiler performs another iteration of local common subexpression elimination to clean up register spill code, if possible. See the “Local common subexpression elimination” section on page 31 for more information.

Table 4 Compiler Toggles (Continued)

Toggle Name	Default Value	Effect of Toggle
Cross_jump	On unless optimization level = 0	<p>When Off, suppresses the cross-jumping (tail-merging) optimization. Suppressing cross jumping can make code easier to debug at the instruction level.</p> <p>Note: Toggle Cross_jump is turned Off when switch -g or -O0 is specified.</p>
Global_CSE	Off unless optimization level >= 2	<p>When On, the compiler performs global common subexpression elimination. See the “Global common subexpression elimination” section on page 33 for more information.</p>
Globals_volatile	Off	<p>When this toggle is On, the optimizer does not move expressions with global or external variables out of loops.</p> <p>Note: Turning this toggle On does not have the same effect as declaring all globally or externally defined variables with the volatile type specifier. When you declare a variable with the volatile type specifier, the optimizer does not retain that variable’s value in a register across statement boundaries. The optimizer also updates volatile variables in memory as soon as the assignment occurs, and does not delete what appear to be “dead” or redundant assignments to these variables.</p> <p>Turn this toggle On if your program relies on signal functions to modify static or global variables. You can turn this toggle On and Off between function definitions.</p> <p>Code generated with this toggle is not portable. If you need portable code, you should declare variables volatile with the volatile type specifier instead of using this toggle.</p>
HP_ABI <i>HP only</i>	On	<p>Use registers for passing structures of eight bytes or less as parameters. When this toggle is On, parameters that are structures of eight or less bytes are passed and returned in registers according to HP calling conventions.</p>

Pragmas and toggles

Table 4 Compiler Toggles (Continued)

Toggle Name	Default Value	Effect of Toggle
Induction_analysis	Off unless optimization level ≥ 5	When On, the compiler analyzes loops, and in some cases readjusts the loop counter to eliminate the compare instruction, thus speeding up your program. See "Loop induction-variable elimination" on page 35 for a description.
List	Off	When On, the compiler writes a listing to standard output. List is typically specified at the start of compilation, but may appear in the source file to turn the listing On or Off around a particular section of source.
Live_dead_iterate	Off unless optimization level ≥ 2	When On, the compiler performs multiple iterations of live/dead analysis to further improve the code. See the "Live/dead analysis" section on page 31 for more information.
Local_CSE_iterate	Off unless optimization level ≥ 2	When On, the compiler performs multiple iterations of local common subexpression elimination to further improve the code. See the "Local common subexpression elimination" section on page 31 for more information.
Loop_reg_rename	Off unless optimization level ≥ 2	When On, the compiler performs lifetime register analysis to improve register allocation. See "Register lifetime analysis" on page 32 for more information.
Mem_refs_from_loop	Off unless optimization level ≥ 2	When On, the compiler attempts to replace variable memory references in loops with references to registers. See the "Loop memory reference elimination" section on page 32 for more information.
Optimize_for_space	Off unless optimization level = s	When On the code generator produces code sequences that minimize space, sometimes at the expense of execution speed.
Print_var_info	Off	When On, the compiler displays where each auto-, register-, or argument-class variable is mapped on a per-function basis. One of the following formats is used:

Table 4 Compiler Toggles (Continued)

Toggle Name	Default Value	Effect of Toggle
		<p>$N ==> R$</p> <p>Specifies that variable N has been mapped to machine register R.</p> <p>$N+offset ==> R$</p> <p>Specifies that a member of struct N has been mapped to machine register R.</p> <p>$N ==> FP + offset$</p> <p>Specifies that variable N has been mapped into the stack frame at $offset$ from base pointer FP.</p> <p>$N ==> <eliminated>$</p> <p>Indicates that variable N was eliminated via copy propagation.</p> <p>$N ==> <unreferenced>$</p> <p>Indicates that variable N was not referenced (or else all references were optimized away). The variable was not mapped.</p>
Read_only_strings	On with -ansi . Otherwise off.	<p>When On, all string constants are considered read-only and are placed in a read-only data section. Multiple occurrences of identical string literals within a single module appear only once in memory.</p> <p>Warning: Do not turn this toggle On if your program modifies the space occupied by a string literal. For example:</p> <pre>static char *p = "a string literal"; ... /* Actually modifies the string literal */ p[0] = 'A';</pre> <p>If this program is compiled with toggle Read_only_strings turned On, a protection fault occurs at run time.</p>

Pragmas and toggles

Table 4 Compiler Toggles (Continued)

Toggle Name	Default Value	Effect of Toggle
		<p>Note: String constants qualified with type <code>const</code> are unconditionally placed in the read-only data section:</p> <pre>extern void foo(const char *p); ... foo("This string is in read-only section.");</pre>
Recognize_library	On	When On, the compiler may substitute in-line code for any function defined in the ANSI standard C library. Candidates for substitution currently include the math and string functions.
Reduce_reg_contention	Off unless optimization level ≥ 2	When On, the register allocator attempts to limit the lifetime of registers to reduce spill code. See the "Enhanced register allocation" section on page 33 for more information.
Reg_alloc_enhance	Off unless optimization level ≥ 2	When On, the compiler attempts to improve register allocation. See the "Enhanced register allocation" section on page 33 for more information.
Source_code_order	Off unless optimization level = 0 or 1	When On, the compiler attempts to produce code in source order. This makes your code easier to debug at the assembly-language level. Note: This toggle is turned On when switch <code>-g</code> is specified.
Strength_reduce	Off unless optimization level ≥ 5	When On, the compiler performs loop strength reduction. See the "Loop strength reduction" section on page 35 for more information.
Tail_recursion	Off	When On, the compiler attempts to optimize recursive functions. See the "Tail recursion" section on page 36.
Trigraphs	Off unless <code>-ansi</code> specified	When On, this toggle enables trigraphs, that is, sequences of three characters that are converted to a single character. Trigraphs are only useful on machines that have limited character sets, as they allow you to represent characters that would otherwise be unavailable.



Chapter 4 Optimizing program performance

This chapter shows you ways to compile a program so it runs faster or requires less memory. In addition to describing the optimizations available at each level, it covers the following topics:

- *Specifying an optimization level*
- *Debugging optimized code*
- *Weighing execution speed against code size*
- *Loop unrolling*







Specifying an optimization level

You can activate some optimizations just by specifying the appropriate optimization level with command-line switch `-On`. Others must be invoked with a separate command-line switch, pragma, or toggle, instead of (or in addition to) specifying an optimization level (see the “Additional ways to optimize performance” section on page 36).

Keep in mind that good coding practice can make a greater difference in code size or execution speed than any amount of compile-time optimization.

Optimization levels are as follows:

0	for faster compilation
1 - 7	for generating faster code
s	for generating smaller code

Each higher numbered level specifies additional optimizations, and includes the optimizations performed at lower levels with a few exceptions. Level 7 is currently the same as level 6.

Use command-line switch `-On` to specify the optimization level, where *n* is the level number (0 through 7) or the letter *s*. If you specify just `-O`, you get level 2 by default. If you do not use `-O` at all, level 1 is the default. In general, the higher the optimization level, the faster the generated code runs, usually at the expense of compile time. If compile time is more important to you than the speed of generated code, specify `-O0`.





Optimizing program performance

Debugging optimized code

When you specify **-O0** or **-O1**, the compiler tries to generate code in source order. This makes your code easier to debug at the assembly-language level.

In general, optimizations make code harder to debug. Debuggers cannot usually determine when a variable's current value is in a register rather than in memory. Some of the effects that impact debugging are:

- Inlining often prevents debuggers from stepping through a program correctly.
- During debugging, cross jumping can result in surprising jumps that are not reflected in the source code.

Execution speed versus code size

Most CenterLine-C optimizations are intended to improve execution speed; some do so at the expense of code size. Some save code space, at the possible expense of execution time. A few save both time and space. You must decide which is more important to your application: faster execution or smaller code. If smaller code is more important to you, specify **-Os**.

Cross jumping

Cross jumping can *decrease time* by decreasing the size of a function and therefore increasing its chances of being held in an instruction cache. On the other hand, cross jumping can *increase time* by introducing extra branch instructions.

Inlining

Inlining may make function definitions longer, resulting in *increased* compile time. Too much inlining can increase the size of a function beyond the ability to fit in a cache. Also, space may be increased unless the inlined function contains no more code than would be generated for the calling sequence that is replaced.

Constants in code

Using constants in code can reduce page writes.



Table 5 shows which optimizations affect execution time and which affect code space, and notes possible disadvantages. A primary benefit is one that almost always happens (optimizations sometimes backfire). A secondary benefit is one that less frequently makes a difference. In a few cases, benefits and disadvantages may seem contradictory.

Table 5 Optimization Benefits

Optimization	Optimize Level	Primary	Secondary	Possible Disadvantages
Key: T= time, S= space, C= compile time				
cross jumping	1,s	S-	T-	T+
iterative local CSE	2,s	T-, S-		
iterative live/dead analysis	2,s	T-, S-		
loop memory-reference elimination	2,s	T-, S-		
register lifetime analysis	2,S	T-, S-		
enhanced register allocation	2,s	T-, S-		
global CSE	2	T-		S+
back-substitution of epilogue	2	T-		S+
improved spill analysis	3,s	T-, S-		
inlining	4	T-		S+, T+, C+
pointer-based vars "behaved"	4	T-		
loop strength reduction	5	T-		
loop induction-variable elimination	5	T-		S+
loop unrolling	6	T-		S+
tail recursion	n/a	T-		
constants in code	n/a	T-		
optimize for space	s	S-		T+



Optimizing program performance

Optimizations at each level

See Table 6 for a list of the optimizations in levels 0 to 7. Each level includes the optimizations of lower-numbered levels.

Table 6 Optimization Levels

Level	Optimizations
Level 7	Architecture-specific optimizations
Level 6	Loop unrolling Architecture-specific optimizations
Level 5	Loop strength reduction Loop induction-variable elimination
Level 4	Inlining Code assumed “well behaved”
Level 3	Low-level instruction scheduling Improved Spill Analysis
Level 2	Loop memory-reference elimination Spill analysis Register lifetime analysis Enhanced register allocation Global common subexpression elimination Back-substitution of epilogue code High-level instruction scheduling <i>Multiple iterations of:</i> Local common subexpression elimination Live/dead analysis
Level 1	Cross jumping (tail merging)
Level 0	Default optimizations <i>One iteration of:</i> Local common subexpression elimination Live/dead analysis





Optimizations at each level

Level 0 optimizations

This level includes default optimizations that you cannot disable. Level 0 also produces code in source order to make code easier to debug at the assembly-language level. Specify **-O0** if compilation speed is important to you.

Global register allocation

Operations using registers are often much faster than equivalent operations that reference memory locations. The compiler allocates registers to variables according to how frequently the variables are used. With global register allocation, the compiler may override explicit register declarations to achieve the most efficient usage.

Operator strength reductions

Sometimes an operation can be replaced by a more efficient operation (or series of operations) that yield the same result in less time. For example, multiplication by a constant can be converted to a series of additions, subtractions, and shifts. Division or modulo by an integer power of 2 can be converted to a shift or masking operation.

NOTE

Some strength reductions, such as those that replace multiplication by additions and shifts, result in larger code size.

Local common subexpression elimination

The compiler eliminates common subexpressions within extended blocks of code. An extended block is a section of self-contained code with one entry and one or more exits.

Constant propagation

When the compiler sees a reference to a variable whose last assigned value was a constant, the reference is replaced by the constant.

Constant expression folding

Arithmetic expressions whose operands are constants are evaluated at compile time. Constant propagation often exposes such expressions that are not otherwise apparent.

Eliminating unreachable ("dead") code

The compiler eliminates code that cannot be reached by any execution path. Such "dead" code may be created as a result of another optimization. For example, constant propagation may have converted a conditional jump into an unconditional jump, eliminating a path out of the associated block.

Live/dead analysis

Computations whose results are never used are eliminated. Assignments of values that are never used ("dead stores") are also eliminated.





Optimizing program performance

Level 1 optimizations

This level produces code in source order as in level 0, and includes the optimizations described below.

Cross jumping (tail merging)

Controls: toggle **Cross_jump**

Two or more basic blocks of code that end in the same sequence of code, and that have a common successor, can be rewritten to eliminate the redundant code. This is an effective space-saving optimization. The effect of cross jumping on execution speed depends on machine characteristics and the nature of your application: it could slow down code, or speed it up, or it may have no effect on speed. You can disable cross jumping by turning Off toggle **Cross_jump**.

Level 2 optimizations

This level includes cross jumping from level 1, and the optimizations described below.

Iterative local common subexpression elimination

Controls: toggle **Local_CSE_iterate**

The compiler always performs local common subexpression elimination once. With this optimization, local CSE is performed iteratively to further improve the code. See the “Local common subexpression elimination” section on page 31 for a description.

Iterative live/dead analysis

Controls: toggle **Live_dead_iterate**

The compiler always performs live/dead analysis once. With this optimization, the compiler performs live/dead analysis iteratively to further improve the code. See the “Live/dead analysis” section on page 31 for a description.

Loop memory reference elimination

Controls: toggle **Mem_refs_from_loop**

The compiler attempts to replace memory references to variables in loops with references to registers.

Register lifetime analysis

Controls: toggle **Loop_reg_rename**

Some disjointed loop variables are referenced in more than one loop but are “dead” on exiting a loop. The compiler tries to allocate such variables to different registers in different loops.





Optimizations at each level

Enhanced register allocation

Controls: toggle **Reg_alloc_enhance**, **Reduce_reg_contention**

The compiler has a number of heuristics available to determine how to best allocate registers. By default, the compiler always uses one heuristic to allocate registers. With this optimization, all the heuristics are analyzed and the best one is used.

The register allocator attempts to limit the lifetime of registers to free more registers and reduce spill code.

Global common subexpression elimination

Controls: toggle **Global_CSE**

The compiler eliminates common subexpressions across all basic blocks of a function. Also, any register computation that is invariant within a loop is moved above the loop.

Back-substitution of epilogue code

Controls: toggle **Back_substitute_epilog**

Whenever there is an unconditional jump to the epilogue of a function, the compiler replaces the jump with the epilogue sequence. This optimization increases execution speed at the expense of code space.

Level 3 optimizations

This level includes all the optimizations performed at level 2, plus the following.

Improved spill analysis

Controls: toggle **Cleanup_spills**

If the register allocator produces spill code, the compiler performs local common subexpression elimination again to eliminate any unnecessary spills.

Level 4 optimizations

This level includes all the optimizations performed at level 3, plus the following.

Inlining

Controls: switch **-Hi**

When a function or iterator is inlined, calls to it are replaced with the logic contained in the function. This speeds up execution, but code size is potentially larger. The substitution saves prologue, epilogue, and parameter-passing overhead, and exposes opportunities for additional optimizations such as copy propagation and constant folding.

You can select individual functions to be inlined, or allow automatic selection of functions according to criteria you provide.





Optimizing program performance

For manual selection, use type qualifier **_Inline** to cause a particular function to be inlined (you must also specify **-Hi** on the command line). For example:

```
_Inline void f() {
    ...
}
```

For automatic selection, use the command-line switches in Table 7 to specify selection criteria. These switches and others available for inlining are described in greater detail in the “Inlining routines” section on page 41.

Table 7 Inlining switches

Switch	Description
-Hia	Functions that have their address taken are not inlined
-Hib=<i>n</i>	Temporary-file buffer size is set to <i>n</i> .
-Hic=<i>n</i>^a	Functions called less than <i>n</i> times are inlined.
-Hih	Status of inlined or defined routines is listed.
-Hin	Nested routines are not inlined.
-Hir	Recursive functions are not inlined.
-His=<i>n</i>	Functions with stack size greater than <i>n</i> bytes are <i>not</i> inlined. You must specify a value for <i>n</i> .
-Hit=<i>n</i>^a	Functions with less than <i>n</i> tree nodes are inlined. There is roughly one tree node for each operator, operand, or other language construct such as a statement, loop, declaration, or function.
-Hiu	Mnemonic inliner-invented names are used for multi-module inlining.
-Hiw	The inliner is invoked in multi-module mode.
-Hix	Exported functions are not inlined.

a. We provide a default value for *n* in the **clcc.cnf** file. The default value is used only when the optimization level is ≥ 4 and you do not specify a value for *n* on the command line





Pointer-based
variables

Controls: toggle **Behaved**

The compiler can be less conservative in generating code for pointer-based objects in a program that follows these rules:

- The address of a union member is never assigned to a pointer.
- A value of a pointer type is never cast to an incompatible pointer type.

With these assumptions, the compiler may be able to generate substantially faster code in referencing pointer-based variables. This toggle works at any level; the default is On at level 4. See Table 4 on page 20 for more information.

**Level 5
optimizations**

This level includes all the optimizations performed at level 4, plus the following.

Loop strength
reduction

Controls: toggle **Strength_reduce**

In the body of a loop, multiplications and array-indexing operations involving the loop counter can be replaced by additions. This optimization usually speeds up operations that are performed on every iteration of the loop. However, it could actually slow down execution if rarely executed multiplications (as in a switch statement) are replaced by additions that are always executed.

Loop
induction-variable
elimination

Controls: toggles **Strength_reduce**, **Induction_analysis**

A loop counter that varies by a constant and whose value is tested to determine when to exit a loop is called an induction variable. If the induction variable is used only to test for loop termination, and not in any other context, it is readjusted to terminate the loop when its value is 0. Your program speed is increased because the compare instruction is eliminated.

Toggle **Induction_analysis** automatically turns **On** toggle **Strength_reduce**. You can turn **On** toggle **Global_CSE** to make more loops available for induction analysis.





Optimizing program performance

Level 6 optimizations

This level includes all the optimizations performed at level 5, plus the following.

Loop unrolling

Controls: pragma **Loop_factor**, switch **-Hunroll**

Loop unrolling (or loop unwinding) involves replacing a loop with the code that makes up the loop body, replicated for some number of loop iterations. This optimization always increases code size. Loop unrolling is discussed in detail in the “Loop unrolling” section on page 38.

Level 7 optimizations

The optimizations at level 7 are currently the same as those at level 6.

Level s optimizations

Level s includes one space-saving optimization plus several time- and space-saving optimizations that also appear at other levels.

Optimizing for space

Controls: toggle **Optimize_for_space**.

The code generator produces code sequences that minimize code space, sometimes at the expense of execution speed.

Other space-saving optimizations

Other level s optimizations include:

- Cross Jumping (Tail Merging) -- Level 1 (see on page 32).
- Loop Memory-Reference Elimination -- Level 2 (see on page 32).
- Enhanced Register Allocation -- Level 2 (see on page 33).

Additional ways to optimize performance

This section describes optimizations that are independent of the optimization level. You must explicitly request these optimizations via command-line switches, pragmas, or toggles. Each description shows the controls you use to invoke the optimization.

Tail recursion

Controls: toggle **Tail_recursion**

If a function calls itself recursively, and the call is the last action the function performs, the code can be modified to branch to the beginning of the function instead of calling it again, with appropriate adjustments for parameter passing and return. When you turn On toggle **Tail_recursion**, the compiler optimizes any recursively called function that can be modified this way.





Constants in code

Controls: toggle **Read_only_strings**

Static variables (exported or local) declared with the **const** qualifier can be mapped to the code area. Lengthy literals can also be placed in code space rather than data space. This can be beneficial where the operating system performs dynamic code loading. In such circumstances code is most often read-only, so literals can be swapped out of memory without the need to write them to the paging medium used by the operating system. Also, when data space is at a premium, this is a useful way to shift potentially large amounts of program from data to code.

When toggle **Read_only_strings** is **On**, all string constants are assumed to be read-only. This permits the compiler to place them in read-only sections, thus reducing system overhead when more than one process is executing the same code.

NOTE String constants qualified with the type **const** are unconditionally placed in the read-only data section.

Multi-module inlining

Controls: switches **-Hiu**, **-Hiw**, **-wide**

You can specify switch **-Hiw** to make the inliner operate on a group of files being compiled as a collection, rather than on each file individually. This is known as multi-module inlining or wide inlining. When you are using separate source files for data-hiding purposes or because more than one programmer is working on a project, inlining a function from one module into another can markedly improve performance. Also, benchmark results may be much improved where a benchmark is split across multiple files.

To achieve multi-module inlining you must also specify **-wide**, for "wide" compilation. See the "Predefined macros" section on page 4 for more information about this switch. See "Inlining routines" on page 41 for more information about multi-module inlining.

Compiler intrinsics

Controls: toggle **Recognize_library**

Calls to several library functions are replaced by fast inline code. The code generated for compiler intrinsics is fast because the compiler generates object code for these functions directly, saving the overhead of the function call, and tailoring the code to the "arguments." The CenterLine-C compiler automatically recognizes ANSI C functions such as **abs()**, **fabs()**, and **memcpy()** and inlines them.





Loop unrolling

This section describes an optimization called loop unrolling (or loop unwinding). Loop unrolling is most effective on machines with a pipelined architecture.

If the next iteration of a loop does not rely on results of the prior iteration, two or more iterations can execute at the same time. This works only for loops with *constant stride*: those whose loop counter is incremented by a compile-time constant for every iteration. Only a small saving in execution time may be realized on machine architectures that do not have pipelined instructions. On machines with an instruction cache, loop unrolling can actually degrade performance if the unrolled loop exceeds the size of the cache.

Selecting loops for unrolling

Use pragma **Loop_factor** to specify criteria for selecting loops to be potentially unrolled. The syntax is:

```
#pragma Loop_factor(loop_count,size)
```

where *loop_count* is the maximum number of times a loop will be unrolled, and *size* is the upper limit on the size of loops to be considered for unrolling. *size* refers to the number of intermediate-language instructions generated by the compiler; this number corresponds loosely to the number of individual machine instructions.

You can specify **#pragma Loop_factor** any number of times, with different values for *size*. You can have any number of loop factors in effect at the same time, each with a different *size*; however, a pragma with the same *size* as a previous pragma replaces that previous pragma. The default for *loop_count* is 4, and the default for *size* is 64.

To disable a previously specified pragma, use

```
#pragma Loop_factor(0,size), where size is the same as specified in the pragma you want to disable.
```

Setting default for number of times loops are unrolled

If you specify **#pragma Loop_factor**(*n*,0), *n* becomes the default number of times a loop is unrolled regardless of *size*. If you also specify (for instance) **#pragma Loop_factor**(*x*,100), loops with less than 100 intermediate instructions will be unrolled *x* number of times, and loops with more than 100 intermediate instructions will be unrolled *n* number of times.





You can also specify loop unrolling on the compiler command line, with switch `-unroll=loop_count,size` (see Table 2 on page 9). For example:

```
clcc cwhtd.c -unroll=3,20 -unroll=4,10
```

is equivalent to:

```
#pragma Loop_factor(3,20)
#pragma Loop_factor(4,10)
```

How loops are unrolled

The loops to be unrolled must have certain characteristics, as well as meeting the selection criteria you specify with `#pragma Loop_factor`. To be a candidate for unrolling:

- The loop control variable must not be modified in the body of the loop.
- A loop must have constant stride. That is, the controlling loop variable must increase or decrease by a constant amount each time around the loop.
- The resulting *loop_count* must be greater than 1. Otherwise, the loop is not unrolled.

Number of times a loop is unrolled

The loop count determines how many times a loop is unrolled. A general loop, that is, a loop for which the number of iterations is not known, is unrolled *loop_count* times, and then a remainder of the iterations (up to *loop_count* - 1) are executed by a following loop that is not unrolled.

A constant loop (one where the number of iterations *n* is known) that has less than *loop_count* iterations is unrolled *n* times, and no loop-test conditional jumps are performed. The same happens for constant loops with more than *loop_count* iterations but less than *loop_count* * 2 iterations. The following example illustrates what may happen for a constant loop with more than *loop_count* * 2 iterations.

Suppose you specify:

```
#pragma Loop_factor(3,100)
```

and the following for loop occurs somewhere in your program:

```
for (i=1; i<=8; i++) {
    /* loop body */
}
```





Optimizing program performance

If code generated for the body of the loop amounts to less than 100 intermediate instructions, the loop may be unrolled as follows:

```
for (i=1; i<=6) {  
    /* loop body */  
    i + 1  
    /* loop body */  
    i + 1  
    /* loop body */  
    i + 1  
}  
/* loop body */  
i + 1  
/* loop body */  
i + 1
```





Chapter 5 Inlining routines

This chapter describes an optional compiler pass called an inliner of functions or routines. An inliner replaces calls to routines with the logic contained within the routines, thus avoiding the overhead of routine calls at the expense of potentially larger code size.







What is inlining?

An inliner replaces calls to routines with the logic contained within the routines, thus avoiding the overhead of routine calls at the expense of potentially larger code size.

Selective inlining

You can specify particular routines to be inlined, and also allow the inliner to select automatically routines that satisfy adjustable criteria, such as the number of calls and the size of the routine.

Inlining saves overhead

The inliner back-substitutes routine bodies where the routines are called, with parameters appropriately replaced with local variables or constants. This saves at least the routine prologue, epilogue, and parameter-passing overhead, and exposes opportunities for additional optimizations, such as copy propagation and constant folding, especially when the arguments themselves are constants.

Inlined routines may be defined

Factors such as recursivity, addressability, and exportedness play roles in determining whether an inlined routine must also be defined. If the routine can be completely inlined, it is not defined. See "How routines are selected for inlining" on page 48 for details.

Operationally, the inliner sits between the compiler's front end and the common optimizing back end.





Inlining routines

Invoking and activating the inliner

You specify inliner switches on the command line to invoke the inliner during compilation.

Using special Inlining switches

Use one or more of the **-Hix** switches described in the next section to activate the inliner to

- Achieve automatic inlining
- Prevent selected categories of routines from being inlined
- Customize the inlining process

Using `_Inline` for individual functions

If you specify just **-Hi** without a suffix, you must use the type qualifier **`_Inline`** in your code to request inlining for individual functions. For example:

```
_Inline void f() { ... }
```

causes `f()` to be inlined when you compile with the **-Hi** switch. Also, you can explicitly specify **`_Inline`** for a function, to override any **-Hix** switch that might otherwise exclude that function from inlining.





Switches for inlining

Table 8 describes each of the inliner switches in detail.

Table 8 Switches for Inlining

Switch	Description
-Hi	Invoke the inliner during compilation. Use this switch when you want inlined only those routines you designate individually with the type qualifier _Inline , and when you are not specifying any other -Hix switch on the command line.
-Hia	Do not inline routines that have their address taken. Such routines may be inlinable, but still must be defined so that they will have an address.
-Hib=<i>n</i>	Specify temporary-file buffer size. The buffer size for the temporary files is set at 65,536 for 32-bit compiler hosts. You can override this by specifying <i>n</i> as something different, for example, smaller to save buffer space. The large buffer size is used because the program does inordinate amounts of seeking/reading through the temporary files.
-Hic=<i>n</i>	Inline routines called less than <i>n</i> times. When you specify -Hic (and <i>n</i> is not zero), routines that are directly called less than <i>n</i> times are automatically inlined. If you specify -Hic and -Hit (with nonzero values), they work together, that is, both conditions must be met before a routine is automatically inlined.
-Hih	Show which routines have been defined or inlined. The -Hih switch prints to standard output a history detailing which routines were defined, which were inlined, and which were cloned. The static nesting level and parent ID (a compiler-assigned number) are printed along with the ID and name of the routine, the size in tree nodes, and the number of calls. Following each entry is a list of status flags for the routine. The status flags are: <ul style="list-style-type: none"> DEF Routine must be defined. INL Routine may be inlined. CLO This is a clone of a previously defined routine. UPL Routine makes up-level references. REC Routine is recursive. ADR Routine's address is taken. ALO Routine contains calls to alloca().



Table 8 Switches for Inlining (Continued)

Switch	Description
	For the purposes of -Hih , a recursive routine <i>R</i> is one for which inlining is requested, and is such that a call to <i>R</i> is produced in the process of inlining <i>R</i> . A routine for which inlining is not requested is not regarded as recursive. If a routine <i>F</i> calls <i>G</i> and <i>G</i> calls <i>F</i> , but only <i>F</i> is requested to be inlined, neither is marked as recursive because the inlining of <i>F</i> does not produce a call to <i>F</i> : it produces only a call to <i>G</i> .
-Hin	Do not inline nested routines. Routines that are statically nested within inlined routines are usually inlined. You can suppress this with -Hin . However, suppressing it may cause nested routines to be cloned, resulting in a new instance of each nested routine for each inline copy of the parent.
-Hir	Do not inline recursive routines. If a directly or indirectly recursive routine (see the -Hih switch for a definition) meets the criteria for inlining, it is inlined only at the first level and then a definition is generated so that it can be called at subsequent levels. You may want to avoid inlining recursive routines; the -Hir switch suppresses inlining of recursive routines even at the first invocation level.
-His=<i>n</i>	Inline routines with stack size less than <i>n</i> bytes. When you specify -His (and <i>n</i> is not zero), only routines with less than a stack size of <i>n</i> bytes are automatically inlined.
-Hit=<i>n</i>	<p>Inline routines with less than <i>n</i> tree nodes. When you specify -Hit (and <i>n</i> is not zero), routines with less than <i>n</i> expression-tree nodes are automatically inlined. There is one tree node, roughly, for each operator, operand, or other language construct, such as a statement, loop, declaration, or routine.</p> <p>We provide a default value for <i>n</i> in the <code>clcc.cnf</code> file. The default value is used only when the optimization level is ≥ 4 and you do not specify a value for <i>n</i> on the command line.</p>
-Hiu	<p>Construct unique names for inter-module communication. For multi-module inlining, the inliner may invent public names to communicate private data or code from one module to another. You may encounter these names when you use a debugger (although debugging inlined code is not recommended). The names are constructed as described below.</p> <p>When function <i>f</i> in one module <i>M1</i> that uses a data item <i>D</i> is inlined into a separate module <i>M2</i>, <i>D</i> must be available to <i>M2</i>. If <i>D</i> is private to <i>M1</i>, the inliner must make it public so that <i>M2</i> can access it. For example:</p> <pre> m1.c: static int i = 7; m2.c: main () { sub(int j){i+=j;} sub (1); } </pre>

**Table 8** Switches for Inlining (Continued)

Switch	Description
	<p>Here i must be available to the body of main() in <i>M2</i> when sub() is inlined. The inliner constructs a unique name for i and makes it public; that name is then used within both <i>M1</i> and <i>M2</i>. The name is usually of the form <code>__inl_</code> followed by V, F, or L, and followed by a unique 32-bit hexadecimal integer computed from module <i>M1</i>, and followed by an integer index. V stands for variable, F for function, and L for a private compiler-generated data item such as a string literal.</p> <p>In the example above the private variable i might be translated into the public variable <code>__inl_Vbd9755ee3</code>. Here bd9755ee is the checksum and 3 indicates variable number 3 in the inliner's internal variable table. The unique integer is computed from the compiler front-end output for <i>M1</i>.</p> <p>If you specify switch -Hiu, the inliner constructs a unique name that can be easily traced back to the defining module. With -Hiu, inl is replaced by the original variable/function name and is followed by the module defining the variable/function. In the case of private compiler-generated data, inl is retained and the module name is added. With -Hiu, the private variable i of the example above would become <code>__i_m1_Vbd9755ee3</code>.</p>
-Hiw	<p>Inline multiple modules. With this switch, the inliner operates on all files being compiled as a collection rather than on each file individually. This is known as multi-module inlining or wide inlining. Multi-module inlining permits inlining a function from one module into another module. This can improve performance when you use separate files for data hiding, or when more than one programmer is working on a project.</p> <p>Benchmark performance can also be improved in cases where a benchmark is split across multiple files. For example, you can realize a significant performance improvement in the Dhrystone 2.1 benchmark with:</p> <pre>dhry1.c dhry2.c -Hiw -Hit=250 -wide</pre> <p>To achieve multi-module inlining you must also specify -wide, for “wide” compilation. See “Command-line switches” on page 7 for more information about this switch.</p>
-Hix	<p>Do not inline exported routines. Exported routines may be inlinable, but still must be defined in order to be exported. The -Hix switch suppresses inlining of exported routines.</p>





How routines are selected for inlining

The switches for determining whether a routine is inlined automatically are overridden if a routine is explicitly designated as inlined with the **`_Inline`** type qualifier.

The **`-Hit`** and **`-Hic`** switches work together. If both are specified (and are not zero), then both conditions must be met. Otherwise, if the specified condition is met, the routine is inlined.

Routines chosen for inlining that are exported, recursive, or have their address taken are inlined but also defined. Routines chosen for inlining that are not exported, not recursive, and do not have their address taken, are inlined but the definition is discarded. Recursive routines are inlined only once and then a call is generated.

Debugging inlined routines

If you specify **`-g`** on the command line, the inliner is suppressed. Reasonable debugging is not possible in the presence of inlined functions, especially when you use multi-module inlining.





Chapter 6 Listings and cross references

This chapter describes the listings and cross references the compiler can generate, and explains how to control the format of the output.





Generating listings and cross references

To get a listing of your source code, specify switch **-Hlist** on the command line (see "Command-line switches" on page 7), or turn On toggle List (see "Compiler toggles" on page 19). The listing goes to standard output unless you specify a different destination as a parameter to **-Hlist** on the command line.

To get a cross reference, use the command **clccxref**, described in the "Using the clccxref command" section on page 54.

Pragmas Page, Skip, Title

You can use pragmas to specify a title for your listing, to control the number of lines per page, and to insert blank lines in the listing.

Page ejects

To cause n page ejects at some point in the listing, insert:

```
#pragma Page( $n$ )
```

where n is the number of form feeds, or pages to eject. Ordinarily you would set n to 1 to get a page break.

Blank lines

To generate n blank lines at some point in the listing, insert:

```
#pragma Skip( $n$ )
```

where n is the number of blank lines.

Title at top of page

To get a title at the top of each successive page, place the following pragma in the source:

```
#pragma Title(Listing_title)
```

where *Listing_title* is the title to be printed.

Each successive **Title** pragma changes the title for subsequent pages. The only way to title the first page is to place the **Title** pragma in the profile and use **#pragma On(List)** at the end of the profile, or wherever the listing should start in the source file. When you request a listing by specifying switch **-Hlist** on the command line, the first page is not titled, because the listing starts before the compiler sees the **Title** pragma.



Listing format

This section explains the format of the listing generated with **-Hlist**.

Ruler

The first line on each page, after any header and title lines, is a “ruler” that defines three fields:

- 1 Level number (`Levels`)
- 2 Line number (`LINE #`)
- 3 Line contents.

The ruler appears as follows:

```
Levels LINE# |-----1-----2-----3-----4-----5-----...
```

Level numbers

Level numbers can be used to find a missing closing brace (`}`) or comment terminator when a message such as “Unexpected end-of-file” is produced by the compiler. All three level numbers are initially zero, but they are printed as blanks rather than 0.

Scope nesting level

The first level number indicates the scope nesting level for **struct** or **union** declarations.

Statement nesting level

The second level number indicates the statement nesting level. It is incremented at the beginning of each opening brace and is decremented at the corresponding closing brace.

Structure initialization nesting level

The third level number indicates the structure initialization nesting level. It is incremented at the beginning of each opening brace and decremented at the corresponding closing brace.

Include files

A first-level include file named **File_name** is shown as starting after a line with **+(File_name** in the line-number field, and ending just before a matching **+)File_name** line. The included lines have **+** in the leftmost column of the line-number field, and those lines are numbered independently of the main source file.

An included file inside an include file has an extra **+** on each line for each level of inclusion, except that line numbers take precedence over **+**'s in the line-number field if the **+**'s would otherwise intrude into that field.





Cross-reference features

Cross references have the following features:

- Source-file line numbers

All cross-reference information refers to line numbers in the files compiled, as opposed to line numbers within a listing. Therefore no listing is necessary to use the cross reference.

- Processing of include files

Included source files are handled properly. That is, they do not interfere with the process, and their names are included correctly in the results.

- Assignment distinguished from use

References that assign values into variables are distinguished from references that use values of variables.

- Optional annotated listing

It is possible to generate an annotated source listing of one or more program files. The listing contains cross-reference information to the right of the source text.

- Multiple modules

A cross reference can span multiple compilation units by cross-referencing many modules at once and showing references from one module into the other. Thus, a single cross reference can be produced for a program that is broken up into separately compiled modules.

- Inter-module usage summaries

A list of the names that one module uses that are located in other files can be produced, organized by file. This can make it easier to understand the module interconnectivity of a large program.





Using the `clccxref` command

The `clccxref` command processes one or more CenterLine-C source files and produces a cross-reference listing on standard output. The listing consists of up to four components as described in "Cross-reference format" on page 58.

NOTE The `clccxref` command is not available on all platforms.

`clccxref` is a driver program that first invokes the compiler with switch `-xref`, to generate cross-reference information from C source files, and then invokes program `xref` to format and display the information. If any of the source files contain errors, appropriate diagnostics are generated. `clccxref` invokes the compiler on each of the source modules, and invokes the cross-reference processor `xref` on the concatenation of the result.

The command has the following form:

`clccxref` [*switches*] [*preprocessor_switches*]... *files* ...

where:

<i>switches</i>	includes zero or more command-line switches (listed below).
<i>preprocessor_switches</i>	Denotes zero or more preprocessor switches (for example, <code>-Idir</code> or <code>-Dname</code>) that are required when compiling the files.
<i>files</i>	Denotes one or more C source files.

The switches for `clccxref` are shown in Table 9.



**Table 9** clccxref Command-Line Switches

Switch	Description
-i	Expand include files in the listing. Ignored if used with -p .
-l	Generate a listing of the source files, annotated with cross-reference information. Include files are not expanded in the listing unless -i is also specified. Ignored if used with -p .
-m	For each module M , produce a listing of the names referenced in M that are defined elsewhere.
-p	Invoke an outboard C preprocessor for each source file, and process the output of the preprocessor instead of the source files themselves. If you do not specify -p , the inboard preprocessor which is part of the compiler is used. The -l and -i switches are ignored when used in conjunction with -p .
-s	Generate statistics related to the cross reference.
-u	Include in the cross reference names that are declared but not referenced. These names do not appear in the cross reference unless the -u switch is specified.





Cross-reference pragmas

You can use pragmas to control the cross-reference output. These pragmas must be specified in a profile called **xref.pro**, which you must create and edit. The pragmas have the same form as CenterLine-C compiler pragmas, except that strings are specified with single quotes (') instead of double quotes ("). The syntax is:

```
#pragma Pragma_name(Xref_parameters)
```

The following *Pragma_names* are supported.

On, Off, and Pop

On, **Off**, and **Pop** act identically to those for the compiler; see the "Compiler toggles" section on page 19. Supply one of the following toggles when you specify pragma **On**, **Off**, or **Pop**:

Annotate_includes Turn **On** this toggle to get an annotated listing of all files involved, not just files that are modules. This toggle is equivalent to the switch **-i**.

Annotated_listing Turn **On** this toggle to get an annotated listing of all modules, but not "header" files. See the description of the annotated source listing below. This toggle is equivalent to the **-l** switch.

List_module_usage Turn **On** this toggle to get a listing for each module *M* of the names used by *M* that are declared in other files. See the description of the module name list below. This toggle is equivalent to the switch **-m**.

List_unused_includes Normally the cross referencer does not list any names found in compiled include files that were never referenced. This default can be overridden by turning **On** toggle **List_unused_includes**. Also, the cross referencer never lists unused intrinsic names. That cannot be overridden. This toggle is equivalent to the **-u** switch.

Statistics Turning **On** this toggle causes a summary of cross-referencer statistics to be produced at the end. This toggle is equivalent to the **-s** switch.



**Columns**

The parameters for the **Columns** (*Declared_name*, *Info*, *Refs*, *Right_margin*) Pragma are integer values that describe the format of the cross-referencer output:

Declared_name Column where declared names begin.

Info Column where related information begins.

Refs Column where references to declared names begin.

Right_margin Last column on page (determines page width).

The default is **Columns(12, 60, 90, 132)**.

Include

The parameter for **Include** (*File_name*) is a string denoting a file to be included





Cross-reference format

Each cross reference consists of four components: an object-name cross reference, a file table (if more than one file is involved), a module name list (optional), and an annotated source listing (optional).

Object-name cross reference

An object-name cross reference is an alphabetized list of all names declared in the program, together with an ordered list of all the references to each name. This list presents the following information for each distinct name in the program:

- The line and column number of the declaration of the name. If the name occurs in a compiled include file, or if several modules are being cross-referenced, the file number is also given.
- The declared name *N*, and its *owner*: the name of the function that contains *N*'s declaration.
- Information about the named object, such as its storage class (**static**, **extern**, **typedef**, **register**, and so forth) and in some cases, the object's type.
- The numbers of any lines containing references to the name. If the references are not in the module being cross-referenced (they may be in an included file), or if several modules are being cross-referenced, the line numbers are presented in the format **fn<I>**, where *n* is the number of the file containing the references and *I* is the list of line numbers. Occasionally the entry in this field is of the form resolved at *ref* where *ref* is a line number or **fn<...>** reference as just described. This means that the name was introduced by an **extern** declaration whose actual definition was given at *ref*.
- References that assign, or may assign, a value to a variable are marked with the character *****.



**File table**

A file table is an alphabetized table of all files used in the program, with a file reference number for each. This table is produced if the cross reference involves more than one file; this happens if more than one module is cross-referenced, or if any compiled include files were involved in the modules being cross-referenced. The file table shows the correspondence between file numbers and file names. References in the object-name cross reference and the list of module names use the file number rather than file names, to keep the listing brief. Use the file table to determine the corresponding file name.

Module name list

A module name list is a list for each module *M* of all the names used by *M* that are defined in other files that are included in the cross reference. This list is produced if the **-m** switch is specified. The list is organized by file, and is useful for determining the interconnectivity between modules.

In the object-name cross reference and the module name list, a reference to a name *N* declared at reference point *P* is changed to a reference to a point *P'*, if the definition at *P'* resolves the declaration at *P*. Typically this happens when *N* is declared in an interface file *F*, is used in a module *M*, and is defined at *P'* in a module *M'*. The module usage in the module name list shows that *M* refers to *P'* in module *M'*, not *P* in interface file *F*. That is, you get references to the implementations rather than the interfaces through which they were supplied.

A module name list is generated only if the sharing of names between modules is done through interface files. The following example shows files that do not share the names **i1** and **i2**, because **i1** and **i2** are not declared in a common interface file.

```
m1.c:  int i1; extern int i2;
      main () { i1 = i2; sub(); }

m2.c:  int i2 = 4; extern int i1;
      sub () {printf("%d %d\n",i1,i2);}
```

In the following example, the modules do share the names, and **i1** and **i2** will be listed in the module name list. The cross referencer is unable to determine that two modules refer to the same name unless they both refer to the name in the same interface file.





Listings and cross references

```

glob.h: extern int i1,i2;

m1.c:  #include "glob.h"
        int i1;
        main () { i1 = i2; sub(); }

m2.c:  #include "glob.h"
        int i2 = 4;
        sub () {printf("%d %d\n",i1,i2);}

```

Annotated source listing

An annotated source listing is an annotated cross reference for each module. This listing is produced if the **-l** switch is specified. The result is a line-numbered listing of the source of the compiled program, with each line annotated on the right with the line numbers of the definitions of names used on the line.

If *n* names are used on the line, *n* line numbers appear to the right of the line, corresponding positionally. A line number alone is a reference into the file being listed. If the letter **i** appears instead, the name referenced is an intrinsic, such as **_find_char** or **_abs**. Finally, a line number followed by **f** and another number means that the name was declared in a file other than the one being listed; the file number can be used to discover that file's name in the file table. For brevity, **Line#File#** is used instead of **File#<Line#>** as in the object-name cross reference.

Distinction of file names

In a multi-module cross reference, a particular interface file may have been included by several modules, because each of the modules being cross-referenced needs the resources in that file. The cross referencer assumes that a repeated declaration of a name in a compiled include file is the same declaration if it appears at the same line and column number of the same include file.

For purposes of determining "sameness of include files" the cross referencer uses the text of the file name, including the path. Therefore, to cross-reference several modules successfully, do not use different names for the same include file.

For example, if module *M1* includes **../utils/trees** and *M2* includes **/prog/utils/trees**, and if these two references denote the same file, the cross referencer will not recognize them as the same.





Chapter 7 System specifics

This chapter describes some system-specific aspects of the CenterLine-C compiler. ANSI document X3.159-1989 requests that each C implementation provide the system specifics described here.





Identifiers

There is no limit on the number of significant characters in an identifier, with or without external linkage.

Data types

This section describes how data types are implemented.

Characters

Source and execution character sets are both Standard ASCII. Each character in the source character set maps into the identical character in the execution character set. Without exception, all character constants map into some value in the execution character set.

A character is stored in a byte, and there are four bytes in an **int**.

The type specifier **char**, when not accompanied by an adjective, denotes a signed character type.

Integers

Integers are represented in twos-complement binary form. Table 10 illustrates the ranges of values to which the various integer types are restricted.

An unsigned integer is converted to a signed integer of the same size by transferring the bits from one to the other. All of the bits are retained. If the value of an unsigned integer is larger than the largest positive signed integer value, it is interpreted bit by bit and yields a negative value.

The result of a bitwise operation on a signed integer is the same as if the integer were treated as unsigned.

The sign of the remainder on integer division is the same as the sign of the dividend.

The right shift of a signed integral type is arithmetic; that is, the sign bit is propagated to the right.

System specifics

Table 10 Integer Ranges

Type	Minimum Value	Maximum Value
signed char	-128	127
unsigned char	0	255
short	-32,768	32,767
unsigned short	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,296
long	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,296
long long^a	-2^{63}	$2^{63}-1$
unsigned long long^a	0	$2^{64}-1$

a. Not supported on HP platforms.

Floating point numbers

Floating-point representation is IEEE Standard 754. The default rounding mode is “round to nearest.” See the “Size and alignment of data types” section on page 69 for the length required for each floating-point type.

When a negative floating-point number is truncated to an integral type, the truncation is toward zero. Thus -2.7 is truncated to -2 and -1.2 to -1 .

CenterLine-C uses the IEEE Standard 754 formats to represent floating-point data. Each floating-point data type has a sign bit, exponent bits, and mantissa bits. The most significant bits are on the right. Table 11 shows the size of each data type.

For details about the IEEE data format used, please refer to the ANSI/IEEE standard 754/1985 documents.

Table 11 Size of Floating-Point Data Type

Precision	Sign Bit	Exponent Bits	Mantissa Bits
float	1	8	23
double	1	11	52
long double ^a	1	15	112

a. **long doubles** are treated as **doubles** on SunOS 4.x platforms.

Pointers

The difference of two pointers is type **int**.

Structures, unions, and bit fields

Signed and unsigned bit fields are supported. A bit field declared as **int** is treated as **unsigned int**. A bit field declared as **signed int** will be interpreted as signed. For more information on structures, unions, and bit fields, see "Storage mapping" on page 67.

Preprocessing directives

A single-character constant in a constant expression controlling conditional inclusion is always non-negative in value, ranging from 0 to 255.

Debugger support

The **-g** switch directs the compiler to generate debugger information. It should not be used with the **-On** switch because the optimizations can confuse the debugger. Most level 0 optimizations are performed anyway. However, stores are not delayed nor dead stores removed when you specify **-g**.





Chapter 8 Storage mapping

When you interface C code with assembly language or with code written in other languages, you may need to know how data is mapped in the modules you are compiling. This chapter describes data types and storage classes.





Size and alignment of data types

Some data types have a fixed size and byte alignment, while others may vary. Table 12 summarizes the size and alignment of various C data types. **char** and **int** types have the same size regardless of whether they are signed; therefore the table does not mention the sign.

Table 12 C Data Types: Size and Alignment

Data Type	Size	Alignment
char	1 (bytes)	1 (bytes)
short int	2	2
int	4	4
long int	4	4
long long int ^a	8	8
float	4	4
double	8	8
long double	8 or 16 ^b	8
enum	4	4
Pointer	4	4
Full-function ^c	8	4
T[n]	$n * \text{sizeof}(T)$	Same as T

a. Only on Sun platforms

b. 8 on SunOS 4.x platforms and HP platforms, 16 on Solaris 2.x

c. A full-function value is a CenterLine-C extension. It consists of a function address and a static link, and is akin to a Pascal routine.

struct and union types

The size of a **union** is the size of the biggest member, padded so that its size is evenly divisible by its alignment. The size of a **struct** is the sum of the sizes of its members, padded between members so that its size is evenly divisible by its alignment.

A **struct** or **union** is aligned according to the most stringent requirements among its members.



Arrays

Arrays are stored in row-major order. This means that the last subscript in a multi-dimensional array varies fastest.

For example, this array:

```
int x[4][4]
```

is stored as:

```
x[0][0],x[0][1],x[0][2],x[0][3],...,x[3][3]
```

Bit fields

Both signed and unsigned bit fields are supported. By default, bit fields are unsigned, that is, **int x:11**; specifies an unsigned bit field and

long x:19; specifies an unsigned long bit field. Use the adjectives **signed** and **unsigned** to specify whether the bit field is signed or unsigned. Table 13 shows the possible widths for bit fields, where w = width.

Storage boundaries for bit fields

A storage unit boundary for a bit field is determined by the number of bytes needed to store a value for the associated type. **char** bit fields have boundaries on even addresses, and **long** bit fields have boundaries on addresses divisible by 4. **long long** bit fields are an exception, since they also have boundaries on addresses divisible by 4.

The compiler allocates bit fields from the most significant bit to the least significant bit. Bit fields normally do not align themselves as they are placed in the struct. However, bit fields never cross a storage unit boundary. (A unit is one or more bytes depending on the data type. For example, a **char** unit is one byte, a **short** unit is two bytes.) In these cases, the bit field will be aligned according to its declared type.

Named vs. unnamed bit fields

Named bit fields impose the same alignment on the enclosing **struct** and **union** as non-bit field members. Unnamed bit fields do not affect the external alignment. However, within the structure, they behave as named bit fields.



Table 13 Bit-Field Widths

Bit-Field Type	Width (w Bits)	Range of values	
signed char	1 to 8	-2^{w-1}	to $2^{w-1}-1$
char		0	to 2^w-1
unsigned char		0	to 2^w-1
signed short	1 to 16	-2^{w-1}	to $2^{w-1}-1$
short		0	to 2^w-1
unsigned short		0	to 2^w-1
signed int	1 to 32	-2^{w-1}	to $2^{w-1}-1$
int		0	to 2^w-1
enum		0	to 2^w-1
unsigned int		0	to 2^w-1
signed long	1 to 32	-2^{w-1}	to $2^{w-1}-1$
long		0	to 2^w-1
unsigned long		0	to 2^w-1
signed long long	1 to 64	-2^{w-1}	to $2^{w-1}-1$
long long		0	to 2^w-1
unsigned long long		0	to 2^w-1

Zero-length bit fields

A bit field of zero length causes alignment to occur at the next unit boundary according to its declared type. Thus, a zero-length bit field of type **int** is aligned on the next four-byte boundary. It also imposes a four-byte alignment on an enclosing **struct**.

For example, the structure definition:

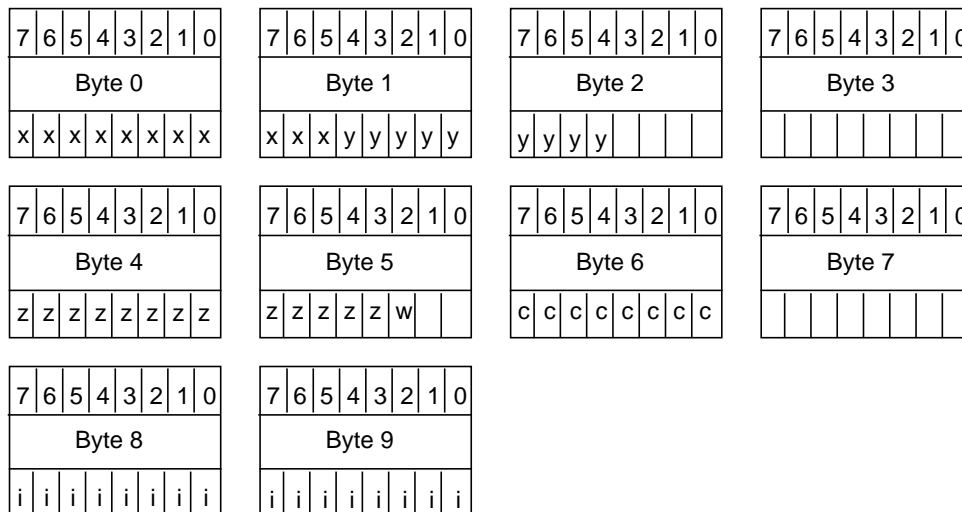
```
struct {unsigned x:11,y:9,z:13,w:1; char c; short i;}
```

is mapped to memory as shown in the following figure. The structure is aligned on address boundaries divisible by 4 because it contains **int** types.

Note that the unnamed bit field forces padding, while alignment rules sometimes pad. If *z* were changed to a **char** type, it would still be forced to begin in byte 4. If there was no unnamed bit field, *z* would begin in byte 2, 3, or 4 depending on whether it could fit in the space remaining without crossing its storage-unit boundary.



Storage mapping



Storage classes

Variables are mapped to storage as follows.

Local variables

Each uninitialized local **static** variable is placed in the BSS section. Each initialized **static** variable is placed in the DATA section.

Global variables

Each uninitialized global variable with no **extern** specifier is defined as a common block. Each initialized global variable is mapped into the DATA section and given the global attribute. Each **extern** variable is given the global and undefined attributes.

const variables

Each **const**-qualified **static** initialized variable is mapped to a read-only section (typically TEXT or LIT).

auto and register variables

Each **auto** variable is assigned either to a machine register or to storage in the routine's stack frame. See the appendix for your platform for details of run-time organization. The compiler attempts to place each **auto**-classed variable in a register provided the variable's type is appropriate and its address is not required. In a function containing calls to **setjmp()**, **auto** variables are mapped to memory unless the **register** class is explicitly specified, so that the semantics of **setjmp()** and **longjmp()** work correctly.





Chapter 9 Inline assembly code

This chapter describes the inline assembler directive provided with CenterLine-C. A function call of the form `_ASM(" string")` places the string in the corresponding place in the generated assembly file.







_ASM and direct register assignment for variables

The `_ASM` directive and direct register assignment provide a powerful facility for inline placement of code. You should use `_ASM` sparingly, however, and never to alter control flow across C constructs.

The `_ASM` directive may be used only if the command-line switch `-S` is specified; otherwise, a diagnostic error is returned. `-S` is always specified for Solaris 2.

The *string* variable must not alter control flow (that is, contain jumps). Furthermore, the only machine registers it can modify are those that can be modified by a call (that is, global registers).

NOTE When you compile with the `-Xa`, `-Xt`, or `-traditional` switches, the keyword `asm` is equivalent to `_ASM`. You cannot use `asm` as a variable when you compile with these switches, and you can never use `_ASM` as a variable.

`_ASM` is a predefined function that takes an argument expressed as a single string constant. This argument is inserted in the assembly output of the compiler at the point of the `_ASM` call. A newline character is appended. No attempt is made to check the contents of the string. This directive may appear only where a function call is permitted; that is, within a function definition.

For example, to read the processor state register:

```
_ASM(" rd %psr, 10");
```

The optimizer normally assumes nothing about registers that might be used in `_ASM`. This means that the optimizer may use registers that you intended for your own purpose in `_ASM`, defeating the `_ASM` function. Direct register assignment for variables is a method of communicating to the optimizer that you intend registers for your own use.





Inline assembly code

Restrictions on using `_ASM`

You should avoid using branch instructions within `_ASM`, and you should be aware that `_ASM` statements may be optimized away.

Avoid branch instructions

You cannot reliably use a target machine's branch instructions within `_ASM`, because you may introduce flow-graph paths of which the compiler is unaware. Such paths can cause the compiler to generate incorrect code. Branch instructions should be contained within a single section of `_ASM` code, not intermixed with C. For example:

```
/* Good example. */
/* ... Some C code here... */
/* Here we make up a pretend assembly language: */
_ASM("      rd      %psr, 10  ");
_ASM("      sethi   63, %g1  ");
_ASM("      or      %g1,768,%g1");
_ASM("      andcc   %i0,%g1,%g1");
_ASM("      be,a    lab      ");
_ASM("      or      %g1,0,%i0  ");
_ASM("lab:");
```

Using a branch and label across C code could have adverse results:

```
/* Bad example.*/
int i;
i = 1;
_ASM("      b      lab");
i = 3;
_ASM("lab:");
int j = 1;      * Compiler assumes i = 3, not 1. */
```

`_ASM` statements may be optimized away

Furthermore, based on control-flow analysis the compiler may discard `_ASM` statements that appear dead:

```
_ASM("      b      lab  ");
return;
_ASM("lab:");
_ASM("      mov    r1,r2");
```

In this example, the two `_ASM` directives after the `return` are discarded because they are dead (code following a `return`; is never executed). It is not reasonable to turn off dead-code elimination, because the compiler uses it to eliminate extra compiler-inserted code that assists in other optimizations. Nor is it reasonable to assume that each `_ASM` breaks up a basic block, because the compiler would have to further assume that each `_ASM` can be branched to from every other `_ASM`, and the connectivity in the flow graph would result in bad code for the C constructs near the `_ASM` directives.





Chapter 10 Diagnostic messages

Messages from the CenterLine-C compiler report

- *file I/O errors*
- *system errors*
- *user errors and warnings.*

This chapter describes the conditions under which the compiler is likely to produce diagnostic messages, shows what the messages look like, and presents ways to control message selection and output format. User errors and warnings that require further explanation are listed by message number along with a description.







File I/O errors

File I/O errors are fatal. They can occur when you try to open a non-existent file, or when the compiler writes an output file and not enough disk space is available.

NOTE Fatal errors may result in compiler temporary files being left on the disk. These are files with a **.tmp** suffix. They should be removed.

The I/O error messages you are most likely to encounter are:

fff: file not found.

This message is produced when a source file specified on the command line cannot be found. In addition, when running the compiler in ANSI mode, the files **clccansi.st**, **clccansi.pt**, and **clccansip.pt** must be accessible.

***Cannot write to intermediate file.

Usually caused by too little space on disk. Remove unnecessary disk files and try again.

System errors

System errors are fatal and you should rarely encounter them. The error messages take the form:

```
>>>> S Y S T E M   E R R O R   n <<<<, in Module:Function
Error message text.
```

where *n* numbers the occurrences of system errors, *Module* is the module name, and *Function* is the function name. The only system error messages with which the user should be concerned are these:

Dynamic array allocation/reallocation failed. Out of memory.

This error indicates that there was too little memory to compile the program. This usually occurs when at least one function definition is extremely large. Such functions should be broken up into smaller ones.





Diagnostic messages

Recover: Exceeded the following limit: *Limit*.

A table overflowed while a syntax error was being repaired. The table limit is fixed, so no increase in memory can improve the situation. Repair the error.

Other system error messages are associated with internal compiler errors or inconsistencies that normally should not occur. If they do occur, send a test case to CenterLine Technical Support.

User errors and warnings

User error messages are grouped into three categories:

- lexical
- syntactic
- constraint

Messages that report errors terminate compilation after the phase issuing the diagnostic, so errors that would otherwise have been detected by later phases are not reported until all earlier errors are repaired and the compiler is reinvoked.

All user diagnostics are accompanied by the file name and a line number *n* in the form

```
"file", line n: [warning] message
```

Line and column numbers show where in the source the error was detected. Messages that require further explanation are listed in "Error and warning messages" on page 82.

For lexical and syntactic errors, the erroneous line is displayed with a caret "^" beneath it at the point of error detection.

Lexical error messages

Lexical error messages are produced when an improperly formed word is detected, such as a string with a missing closing quote. Here's an example:

```
"hello.c", line 6: (lexical) String constant not terminated.
```



**Syntactic error messages**

Syntactic error messages are produced for programs that are ill formed at the phrase level, such as a missing semi-colon or inserted spurious symbol. The message is accompanied by a statement of the **REPAIR** that the compiler effected so it could keep processing input.

```
"print.c", line 13: (syntactic) unexpected symbol: '<IDENTIFIER>':f
REPAIR:    '!=' was inserted before '<IDENTIFIER>':f@"print.c",L13/C8
```

Constraint error and warning messages

Constraint error and warning messages diagnose more subtle problems, such as an undeclared identifier or type mismatch. There are more than 200 such diagnostics, each of which is meant to be self explanatory. Most of them prevent the generation of object code, but some are merely warnings intended to assist the programmer.

```
"file.c", line 3: Undeclared_identifier: This is undeclared.
1 user error    No warnings    453K of memory unused.

"file.c", line 2: i: Variable is set but is never referenced.
"file.c", line 2: warning: Unused: Variable is never used.
"file.c", line 3: warning: Division by zero.
1 user error    2 warnings    457K of memory unused.
```

Warning levels

The compiler supports five levels of warning diagnostics. Level 1 warnings are considered to be of the most interest to the programmer. Levels 2 and 3 are less so; level 4 is merely informational. Level 0 suppresses all warnings.

The warning level can be set on the command line with the **-wn** switch, or set in source with the **Warning_level** pragma. For a description of the **-wn** switch see the "Command-line switches" section on page 9.

The **Warning_level** pragma has the following form:

```
#pragma Warning_level(n)
```

where *n* is a decimal integer constant denoting the level. Setting the warning level to *n* means that warning diagnostics with severity level *n* or less are displayed. A setting of 0 suppresses all warnings. The default warning level is 3.





Diagnostic messages

Messages and classifications may change as the compiler is revised.
Warning messages are classified as follows:

Level 4 messages:

Old-style K&R C would possibly produce different results for relational.

Level 3 messages:

'=' encountered where '==' may have been intended
Function called but not defined.
Unsigned compare with zero always false/true.
Variable "name" is never referenced.

Level 2 messages:

Expression has no side-effects.
Prototype causes non-standard conversion from "type1" to "type2".
Static function is not referenced.
Toggle "name" is unrecognized.
Variable "name" is possibly referenced before set.

All other warning messages are level 1.

Error and warning messages

This section presents an alphabetical list of compiler diagnostic messages (except automatically generated lexical and syntactic messages) that may need further explanation.

'=' encountered where '==' may have been intended.

= was detected as an operator in a Boolean expression, such as **if (x = y) {...}**. Often this is a mistake, as **if (x == y) {...}** was intended.

Argument name is missing in function declaration.

For function definitions, argument names must be supplied. For example, **void f(int, float g) {...}** is illegal because the first argument lacks a name.

Argument name is not specified in function header.

For function definitions, arguments declared must be present in the function header. For example, **void f(a,b) int x;...; {...}** is illegal because **x** is not present in the function header.





Argument number *n* not named.

The *n*th argument in a named call is not specified. Here's an example:

```
int f ( int a, int, int c ) {...}
...
f(a=>1,c=>2);
```

where the second argument is not specified.

"auto" must appear within a function. "static" assumed.

Storage class **auto** cannot be given for declarations that do not appear within a function.

Bit field is not valid as an argument to "sizeof."

Since bit fields need not occupy an integral number of bytes, taking their **sizeof** is prohibited.

Call to a function returning an incomplete or zero-length type is attempted.

A function cannot return a **struct** or **union** type whose fields have not yet been specified. For example, **struct s; struct s *f() {...}** is legal since **f** returns a pointer to an incomplete **struct** type, but **struct s; struct s g() {...}** is illegal.

Calling convention not consistent with declaration at xxx.

The calling convention being used at a function definition does not correspond to the calling convention that was designated at a previous declaration of the function (at location xxx).

Cannot dereference a pointer to void.

Type **void*** was introduced as a means of defining a "generic pointer" compatible with other pointers. But there is no such thing as an object of type void. Therefore, dereferencing a pointer to void is illegal.

Cannot return from iterator for-loop body.

As an implementation restriction, one cannot return from an iterator for-loop body; one can, however, break, continue, or use a goto inside an iterated for-loop body.

enum tag is not defined.

A declaration such as **enum x**; was encountered. Tag **x** has no definition.

Expression has no side-effects.

An expression used in a statement context has no side effect; therefore the expression is useless. For example, **2+3**;





Diagnostic messages

Function called but not defined.

Any function that was called but not defined is noted as a warning. Although this practice is permissible in C and is especially useful when calling library functions, a common error is to misspell a function name. Without this warning, the error goes undetected until link-time. Furthermore, errors in parameter linkage can occur when a call is made to an undefined function. We recommend that the library **header.h** header files always be included to get parameter checking, and that function prototypes be used for external function declarations, rather than making use of the “feature” of C for calling undefined functions.

Function declaration is inconsistent with the "int"-returning function declaration imputed at *Ln/Cm*.

A function called before it is declared is assumed to be a function returning **int**, and any subsequent declaration of the function must declare it to be so. For example,

```
main () { (...) f(3);(...) } void f() {...}
```

is illegal since **f** was called before being defined and therefore assumed to return **int**.

Function definition semantics overridden by prototype at *Ln/Cm*.

A prototyped function declaration takes precedence over an ordinary function definition. Here's an example:

```
int f(int a, int b);
...          /* Prototyped functionality. */
int f (a,b)
int a,b; {...}
```

The function declaration in the function definition is overridden by the prototype declaration. This warning may be suppressed by turning off the **Prototype_override_warnings toggle**.

Function has no return statement.

A function with a non-void return type contains no return statement. This typically happens with “old” C programs that did not use void to indicate that a function returns nothing.

Function parameter names are allowed only on function definitions, not declarations.

int f(a,b,c); is a function declaration that names the parameters (**a,b,c**). This is illegal unless function prototype syntax is used, as in **int f(int a, int b, int c);**



Insufficient number of arguments to iterator.

The actual number of arguments passed to an iterator must match the formal argument list.

Insufficient number of arguments to macro.

The number of arguments to a macro must agree exactly with the number of parameters in its #define.

Iterator call required.

An iterator was called that was not defined.

Iterator functions must be of type void.

Iterators cannot yield a value through the return statement. Thus, all iterators must have return type void.

Label identifier happens to have the same name as an enum identifier. Perhaps a preceding "case" is missing.

The compiler issues this warning for label B in the following program segment:

```
enum {A, B, C} i;
...
switch(i){
    case A: B: ... /* Meant to say "case B:" */
    case C: ...
}
```

The user most likely intended the keyword case to precede the label B.

Left side of '=' is an array, which is not an lvalue.

In this context a so-called *lvalue* is required but was not found. An *lvalue* is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to &, ++, and --. An array is *not* an example of an *lvalue*.

Lower bound of range greater than upper bound.

This can happen only in CenterLine-C case statements where range expressions are allowed as labels (an extension).

More iterator arguments specified than required.

More values were provided for in the iterator-driven for-loop than were yielded.

Named bit-field of length 0 not permitted.

A declaration such as `struct { int i:0, j:2 };` was encountered. `i` must be omitted. As is, it is possible to refer to the field. Such a reference would be illegal.



Diagnostic messages

No "#pragma Data" is active.

#pragma Data; was encountered without a preceding, and matching, **#pragma Data(...);**.

Parameter not found or specified more than once.

In a function call using named parameter association, a parameter was named twice, or a non-existent parameter was referenced.

Parameter "*name*" not supplied.

Named parameter not specified. Here's an example:

```
void P ( int a, int b, float c ) {...}
...
P (a=>1, c=> 3);
```

This does not initialize the named parameter **b**.

Passing an argument of type "*type1*" where "*type2*" is expected.

An attempt was made to pass an argument of a wrong type to a function such as passing a **float** for a parameter that is a **struct**.

Pointer to an object of unknown length is being indexed.

Attempting to index a pointer to an object of unknown length. **struct x *p; *p[0]** is an example of this type of error.

"#pragma Data" active at end of module.

A **#pragma Data (...)** was given in a module or function, with no terminating **#pragma Data**. This is permitted but the programmer may have forgotten to supply the terminating pragma, thus perhaps including more data declarations in a data segment than intended.

Previous "#pragma Data" is still active.

#pragma Data (...) was given in the context of an already active **#pragma Data (...)**. Insert **#pragma Data()** preceding the offending pragma to "turn off" the active pragma.

Previous definition of macro superseded.

Redefinition of a macro is permitted. The redefined macro takes precedence over existing definition of the macro.

Prototype causes non-standard conversion from "*type1*" to "*type2*".

The prototype syntax causes the non-standard conversion from *type1* to *type2*. This is signaled as a warning to indicate to the user that such a conversion is taking place.





Error and warning messages

"register" must appear within a function. "static" assumed.

Storage class **register** cannot be given for declarations that do not appear within a function definition.

sizeof being applied to incomplete struct/union.

The sizeof a **struct** or **union** type whose fields have not yet been specified is not known. For example, **struct s; (...)** **sizeof (struct s) (...)** is illegal because the size of the structure is unknown.

Specifier "class" ignored; no variable being declared.

In a declaration such as **static struct s{ int x;}**, the storage class **static** is useless since no object was declared.

Static function is not defined nor referenced.

A function of storage class **static** is neither defined nor called anywhere in the compilation unit.

Static function is not referenced.

A function of storage class **static** is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function and it is essentially deleted.

Type "type1" is not assignment compatible with type "type2".

(a) In an assignment expression, the right operand of type *type1* may not be assigned to the left operand of type *type2*.

(b) In a function call, an argument of the type *type1* may not be passed to a function that expects a parameter of type *type2*.

Type "type1" is not compatible with the type "type2".

In a comparison, the left operand of type *type1* may not be compared with the right operand, of type *type2*.

Unsigned compare with zero always true.

An expression of type **unsigned int** is never less than zero. Thus, a comparison to check whether the unsigned **int** expression is greater than zero will always be true, and a comparison to check if it is less than zero will always be false.

Variable "name" is possibly referenced before set.

This warning is issued by the optimizer when it has found an auto- or register-class variable that is "live" at its declaration. In other words, the compiler has detected a potential path in which such a variable is referenced prior to being assigned a value.





Diagnostic messages

This condition is not necessarily an error in that the “potential” path may never occur. For example, the variable *x* in the following code fragment would be diagnosed with this warning:

```
static void foo(int i) {
    int x;
    if (i >= 0) x = 1;
    if (i < 0) x = 2;
    printf("x=%d\n",x);
}
```

Variable expected.

In this context a so-called **lvalue** is required but was not found. An **lvalue** is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to **&**, **++**, and **--**. The rules of C require the automatic conversion of some objects into non-lvalues. For example, the operand of **&** must be an **lvalue**, so **int i = &(a+b)** produces the **Variable expected**. diagnostic. A common cause of this message is the use of a construct such as:

```
int * p;
c = * ((char*)p)++;
```

which is legal on most PCC-compatible compilers, but disallowed by the ANSI Standard. Use instead:

```
int * p ;
c = *( * (char**) &p)++;
```

to circumvent the restriction. This solution works only when *p* is not a **register** variable; unfortunately we know of no solution for **register** variables.

"volatile" qualifier in cast has no effect.

The warning is issued to remind the user that simply casting a variable to a volatile type does not force the variable to be volatile. This is because the variable being cast is actually an rvalue. For example, the cast in the following assignment accomplishes nothing:

```
int a,b;
...
a = (volatile int)b;
```

The effect of forcing *b* to be volatile can be attained by the following sequence:

```
a = *(volatile int *)&b;
```





Appendix A HP run-time organization

This appendix describes the following platform-specific aspects of the CenterLine-C compiler on the HP platform:

- *Stack-frame layout*
- *Register usage*
- *Parameter passing*
- *Return values*
- *Prologue/epilogue code*





Stack-frame layout

The HP 9000 Series 700 run-time model implements a memory stack frame and a fixed number of integer and floating point registers. The memory stack frame is used for local variables that cannot be stored entirely in registers, and as temporary storage for certain registers.

Figure 1 shows the memory stack frame organization. The stack frame grows from low to high address, and is 64-byte aligned. When called, a procedure allocates space on the memory stack frame in 64-byte multiples. The stack frame size includes the following areas required by the procedure:

<i>Saved Registers Area</i>	Non-volatile registers modified by the procedure are saved here.
<i>Local Variables Area</i>	Local variables not mapped to registers are saved here.
<i>Temporaries Area</i>	Temporary variables not mapped to registers are saved here.
<i>Variable Argument Area</i>	Arguments not mapped to registers are saved here.
<i>Frame Marker and Fixed Argument Area</i>	The minimum fixed-space overhead for making a procedure call (excluding millicode call, which is not generated by the compiler and does not require a stack frame.). This includes space for four integer arguments or two double-precision floating-point register arguments, and space for return pointers and static links.

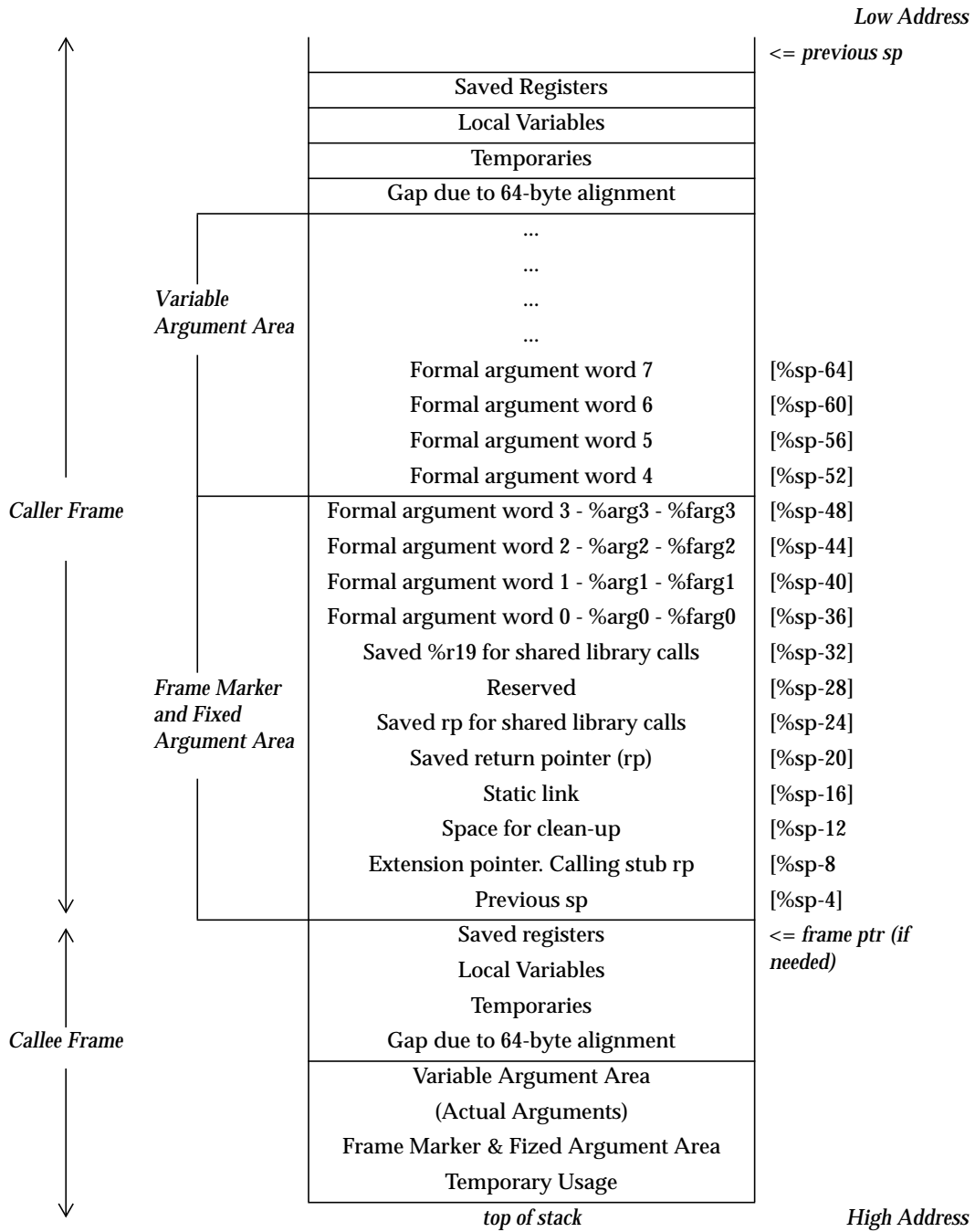


Figure 1 HP Memory Stack Frame Organization

Register usage

Table 14 illustrates the register-usage and naming conventions. Many registers have multiple usage. For example, `%r3` can be used as a frame pointer or a non-volatile general register.

A floating-point register can be accessed as a 64-bit entity, or as two 32-bit half-registers using an L (left) and R (right) suffix. The code generator attempts to map a floating-point object to a 32-bit half-register to maximize the number of floating-point registers available to the register allocator.

Calling `alloca()` will increment the stack pointer by multiples of 64 bytes.

Table 14 HP Register Usage and Naming Conventions

Usage	Register
Zero value	<code>%r0, %fr0</code>
Hard-coded temporary	<code>%r1</code>
Return pointer	<code>%rp</code>
Integer and floating-point argument registers and temporary registers	<code>%arg0-%arg3</code> (alias for <code>%r26-%r23</code>) <code>%farg0-%farg3</code> (alias for <code>%fr4-%fr7</code>)
Temporaries (volatile integer and floating-point registers)	<code>%t1-%t4, %ret0, %ret1, %mrp</code> (alias for <code>%r22-%r19, %r28, %r29, %r31</code>), <code>%fr8-%fr11, %fr22-%fr31</code>
Non-volatile integer and floating-point registers	<code>%r3-%r18, %fr12-%fr21</code>
Frame pointer ^a	<code>%r4</code>
Data pointer	<code>%dp</code> (alias for <code>%r27</code>)
Return value	<code>%ret0</code> (alias for <code>%r28</code>)
Millicode return value	<code>%ret1</code> (alias for <code>%r29</code>)
Stack pointer	<code>%sp</code> (alias for <code>%r30</code>)
Millicode return pointer	<code>%mrp</code> (alias for <code>%r31</code>)

a. Use of `%r4` as a frame pointer register is arbitrary. HP function-calling conventions do not specify a frame pointer register.

How function arguments are passed

A maximum of 16 bytes is passed in either `%arg0-%arg3` and `%farg0-%farg3` for integer or single-precision floating-point arguments respectively, or `%farg0` and `%farg3` for double-precision floating-point arguments. The calling convention allows the use of both integer and floating-point registers to pass arguments. However, you cannot use registers `%arg0-1` concurrently with `%farg0-1`, nor can you use `%arg2-3` concurrently with `%farg2-3`. Arguments that do not fit into these registers are passed in the variable argument area of the memory stack frame.

Table 15 illustrates the calling convention:

Table 15 HP Calling Convention

Function signature	Argument	Passed in...
func (<i>int, int, int, int, int</i>)	1	<code>%arg0</code>
For example: f (1, 2, 3, 4, 5)	2	<code>%arg1</code>
	3	<code>%arg2</code>
	4	<code>%arg3</code>
	5	in memory
func (<i>double, int, double</i>)	1.2	<code>%farg1</code>
For example: f (1.2, 2, 3.3)	2	<code>%arg2</code> (<code>%arg3</code> skipped)
	3.3	in memory
func (<i>int, double, double</i>)	1	<code>%arg0</code> (<code>%arg1</code> skipped)
For example: f (1, 2.2, 3.3)	2.2	<code>%farg3</code>
	3.3	in memory
func (<i>float, int, float</i>)	1.2	<code>%farg0</code>
For example: f (1.2, 2, 3.3)	2	<code>%arg1</code>
	3.3	<code>%farg2</code>
func (<i>int, float, float</i>)	1	<code>%arg0</code>
For example: f (1, 2.2, 3.3)	2.2	<code>%farg1</code>
	3.3	<code>%farg2</code>

All objects are four-byte aligned (doubles are eight-byte aligned). A single character is filled (according to its type) to four bytes, and all four bytes are passed.

There are two optional hidden parameters: a structure return pointer (passed in `%ret0` if the callee returns a structure) and a static link (passed in `%ret1`).

A structure, union, or long double passed by value is actually passed by reference to the called procedure. The called procedure makes a copy of the fields it modifies to preserve the passing-by-value semantic. For more information, refer to the *PA_RISC Procedure Calling Convention Manual* from HP, order number 09740-90015-E0191.

How function values are returned

Table 16 shows the registers used to return values of types **double**, **float**, **int**, and **struct**.

Table 16 HP Registers Used for Return Types

Return Type	Register
double	<code>%fr4</code>
float	<code>%fr4L</code>
int	<code>%ret0</code>
struct	<code>%ret0</code> ^a

a. A **struct** type is returned as a structure return pointer



Prologue/epilogue code

The prologue/epilogue code establishes the environment needed by the body of a function. The code creates stack frames, saves important information passed in temporary registers in less temporary places, and returns to the caller.

The considerations in creating such code are

- The size of the stack frame (three variants: 0, < 8192 bytes, >= 8192 bytes).
- Whether this function is a leaf.
- Whether this function uses an **alloca**-like function.

Because most (but not all) are independent of the others, there are many variations. All variations can be derived from the worst case (see the third example) as simplifications. The simplest case is also shown.

A simple non-leaf function

First, a “simple” non-leaf function. This is expected to be the normal case.

```
foo    .PROC
       .CALLINFO
       .ENTRY
       stw    %rp, -20(%sr0, %sp)
       ldo    64(%sp), %sp
       ...
       ldw    -84(%sr0, %sp), %rp
       bv     %r0(%rp)
       .EXIT
       ldo    -64(%sp), %sp
       .PROCEND
```

A simple leaf function

Now a “simple” leaf function with a parameter and a single return value:

```
foo    .PROC
       .CALLINFO
       .ENTRY
       ...
       bv     %r0(%rp)
       .EXIT
       ldo    1(%r0), %ret0      ; Return 1
       .PROCEND
```



A more complicated non-leaf function

And last, a maximally complicated function call and return:

```

foo      .PROC
        .CALLINFO
        .ENTRY
        stw      %rp,-20(%sr0,%sp)
        stwm     %r4,8(%sr0,%sp)
        ldo      -8(%sp),%r4
        stwm     %r18,4(%sr0,%sp)
        stwm     %r17,4(%sr0,%sp)
        stwm     %r16,4(%sr0,%sp)
        stwm     %r15,44(%sr0,%sp)
        ...

;449 | char *p = alloca(len);
      ldo      63(%r17),%arg0
      ldo      -64(%r0),%arg1
      and      %arg0,%arg1,%arg0
      or       %sp,%r0,%r18;
      add      %arg0,%sp,%sp ; For non-leaf nodes
                                ; the value will be
                                ; adjusted appropriately
                                ; Call to alloca will
                                ; further adjust the
                                ; stack pointer.
      ...
      ldo      64(%r4),%sp      ;Restore stack
      ldwm     -44(%sro,%sp),%r15 ;pointer
      ldwm     -4(%sr0,%sp),%r16
      ldwm     -4(%sr0,%sp),%r17
      ldwm     -4(%sr0,%sp),%r18
      ldw      -28(%sr0,%sp),%rp
      bv       %r0(%rp)
      .EXIT
      ldwm     -8(%sr0,%sp),%r4
      .PROCEND

```





Appendix B Sun run-time organization

This appendix describes the following platform-specific aspects of the CenterLine-C compiler on the Sun platform:

- *Stack-frame layout*
- *Register usage*
- *Parameter passing*
- *Return values*
- *Prologue/epilogue code*
- *Assembly-language communication*





Stack-frame layout

The SPARC run-time model implements a memory stack frame and a fixed number of register windows. The memory stack frame is used for local variables that cannot live entirely in registers, and as a place to save the register windows. Figure 2 illustrates the memory stack frame.

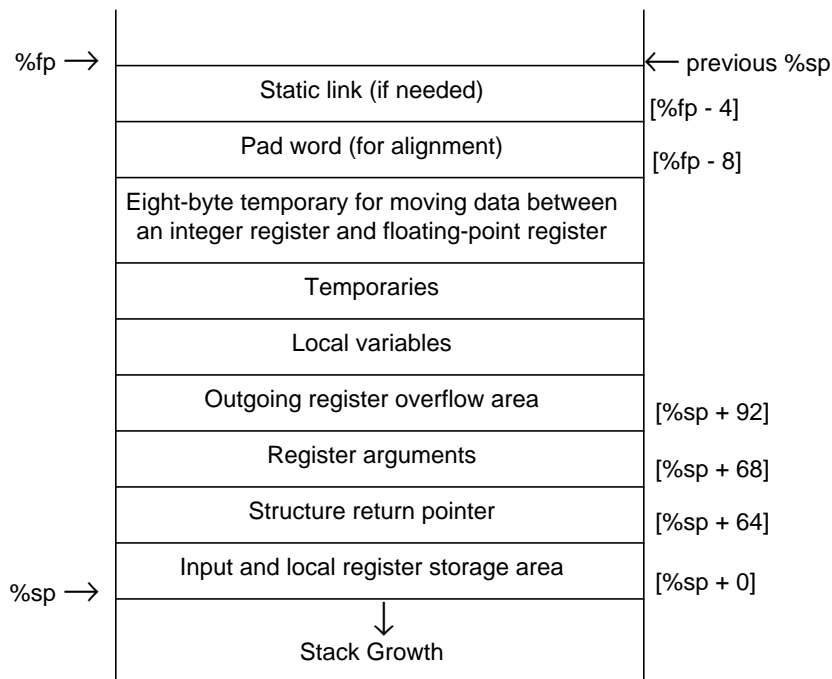


Figure 2 Sun Memory Stack Frame Organization

The memory stack frame grows downward in memory (from high to low address) and is always eight-byte aligned. At entry time, procedures must allocate space on the memory stack frame in multiples of eight bytes. The size of the memory stack frame is unlimited.



Register usage

Figure 3 illustrates the register-usage and naming conventions.

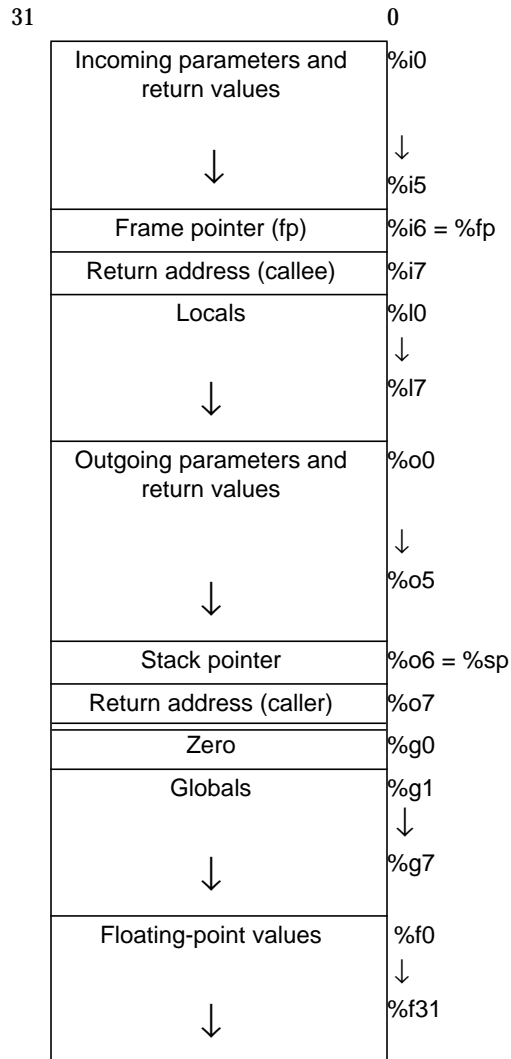


Figure 3 Sun Integer and Floating-Point Registers

Registers **%g1** through **%g7**, **%o0** through **%o5**, and **%f0** through **%f31** are not preserved across calls.





Circular-model registers

The integer registers are divided into register windows and a set of global registers. Each register window is 24 registers. Register windows for procedures overlap. The overlapping register windows are arranged in a circular stack called the *circular* model. Figure 4 illustrates overlapping windows.

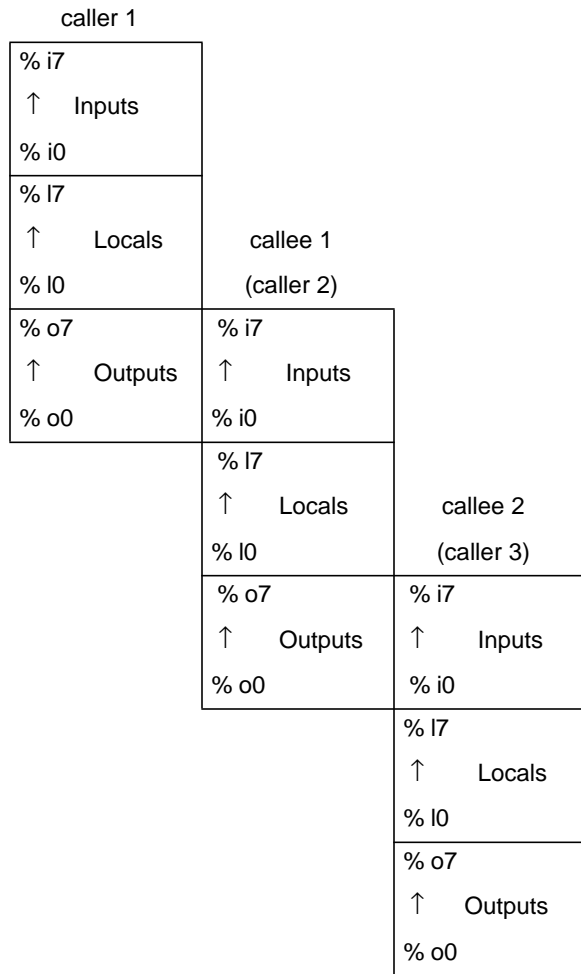


Figure 4 Sun Circular-Model Register Windows

The circular model eliminates the need to save and restore registers across function calls. When a process is running, the circular model saves time and function call overhead. But, when a process is interrupted, multiple register windows must be saved to memory.





How function arguments are passed

The first six words of function arguments are passed in **%o0** through **%o5**. On the receiving side, the registers containing these values are renamed **%i0** through **%i5** in the new register window (see Figure 4). **%i6** contains the stack pointer and **%i7** contains a temporary value. Additional parameters are passed in memory starting at the address referenced by **[%sp+92]**.

All objects take at least one word. Therefore a single character is filled (according to its type) out to four bytes, and all four bytes are passed.

There are two optional hidden parameters: a structure return pointer and a static link. The first is passed in memory at **[%sp+64]**; the second is passed in global register **%g1**. **%g1** may be used as a temporary by any function. It is up to the callee to preserve the static link if necessary.

Passing structs

A **struct** or **union** that is to be passed as a parameter is first copied to a temporary location; the address of the temporary is then passed in the same manner that a pointer is passed.

Structure return pointer

The structure return pointer is passed from caller to callee when the callee returns a structure.

floats and doubles

floats and **doubles** are passed in a sequence of one or two output registers (**%o0** through **%o5**). No attempt is made to align **doubles** on an eight-byte boundary. **long doubles** are treated like **structs** in regard to parameter passing.

How function values are returned

Return values are dealt with in a manner similar to arguments. A **float** is returned in **%f0**; a **double** is returned in the **%f0/%f1** pair. Integers and pointers are returned in **%o0** (from the caller's point of view; **%i0** from the callee's point of view). Structure values and **long doubles** are returned in the memory location passed in the structure return pointer. (See "How function arguments are passed".)





Prologue/epilogue code

The prologue/epilogue code establishes the environment needed by the body of a function. The code creates stack frames, saves important information passed in temporary registers in less temporary places, and returns to the caller.

The considerations in creating such code are

- The size of the stack frame (0 bytes, < 4096 bytes, or >= 4096 bytes).
- Whether this function calls another (two variants: leaf(no), non-leaf (yes)).
- Whether this function is passed a static link.
- Whether this function returns a structure.
- Whether this function uses an **alloca**-like function.

All variations can be derived from the worst case (see the last example below) as simplifications. The simplest case is also shown. These instruction sequences need not be duplicated, only their effect.

Here we provide a few examples of function prologue and epilogue code. In each example, **frame_size** is the size of the stack frame needed by the called function and is double-word aligned.

A simple non-leaf function

First, a “simple” non-leaf function. This is expected to be the normal case.

```

caller:
    ...
    call    callee
    nop
    ...
callee:
    save    %sp,-frame_size,%sp
           ! Create stack frame and
           ! register window.
    ...
    ret
    restore ! Remove frame and window.

```



**A simple leaf function**

Now a “simple” leaf function with a parameter and a single return value:

```

caller:
    ...
    call    callee
    mov     1,%o0      ! Pass a '1'.
    ...

callee:
    ...
    retl
    mov     12,%o0     ! Return a '12'.

```

A more complicated non-leaf function

And last, a maximally complicated function call and return:

```

caller:
    ...
    add     %fp,-40,%o0 ! Structure return
                                ! pointer.
    mov     1,%o1      ! Pass '1'.
    call    hard
    mov     %fp,%g1    ! Pass static link.
    ...

callee:
    sethi   %hi(-frame_size),%g7
    or      %g7,%lo(-frame_size),%g7 ! Big frame
                                ! size.
    save    %sp,%g7,%sp          ! Create it.
    mov     %g1,%i3             ! Save static link.
    ...                          ! Return value
    ...                          ! stored in memory
    ...                          ! pointed to by %i0.
    ret
    restore

```





Assembly-language communication

This section explains how to write assembly-language routines that are accessible to CenterLine-C.

Writing assembly routines

The “Prologue/epilogue code” section on page 105 describes the code an assembly routine must execute in order to be callable from C. Write assembly routines according to the following guidelines, where *name* is the function's name as referenced from C, and *frame_size* is the size of the local stack frame.

```
.seg    "text"
.global name
name:   save %sp, -frame_size,%sp
        ! The body of the routine goes here.
        ret
        restore
```

For a description of how arguments are passed and how function results are returned, see the “How function arguments are passed” section on page 104.

Function naming conventions

A function that is global, that is, callable across module boundaries, must have information provided to the linker that associates its name with its address. This is done by placing a corresponding name in the object module's external symbol table. This name need not be the same as the actual name of the function; it need only be derivable from it.

In assembly language, the name that is placed in the symbol table is lexically identical to the name of the corresponding symbol. In C under SunOS 4.x, the entry in the symbol table is derived from the corresponding symbol by prefixing an underscore “_”. An external routine **foo** has a symbol table entry of **_foo**. For Solaris 2.x, the underscore prefix is not used.

Examples

This section shows examples of calling assembly routines from C and vice versa. For SunOS, function labels must begin with an underscore.

Calling assembly routines from C

```
.seg    "text"
.global peek
peek:   retl                                ! "leaf" return.
        ld      [%00],%00                  ! Delayed.
```





Calling C from
assembly

```
.seg    "text"
...
save   %sp, -frame_size, %sp! Create frame.
...
sethi  %hi(msg), %o0      ! Load address
or     %o0, %lo(msg), %o0 ! of msg.
call   write_string
nop
...
ret
restore                                ! Remove frame.
```

Data communication

Global variables in CenterLine-C programs are aliased in the same manner as public functions. A global variable **x** appears in the symbol table as **x**.

"Storage mapping" on page 67 explains how the various C data types are mapped into storage. Note that uninitialized global variables without the **extern** qualifier are actually defined as individual common blocks.

The following examples illustrate the sharing of variables across C and assembly modules.

CenterLine-C

```
int  alpha,beta;
char hextable[] = "0123456789ABCDEF";
extern char *names[]; /*Read-only table of names.*/
extern short status;
```

Assembly

```
.common alpha,4
.common beta,4
.seg    "text"
.align 4
.global names
names:  .word  L01
        .word  L02
        .word  L03
        .word  0
        .align 4
L01:    .ascii "alfred\0"
        .align 4
L02:    .ascii "bonny\0"
        .align 4
L03:    .ascii "charlie\0"
        .seg    "data"
        .global status
status: .word  0
```



Index

A

- anno** (C command-line switch) 9
- Annotate_includes** pragma 56
- annotated source, cross-reference format 60
- Annotated_listing** pragma 56
- ANSI
 - characters in identifiers 63
 - definition 4
 - floating point 64
 - library 6
- ansi** (C command-line switch) 9
- _ASM** inline assembler directive 73
- asm** keyword 75
- assembly language
 - calling C 108
 - calling from C 108
 - communication 107
 - function naming conventions 107
 - inline directive 73
 - routines 107
 - sharing variables with C 108

B

- B** (C command-line switch) 9
- Back_substitute_epilog** toggle 20
- Behaved** toggle 20
- bit fields, ANSI requirements 65

C

- c** (C command-line switch) 9, 10
- characters, ANSI requirements 63
- __CLCC__** variable 5
- clcc** command 3
- CLCC_CORE** environment variable 4

- clccxref** command 54
 - options 54
 - switches 55
- Cleanup_spills** toggle 20
- code, listing 73
- columns in code listing 57
- cpp** (C command-line switch) 9
- Cross_jump** toggle 21
- cross-reference
 - distinction of file names 60
 - features 53
 - format 58
 - listing 54
 - pragmas 56

D

- diagnostics
 - messages 77
 - pragma for 18
- Dname** (C command-line switch) 9
- dryrun**(C command-line switch) 9

E

- E** (C command-line switch) 10
- environment variable
 - CLCC_CORE** 4
- errors 77
 - file errors 79
 - messages 82
 - user errors and warnings 80
 - warnings 81

F

- file name
 - ending with **.c** 3
 - ending with **.o** 3
 - ending with **.s** 3

Index

in cross-reference 60
 file table cross-reference format 58, 59
 floating point representation 64
-fsingle (C command-line switch) 10
 functions, referencing across libraries 6

G

-g (C command-line switch) 10
Global_CSE toggle 21
Globals_volatile toggle 21

H

-Hi switch 45
-Hia switch 45, 34
-Hib=*n* switch 45, 34
-Hic=*n* switch 45, 34
 __HIGHC__ variable 5
-Hih switch 45, 34
-Hin switch 46, 34
-Hir switch 46, 34
-His=*n* switch 46, 34
-Hit=*n* switch 46, 34
-Hiu switch 46, 34
-Hiw switch 47, 34
-Hix switch 47, 34
-Hldopt switch 10, 11
-Hlines (C command-line switch) 11
-Hlist (C command-line switch) 51, 10, 11
-Hoff switch 19
-Hon switch 19
 activating optimizations 27
 __hp9000s700 variable 5
 __hppa variable 5
-Hunroll (C command-line switch) 11

I

-I (C command-line switch) 11

identifiers, ANSI requirements 63
Induction_analysis toggle 22
 inlining
 debugging inlined routines 48
 selecting routines 48
 switches 45
 integers, ANSI requirements 63
 invoking the compiler 3

L

-L (C command-line switch) 12
-l (C command-line switch) 11
 library, using ANSI 6
List toggle 22
List_module_usage pragma 56
List_unused_includes pragma 56
 listings 49
 cross-reference 54
 pragmas used for 18, 51
 setting column width 57
Live_dead_iterate toggle 22
Local_CSE_iterate toggle 22
 loop unrolling 38
Loop_factor pragma 18
 setting 38
Loop_reg_rename toggle 22

M

-M(C command-line switch) 12
 macro preprocessor 4
Mem_refs_from_loop toggle 22
-MM(C command-line switch) 12
 module name, cross-reference format 59

O

-O (C command-line switch) 12
-o (C command-line switch) 12

object name cross-reference format 58
Off pragma 18, 19, 56
On pragma 18, 19, 56
optimization
 back substitution of epilogue code 33
 benefits and disadvantages 29
 and code size 28
 constants in code 37
 cross jumping 32
 and debugging 28
 enhanced register allocation 33
 inlining switches 33
 level 0 31
 level 1 32
 level 2 32
 level 3 33
 level 4 33
 level 5 35
 level 6 36
 level 7 36
 level s 36
 live/dead analysis, iterative 32
 loop memory reference elimination 32
 loop strength reduction 35
 loop unrolling 36, 38
 how loops are unrolled 39
 multi-module inlining 37
 pointer-based variables 35
 pragmas used for 18
 register allocation, enhanced 33
 space 36
 specifying level 27
 spill analysis, improved 33
 subexpression elimination
 global common 33
 iterative local common 32
 tail merging 32
 tail recursion 36
 unrolling, selecting routines 38
Optimize_for_space toggle 22
-Os (C command-line switch) 12

P

-P (C command-line switch) 12
-p (C command-line switch) 12
Page pragma 18
parameter passing 94, 104
-pg (C command-line switch) 12
-PIC (C command-line switch) 12
-pic (C command-line switch) 12
pointers, type resulting from subtracting two
 pointers 65
Pop pragma 18, 19, 56
POSIX macros, not defined 6
pragmas 17
 used with **-Hlist** 51
preprocessing directives, ANSI requirements 65
preprocessor 4
Print_var_info toggle 22
printing
 format 52
 source code, pragmas 51
-proto (C command-line switch) 12

R

-R (C command-line switch) 13
Read_only_strings toggle 23
Recognize_library toggle 24
Reduce_reg_contention toggle 24
Reg_alloc_enhance toggle 24
registers
 circular-model registers 103
 usage 93, 102

S

-S (C command-line switch) 13
Skip pragma 18
__sol__ variable 5
_SOL variable 5



Index

source code, listing 73
Source_code_order toggle 24
 __sparc__ variable 5
 _SPARC variable 5
 sparc variable 5
 stack-frame layout 91, 101
Statistics pragma 56
 __STDC__ variable 5
 storage mapping 67
 data types 67
 prologue/epilog code 96, 105
 storage classes 72
Strength_reduce toggle 24
 structures, ANSI requirements 65
 __sun__ variable 5
 _SUN variable 5
 sun variable 5
 switches
 command-line switches 9
 for inlining 33

T

Tail_recursion toggle 24
Title pragma 18
 toggles 18, 19
 pragmas used for 18
-traditional (C command-line switch) 13
Trigraphs toggle 24
 types
 character 63
 floating-point arithmetic 64
 integer 63
 pointer 65
 structures and unions 65

U

-Uname (C command-line switch) 13
 unions, ANSI requirements 65
 __unix__ variable 5
 unix variable 5

V

-v (C command-line switch) 13

W

-w (C command-line switch) 13
-w[n] (C command-line switch) 13
Warning_level pragma 18
 warnings 77, 81
 messages 82

X

-Xa (C command-line switch) 13
-Xt (C command-line switch) 14

