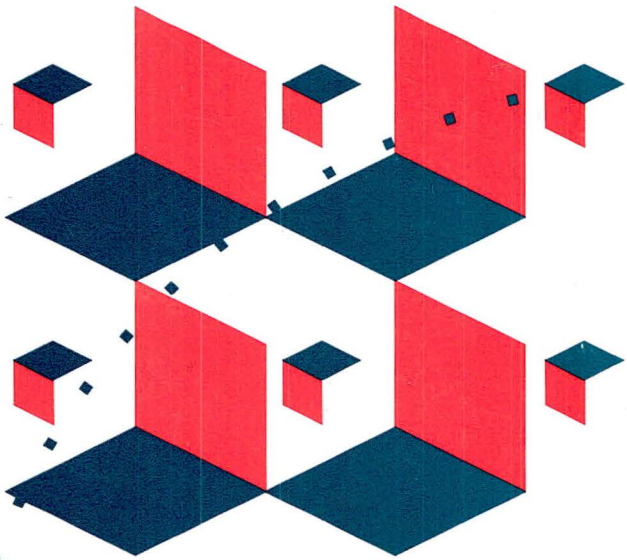


CONVEX

C User's Guide

Third Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214) 497-4000

C User's Guide

Order No. DSW-086

Third Edition
October 1994

CONVEX Press
Richardson, Texas
United States of America

C User's Guide

Order No. DSW-086

This book is recyclable including cover and binding.

Printed in the United States of America

Revision Information for

C User's Guide

Edition	Document No.	Description
Third	720-000630-210	Released with CONVEX C software V6.1, October 1994.
Second	720-000630-206	Released with CONVEX C software V5.0, January 1993.
First	720-000630-205	Released with CONVEX C software V4.1, April 1991.

Contents

Figures

Tables

About this guide

This document describes how to use the CONVEX C compiler, which is compatible with ANSI C and the POSIX operating system standard. This compiler is also backward-compatible with previous versions of CONVEX C.

Organization

The first part of this guide is organized as follows:

- Chapter 1, "Introduction," provides an introduction to basic features of the CONVEX C compiler.
- Chapter 2, "Compiler fundamentals," describes options that are available on the compiler command line.
- Chapter 3, "Compatibility modes," describes the four compatibility modes, how they are specified on the command line, how to convert an application to a particular mode, and differences among modes.
- Chapter 4, "Development tools," describes tools that assist in program development, debugging, and profiling.
- Chapter 5, "CONVEX C intrinsics,'" explains what intrinsic functions are, problems encountered when using them, and how to correct those problems.
- Chapter 6, "Input and output," discusses some of the functions that provide input and output for a program.
- Chapter 7, "Runtime library," provides a brief summary of the functions declared in the ANSI C header files.
- Chapter 8, "Preprocessor directives," describes the C preprocessor directives.
- Chapter 9, "The `asm` statement," describes the syntax of the `asm` statement, a CONVEX extension that inserts assembly-language statements into a C program.

This guide contains the following appendices:

- Appendix A, "Data types and representations," lists CONVEX data types and their representations.
- Appendix B, "Pragmas and directives," describes pragmas and directives that the compiler recognizes.
- Appendix C, "Cray intrinsics," describes Cray-compatible intrinsic functions that manipulate bits.
- Appendix D, "Implementation-defined features," lists the features of CONVEX C that may inhibit porting CONVEX C code to other ANSI C compilers.
- Appendix E, "Error messages," describes the syntax and options of the `-d` compiler option and lists `cc` and `lint` error messages. Each error message explanation includes a short example and a corrective action.

Notational conventions

The following conventions are used in this guide:

- Brackets (`[ ]`) designate optional entries.
- Horizontal ellipses (`. . .`) in command syntax examples indicate repetition of the preceding item(s). In code examples, horizontal ellipses indicate that items or statements are omitted.
- Vertical ellipses show continuation of a sequence in which not all of the statements in an example are shown.
- References to the online man pages appear in the form cc(1c), where the name of the man page is followed by its section number enclosed in parentheses.
- *Italics* within text denote user-supplied variable information.
- `Monospaced text` denotes screen output, code examples, nonvariable text in command forms, file names, utility names, and in general, text that appears exactly as shown on output or must be entered exactly as shown.
- Within command sequences set apart from text, *italics* indicate user-supplied variable information. Substitute actual information for the italicized words. For example, the command sequence

  `ld` [*options*] [*object files*] [*libraries*]

  instructs you to type the command `ld`, followed by your choice of options, object files, or libraries.

Associated documents

This section lists additional information resources recommended to both C Series and SPP Series C programmers.

C Series and SPP Series publications

The publications that follow address both C Series and SPP Series C programmers.

- *CONVEX C Quick Reference* (DSW-087) provides quick access to function prototypes, compiler directives, compiler options, and language features.
- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251) (optional product) describes how to use the interactive profiler.
- *CONVEX CXdb Concepts* (DSW-471), *CONVEX CXdb User's Guide* (DSW-473) and the *CONVEX CXdb Reference* (DSW-472) describe all aspects of the optional CXdb debugger. In this manual, see Chapter 4, "Development tools," for more information about CXdb.

C Series publications

C Series only

The following publications are recommended to CONVEX C Series programmers. The information in these references is specific to CONVEX C Series computers.

- *C Optimization Guide* (DSW-089) presents a step-by-step method for program optimization. Background information is included and forms a foundation for concepts presented throughout the document.
- *CONVEX Interlanguage Programming Guide* (DSW-043) outlines techniques for calling CONVEX C routines from programs written in other languages, and for calling routines written in other languages from CONVEX C.
- *ConvexOS Primer* (DSW-133) provides an introduction for users who have not previously used the ConvexOS operating system.
- *CONVEX adb Debugger User's Guide* (DSW-009), a tutorial and reference manual, describes the functions and operations of the CONVEX adb debugger.
- *CONVEX Consultant User's Guide* (DSW-025) (optional product) describes the functions and operations of the CONVEX csd debugger, postmortem dump (pmd) utility, and the gprof profiler.

- *CONVEX Compiler Utilities User's Guide* (DSW-096) describes the CONVEX loader and the CONVEX assembler.
- *CONVEX Application Compiler User's Guide* (DSW-401) (optional product) describes how to use the CONVEX Application Compiler to optimize programs.
- *CXmetrics User's Guide* (DSW-475) (optional product) describes software metrics data and explains how to use CONVEX CXmetrics to report on this data.
- *ConvexOS Man Pages for Programmers* (DSW-332) is the standard reference for the ConvexOS function calls. This document contains an appendix that describes how to use the `contact` utility to report problems.
- *ConvexOS System Management Documentation Kit* (DSW-015) contains information needed to manage and administer a CONVEX supercomputer system.
- *ConvexOS Tutorial Papers* (DSW-170) is a collection of previously published papers that provides instruction in document preparation, programming, text editing, supporting tools and languages, system maintenance, and system implementation.
- *ConvexOS Compiler Utilities User's Guides Documentation Kit* (DSW-005) provides detailed information on the CONVEX Assembler, Linker, text editors, and `adb` debugger.
- *CONVEX CXpa User's Guide* (DSW-251) provides information required to use the CONVEX Performance Analyzer tool, an optional product.

SPP Series publications

SPP Series only

Publications listed in this section are recommended to CONVEX SPP Series programmers. This information is specific to CONVEX SPP Series computers.

- *Exemplar Programming Guide* (DSW-067) describes efficient methods for programming in C and Fortran on Exemplar (also known as SPP Series) computers.
- *Exemplar Architecture* (DHW-014) describes the architecture of Exemplar (SPP Series) computers.
- *SPP-UX System Administration Guide* (DSW-853) provides information on administering SPP-UX.
- *HP-UX System Administration Guide* (Hewlett-Packard order number B2355-90004) discusses additional material pertinent to SPP-UX.

Other documents

For more information on the C language, refer to the following books:

- ISO/IEC 9899:1990(E) document *International Organization for Standardization and the International Electrotechnical Commission —Programming Languages C*
- *C: A Reference Manual*, by Samuel P. Harbison and Guy L. Steele, Jr., Prentice-Hall, Inc., 1987.
- *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall Inc., 1988.
- *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Std 1003.1-1988.

Online man pages

In addition to the references listed, the online man pages provide information useful to C programmers. The `man` utility formats and displays the information contained in the man pages.

Sections 1 and 7 primarily contain operating system information for users. Sections 2 through 5 contain information for programmers. The following list summarizes the topics addressed in each section of the man pages:

- **Section 1**—Commands of general utility and commands for communicating with other systems.
- **Section 2**—System calls and error numbers.
- **Section 3**—Various library functions.
- **Section 4**—Special files, related driver functions, and networked support.
- **Section 5**—Various file formats.
- **Section 7**—Miscellaneous commands, primarily related to text processing and terminal environments.

You can access the online man pages by entering:

```
% man entry-name
```

where *entry-name* is the name of the man page to be displayed. For more detailed information, refer to the man(1) man page.

Ordering documentation

To order this document or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, Texas 75083-3851 U.S.A

Order documents by title, requesting the most recent edition. In some situations, you may not want the current edition. To receive a specific edition of a manual, contact the local CONVEX office or call the CONVEX Technical Assistance Center (TAC).

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC):

- Within the continental U.S., use 1 (800) 952-0379.
- From locations in Canada, use 1 (800) 345-2384.
- From all other locations, contact your local CONVEX office.

Introduction

1

This chapter provides a brief introduction to compilation. It includes several small examples that show how to compile a program. It also defines some basic terms that are used frequently in this book.

If you already understand how compilers, linkers, and libraries interact, skip this chapter. In this chapter, you must use a text editor to create or modify the source code for your programs; if you do not know how to edit files, refer to the *CONVEX Text Editors User's Guide* (DSW-010) for information about text editors.

The CONVEX C compiler automatically generates code that takes full advantage of the architecture of the CONVEX family of supercomputers. In addition, using optimization options causes the compiler to optimize, vectorize (available only on CONVEX C Series machines), and parallelize source code to maximize execution efficiency.

On CONVEX Exemplar machines, floating-point numbers comply with the IEEE standard (IEEE/ANSI standard 754-1985). For C Series machines equipped with optional IEEE support hardware, the CONVEX C compiler provides an option for requesting IEEE-standard representation of floating-point values.

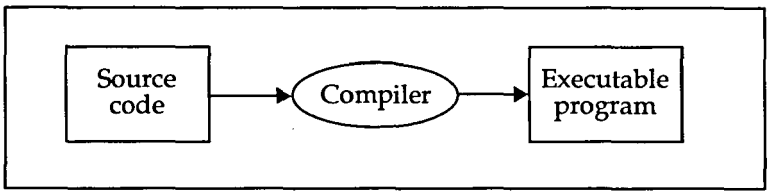
To assist in program checkout, the compiler interfaces with several debuggers and utility routines, including a postmortem dump utility, a source-level debugger, and an assembly-level debugger.

CONVEX C programs can also be compiled under COVUEshell on C Series machines. COVUEshell is a CONVEX product for C Series machines that provides a VMS-like interface and supports much of the Digital Command Language (DCL). For further information, refer to the *CONVEX COVUEshell Reference Manual*.

What is a compiler?

Compilers are programs that translate instructions, usually written in a high-level language, into machine or assembly language. The input to a compiler is source code for a program; the output is an executable program. Figure 1 illustrates the compiler's role.

Figure 1 Role of the compiler



Creating a source file

To create a C program to display the line `A simple program`, use a text editor to create the following code.

```
#include <stdio.h>
main()
{
    (void) puts("A simple program");
}
```

Name the file `myprog.c`, save it, and exit from your editor.

Functions, also called subprograms or subroutines in other programming languages, are a basic building block in C. A function performs a small task in a program. The source file that you have just created contains one function definition, `main`. A C program contains only one `main` function. The `main` function in this source file calls the function `puts`, which displays the words contained in the double quotes.

Compiling one source file

The name of the source file that you created has a `.c` extension. All files that contain source code for a C program must have this extension. (The COVUEshell environment requires a `.C` extension.) The compiler cannot compile a C language source file that does not have the `.c` extension.

To invoke the compiler, enter:

```
cc myprog.c
```

cc is the command name of the compiler, and myprog.c is the name of the file to compile.

The compiler stores the executable program in a file called a.out. To run the resulting program, enter:

```
a.out
```

The following text is output:

```
A simple program
```

If a.out had already existed, its contents would have been overwritten.

The compiler has many optional features that you select by including options on the command line of the compiler. The format of an option consists of a hyphen followed immediately by the option name. Some options require arguments. The syntax for an option varies; some require a space to delimit the argument, while others do not. Refer to Chapter 2, "Compiler fundamentals," for more information on command-line compiler options.

The -o option controls the name of the executable file. The format of this option is

-o *filename*

where

filename is the name of the executable file.

For example, the following command-line creates an executable file named *myprog*:

```
cc myprog.c -o myprog
```

Compiling more than one source file

Sometimes it is necessary to divide a program into multiple source files. For example, if the program is large, recompiling a piece of it takes less time than recompiling the entire program. Also, the functions of the program can be divided into source files containing related functions. For example, all input functions can be grouped in one file.

To understand how to create an executable program from several source files, create the two sample programs illustrated in Figure 2 and Figure 3. Name these files prog2.c and file2.c, respectively.

Figure 2 Sample program, `prog2.c`

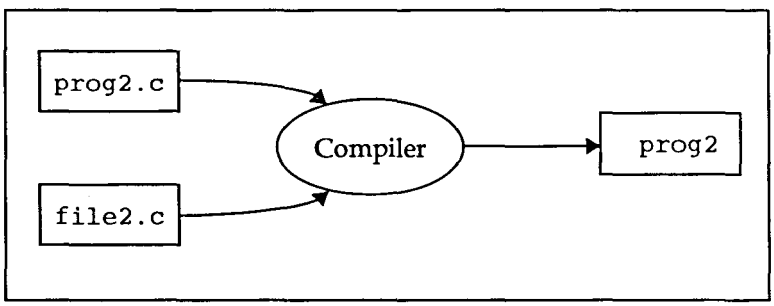
```
#include <stdio.h>

extern void second_line( void );

main()
{
   (void) puts("The first line of my second program.");
   second_line();
}
```

Figure 3 Sample program, `file2.c`

```
#include <stdio.h>
void second_line( void );
void second_line()
{
   (void) puts("The second line of my program.");
}
```

These two files comprise one program in which two functions are defined: `main` is defined in `prog2.c` and `second_line` is defined in `file2.c`. The actions of the compiler are illustrated in Figure 4.

Figure 4 Compiling multiple source files



The following command creates the executable program `prog2`:

```
cc prog2.c file2.c -o prog2
```

The compiler compiles the source files specified on the command line. After the executable program is created, run it by entering **`prog2`**.

Compiler and linker

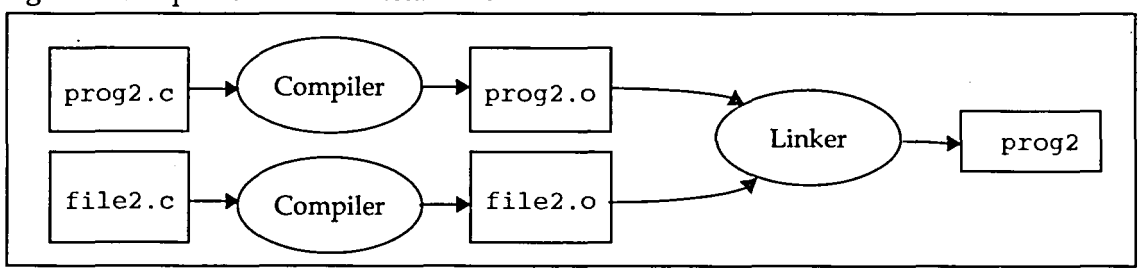
The compiler does not create an executable program in one step. First, it creates or causes the assembler to create an intermediate file that contains machine code; this intermediate file is called an object file. The object file has a `.o` file extension. For example, `prog2.o` is the object file of `prog2.c`, and `file2.o` is the object file of `file2.c`.

The C Series compiler creates object files directly, by default. However, the Exemplar compiler uses the assembler (`as`) to generate the object files; by default, `cc` automatically invokes the assembler.

Most of the machine code is a translation of the source file, but some of it refers to source code in other files. The intermediate file obtained from the `prog2.c` source file created previously contains references to the source code in `file2.c`. Such references are called external.

After the compiler has compiled all C source files listed on the command line into object files, it automatically invokes another program called a linker. The linker resolves the external references contained in the separate object files by combining the object files. When the object files are combined, there are no external references. The input to the linker is the object files generated by the compiler, and the output from the linker is the executable program. The process used to create the sample program `prog2` is shown in Figure 5.

Figure 5 Compiler and linker interactions



When the `-c` option is used, the source file is compiled but not linked. For example, the command line

```
cc -c file2.c
```

creates the object file `file2.o`, but not an executable file.

Similarly, it is possible to skip the compilation process and proceed directly to the linking process.

For instance, if the object files `prog2.o` and `file2.o` already exist, the command line

```
cc prog2.o file2.o -o prog2
```

invokes the linker and creates the executable program `prog2` because there are no source files to compile.

If `prog2.c` is modified but `file2.c` is not, the following command line suffices:

```
cc prog2.c file2.o -o prog2
```

As another example, the command line

```
cc prog3.c file3.o file4.o -o prog3
```

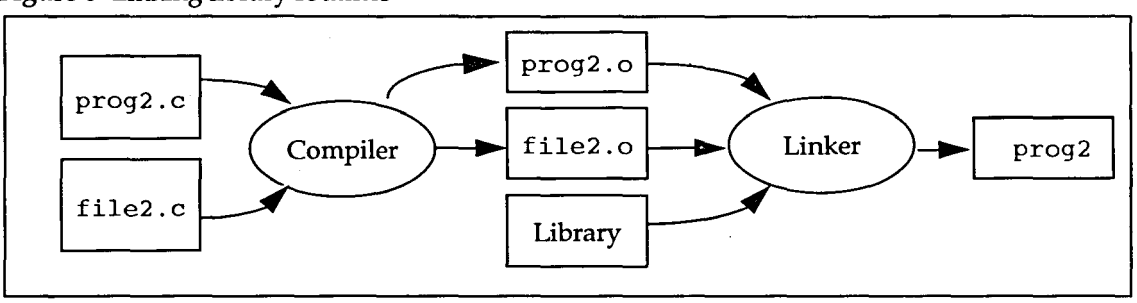
compiles `prog3.c` and then links `prog3.o`, `file3.o`, and `file4.o`.

Libraries

Libraries are composed of one or more object files. Some libraries contain functions that are defined to be part of the C language. Others might be collections of functions that you create for your programs. Each of the functions in the libraries can have external references. Like the external references in object files, the external references in libraries are not resolved until the linker is invoked.

For example, `prog2.c` calls a function called `puts`, which resides in a C library. After the source file is compiled, the object file contains a reference to the `puts` function. This reference is resolved by the linker. Figure 6 outlines when library routines are linked.

Figure 6 Linking library routines



The linker resolves external references in libraries and object files. When you invoke the `cc` command, the compiler automatically searches default libraries. Some command line

options cause the compiler to search specific libraries. If you have developed your own libraries to link into an application, include them on the command line as you would an object file. For example, if `mylib.a` is the name of a library that contains functions called by a function in `myprog.c`, the following command line creates the executable program myprog:

```
cc myprog.c mylib.a -o myprog
```

The library `mylib.a` contains object files that the compiler uses to resolve function calls in `myprog.c`. Consequently, `mylib.a` appears after all other source and object files. The order in which you specify libraries on the command line can be significant. In the following example, `mylib.a` precedes all source and object files on the command line:

```
cc mylib.a myprog.c -o myprog
```

If `myprog.c` calls a function contained in the `mylib.a` library, no executable program will be created.

Compiler fundamentals

2

This chapter discusses options that you can specify on the command line of the CONVEX C compiler and explains how they are used.

The CONVEX C compiler translates a program written in C into an object file that can be combined with other object files and executed on a CONVEX computer. Previously compiled programs written in CONVEX C, CONVEX Assembly Language, CONVEX ADA (C Series only), or CONVEX FORTRAN can be linked with CONVEX C object code to produce an executable program. In addition, the CONVEX C compiler running on an SPP Series (Exemplar) computer can link with objects and libraries produced under Hewlett-Packard Unix (HP-UX).

References are made to the Common C compiler. This compiler was formerly called the Portable C compiler and was delivered as part of ConvexOS. This compiler and previous releases of CONVEX C are no longer supported.

Compiler features

The current release of the CONVEX C compiler conforms to the ANSI C standard, as specified in the ISO/IEC 9899:1990 document, formerly ANSI X3.159-1989. Programs written for the ANSI C conforming mode of the CONVEX C compiler can be compiled by other compilers with little or no modification to the source code. The converse is also true; conformance to ANSI C specifications increases portability of C programs across different computer systems.

While the new C compiler supports the ANSI C standard, it can also be extended with other environments. The compiler:

- Is compatible with most UNIX C compilers
- Contains extensions that customize code for CONVEX computers

- Complies with IEEE Standard 1003.1-1988 (POSIX)
- Supports CONVEX SPP Series and CONVEX C Series platforms

On CONVEX SPP Series systems, floating-point numbers comply with the IEEE standard (IEEE/ANSI standard 754-1985).

Multiple compilers on a single host

CONVEX C supports multiple installed compilers on a single host. The system administrator can select the default compiler. Individual users can select their own defaults through the `CCOPTIONS` environment variable or through a command line option.

The command line components are installed in /usr/lib/cc*Ver*, where *Ver* is the version number of the compiler:

- cc
- lint
- cpp
- lint1
- lint2
- errmsg.cc
- errmsg.cpp
- errmsg.lint
- cocc

Links are made from /bin/cc, /usr/bin/lint, and /lib/cpp to the appropriate executable.

File naming conventions

The compiler discerns a file type by the extension added to the end of the file name. A C source file is identified by the extension `.c`. The compiled object file has the same name as the source file except that it ends with `.o`. The compiler can also generate symbolic assembly-language files, which end with the extension `.s` (see Table 1).

When you compile and load a single source file during one invocation of the compiler, the symbol and object files are normally deleted.

Unless you use the `-o` option to specify otherwise on the compiler command line, the executable module produced by the loader is placed into the file `a.out`.

Although the compiler normally produces object code directly on C Series machines, you can request that symbolic assembly code be generated. In this case, the name of the file that is produced is the same as that of the source file except it ends with `.s`.

On SPP Series systems, the compiler first generates symbolic assembly-language files (`.s`) then creates the object (`.o`) files. Both the .s and .o files are deleted after compilation by default.

Files specified to the CONVEX C compiler use the standard file extensions shown in Table 1.

Table 1 File-extension conventions

Files	File extensions
C source files	`.c`
Compiled object files	`.o`
Symbolic assembly-language files	`.s`
Library files	`.a`

Note

If you are compiling under COVUEshell, the extension `.c` identifies C source files; for example, `myfile.C` is a C source file in the COVUE environment. COVUE is available only on C Series machines.

Command line format

The format of the C compiler command line is illustrated below:

```
cc [options] files
```

where

options

is an optional argument specifying one or more of the options described in the remaining sections of this chapter.

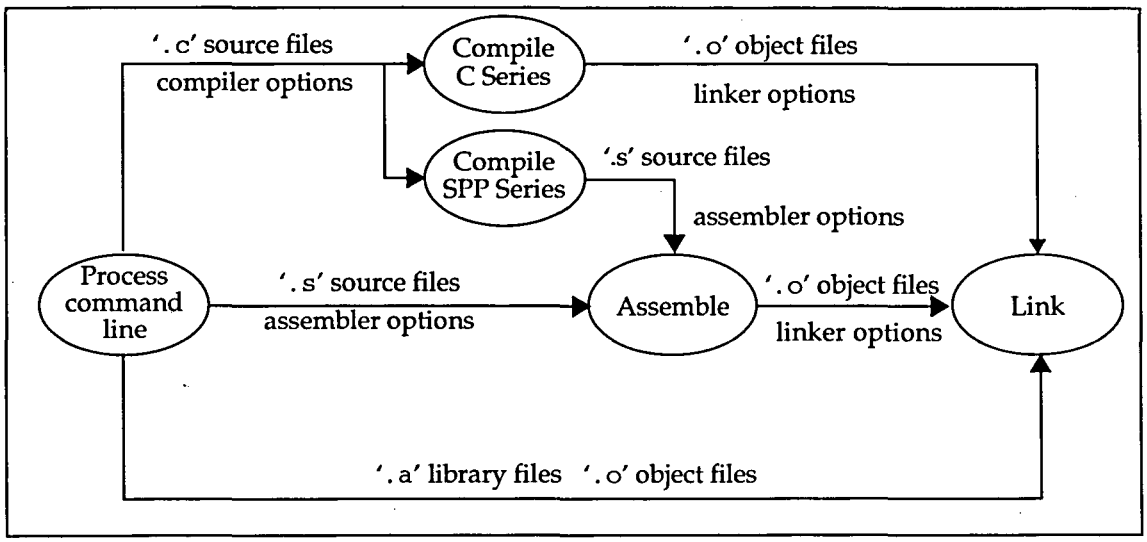
files

represents one or more source files to be compiled, object files to link, assembly-language files to be assembled, or libraries to link.

Compilation process

Figure 7 shows the compilation process. The text that accompanies each arrow indicates the destination of the options and files that are entered on the command line.

Figure 7 Compilation process



Options accepted by the assembler are common to the CONVEX C compiler and the CONVEX assembler. The *CONVEX Compiler Utilities User's Guide* describes C Series assembler options.

Compiler command line options

Compiler options include:

- Compatibility modes
- Preprocessor options
- Code generation options
- Diagnostic options
- Debugging options
- Optimization options
- Miscellaneous options

The following sections discuss each category of compiler options.

Note

Options that are specific to C Series or SPP Series machines are marked. These options are ignored as unrecognized options on the opposing machines.

Compatibility modes

The CONVEX C compiler can compile source code that conforms to the ANSI C standard, the POSIX standard, or both. Table 2 shows the four compatibility modes.

Table 2 Compatibility modes

Mode	Option	Language	Default functions
Extended (default)	`-ext`	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	`-std`	ANSI C	ANSI C, POSIX
Strict (C Series only)	`-str`	ANSI C	ANSI C
Backward-compatible	`-pcc`	Non-ANSI C	CONVEX

Choice of compatibility mode depends on the following:

- Portability of source code
- Existing source code
- Customization of source code for a system

For example, use backward-compatible mode for applications written for non-ANSI C compilers. Use strict mode for programs that must be ported to other systems.

Refer to the "Modes defined" section on page 51 for more information about modes and POSIX standard conformance.

Compatibility mode command-line options

Use these options to select the compatibility mode:

`-ext`

Selects extended mode. This mode provides an extended implementation of ANSI C and ANSI C-compatible and POSIX-compatible library extensions. It also defines the `_POSIX_SOURCE` constant. This is the default. Refer to the "Predefined symbols" section on page 37 for information on `_POSIX_SOURCE`.

`-std`

Selects standard conforming mode. The compiler acts as a conforming ANSI-C compiler. Certain extensions, notably the `long long` type, are not available in this mode. When linking occurs in this mode an ANSI C and POSIX P1003.1

conforming library system is used; many of the extensions provided by the default library system are not available.

C Series only

`-str`

Selects strict mode. The compiler operates as an ANSI C conforming compiler and attempts to detect features of the program which cause it not to conform to ANSI Standard C. When linking occurs in this mode, only the functions defined by ANSI Standard C are available.

`-pcc`

Selects backward-compatible mode. Forces language and library interpretation based on the original Kernighan and Ritchie definition. This mode is compatible with CONVEX C V3.0 and other C compilers previously shipped by CONVEX. This mode is also the most compatible with non-ANSI C compilers. Chapter 3, "Compatibility modes," explains how to compile a Common-C application in backward-compatible mode.

Note

The compatibility options are interpreted before all others, regardless of their position on the command line.

Example command lines

C Series only

On C Series machines only, to compile a source file that strictly conforms to the ANSI standard, type:

```
cc -str file.c
```

To compile a source file that contains extensions, type:

```
cc file.c
```

To compile an old (non-ANSI) CONVEX C program, type:

```
cc -pcc file.c
```

To compile a POSIX-conforming program, define the `_POSIX_SOURCE` constant on the command line:

```
cc -std -D_POSIX_SOURCE file.c
```

Preprocessor options

Preprocessing is the first stage of compilation. In addition to translating the same file into tokens, preprocessing also involves the following:

- Defining and expanding macro constants
- Conditional compiling
- Including source files

The preprocessor is also available as a standalone program. Refer to the cpp(1) and cc(1) man pages for additional information.

Several options control the preprocessor.

`-C`

Prevents the preprocessor from removing comments.

`-D`*name*

Defines the macro *name* with a default value of 1.

`-D`*name=def*

Defines the macro *name* as if by the `#define` preprocessor directive.

`-E`

Runs only the C preprocessor on the named C source files and sends the result to standard output.

`-I`*dir*

Names an alternate directory to search for include files. The compiler searches alternate directories in the order specified on the command line.

`-I-`

Inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include`. There is no way to override the effect of `-I-`.

`-k`

Preprocesses specified source files and generates dependency descriptions for `make(1)`; prints results on the standard-output stream. Files that are not C source files are ignored.

`-P`

This option gives you preprocessed code without the control lines. It does not allow you to redirect the output; the output is placed in a file in the current directory with the same root name as the source file name and a suffix of '`.i`'.

`-U`*name*

Removes any initial preprocessor definition of *name*. Predefined identifiers of CONVEX C are described later in this chapter.

Debugging option usage

The following example demonstrates how the `-D` option can be used effectively. Debugging code is included in a program to help locate errors. The preprocessor can be used to eliminate such code before it is compiled. For example, the following code is compiled only if the manifest constant DEBUG exists:

```
#ifdef DEBUG
printf("The file just written to is %s\n",
        filename);
#endif
```

To activate the debugging code, use the command line

```
cc -DDEBUG prog.c
```

where `prog.c` is the source file that contains the example. Using this option is easier than defining the constant DEBUG within the source code itself.

Creating dependencies

For large programming projects with many source files and header files, relationships between the files might not be obvious. In such cases, the `-k` option is useful. This option creates a list of dependencies that can be used in a makefile. The "make utility" section on page 74 discusses the format of files that the make utility uses.

Code generation options

The compiler includes some options that affect the code that is generated. For example, one option permits files to be compiled only; they are not linked after they are compiled. Other options control the format of floating-point numbers in the program. Code generation options are as follows.

`-align {cseries | spp}`

Causes alignment of members of structs and unions to conform to the target machine. On C Series machines, `-align cseries` is the default. On SPP Series machines, `-align spp` is the default. For example, if you are transferring data structures originating on a C Series machine onto an SPP Series system, you would use:

```
-align spp
```

`-c`

Suppresses the linking phase of compilation. The object file that is generated from the file `file.c` or `file.s` is left in `file.o`.

Note

Because both cc and as place the object file in `file.o`, source files for assembly language and C must not have the same file name stem (file name without the extension).

C Series only

`-compat rrf=`*option*

The CONVEX C compiler ordinarily returns `unions` and `structs` using memory allocated by the called function. CONVEX C returns `unions` and `structs` using memory allocated by the calling function. Because this can cause problems with certain coding constructs, CONVEX C provides a (stack-based) `union`- and `struct`- return mechanism.

`-compat rrf=stack`

You can compile a program to use the stack return mechanism by specifying `rff=stack`.

`-compat rrf=old`

You can explicitly request the old return mechanism by specifying `rrf=old`, but you do not need to do so. The compiler uses the `old` return mechanism by default.

The two `struct` return mechanisms are not compatible. If a function returns a `union` or `struct` to another function, both must be compiled to use the same mechanism.

To identify code that may not compile properly with the old return mechanism, use these options:

```
-d record_fn_defn=w
```

tells the compiler that function definitions may not compile properly.

`-d record_fn_call=w`

tells the compiler that function calls may not compile properly.

`-extern {distinct | same}`

The `-extern` option can be used to control the interpretation of a program in which two different files declare an uninitialized variable (for example, `int i;`) at file scope.

`distinct` The files refer to two distinct variables; the linker detects a multiply-defined symbol. This is the definition provided by the ANSI C standard. It is traditional on AT&T UNIX System V.

`same` Both files refer to the same variable. This is the default in all modes; it is traditional on BSD UNIX systems and is a conforming extension to Standard C.

`-fd`

This is a synonym for the `-float sp_ops` compiler option.

C Series only

`-fi`

Translate floating-point values into IEEE format and process in IEEE mode. This option requires that the machine be equipped with IEEE support hardware. If no floating-point format is specified, the site default is used.

`-float {dp_const | sp_const | dp_ops | sp_ops}`

`dp_const` Translate all unsuffixed floating-point constants into `double` data types. This is the default action of the compiler.

`sp_const` Translate all unsuffixed floating-point constants into `float` data types.

`dp_ops` Generate code to perform operations on 32-bit floating-point values using 64-bit instructions. The increase in precision can cause the program to run more slowly. This is the default in the backward-compatible mode.

`sp_ops` Generate code to perform operations on 32-bit floating-point values using 32-bit instructions rather than 64-bit instructions. The program can run faster, but results might not be as precise as

those using 64-bit calculations. This is the default in the strict, conforming, and extended modes of CONVEX C.

C Series only

`-fn`

Translate floating-point values into native CONVEX format and process in native mode. If no floating-point format is specified, the site default is used.

C Series only

`-mi` *n*

Specifies the expected memory interleave on the target machine. The argument *n* is an integer representing the expected memory interleave. When this option does not appear, the compiler uses the interleave of the machine on which the compiler is running. You can use the `getsysinfo` command to determine memory interleave on a machine.

`-parens {explicit | ignore | implicit}`

The `parens` option affects the interpretation of parentheses used in floating-point operations.

`explicit` Parentheses are honored regardless of the compatibility mode.

`ignore` Parentheses are ignored; the compiler can reorder any floating-point expression according to rules of associativity and commutativity. This is the default in the backward-compatible and extended modes.

`implicit` The compiler honors all explicit parentheses, as well as those implied by grammatical precedence and associativity rules; no reordering can be performed. This is the default for the strict and conforming modes.

`-S`

Generate assembly code for each program unit in a source file. Assembler output for source file `file.c` is put in file `file.s`; no object file is generated and the linker is not invoked.

`-string {read_only | read_write}`

Determines whether string literals can be modified.

`read_only`
The program may not modify string literals. This is the default in the strict, conforming, and extended compatibility modes of CONVEX C.

`read_write`
String constants may be modified. This is the default in the backward-compatible mode of CONVEX C.

`-tm` *target*

Specifies the target machine architecture. The default value is the type of machine hosting the compiler. Table 3 shows `-tm` *target* options and corresponding target architectures.

Table 3 Target machine architectures

target	Target architecture
`c1, C1`	C1 Series computer
`c2, C2`	C2 Series computer
`c32,C32`	Either a C3200 Series computer or a C200 Series computer equipped with scalar accelerator hardware. (Use the `getsysinfo` command to determine whether your C2 computer has this hardware.)
`c34,C34`	C3400 Series computer
`c34j,C34J`	C3400-ES Series computer
`c38,C38`	C3800 Series computer
`c4`	C4600 Series computer
`spp1`	SPP1000 (Exemplar) system. This option is not recognized on C Series machines.

Note

`-tm spp1` is the *only* valid option on SPP Series systems. There are no cross-compiler capabilities between C Series and SPP Series systems.

The `-tm` option is useful, for example, when you are compiling a program for a C2 computer on a C1 computer. The command line

```
cc -tm C2 prog.c
```

compiles a version of program `prog.c` for use on a C2 computer. The resulting program will not run on a C1 computer.

When invoking the linker, the compiler uses machine-dependent versions of some libraries:

- If you do not specify a target machine, the compiler generates instructions for the class of machine on which it is running.
- If you do specify a target machine, the compiler uses that machine's instruction set regardless of the machine on which the compiler is running.

Diagnostic options

These options produce additional information that can be used to judge the quality of the source file. Different options can be selected for varying levels of detail.

`-d` *name*`[=[w|e]]`

Control issuance and severity of messages represented by *name*. This option can be used to convert a warning message to an error message and vice versa. (Appendix E, "Error messages," lists warning and error messages.) If *name* is followed by `=w`, it is converted to a warning message. If *name* is followed by `=e`, it is converted to an error message. If *name* is not followed by `=w` or `=e`, the diagnostic message associated with *name* is suppressed.

`-nv`

Synonym for `-or none`.

`-nw`

Suppresses all warning messages.

`-or [all | none | array | private | loop]`

Specifies the contents of the optimization report to be produced. For details, see the discussion of the `-or` option on page 32.

`-sc`

Instructs the compiler to check the program for errors without performing any optimization or code generation.

The `-sc` option is useful during the initial development of a program. This option reduces the load on a system by checking only for syntax errors.

Debugging options

Several debuggers and profilers are available. Each of these tools requires the inclusion of extra information, called instrumentation, in the compiled code to permit suitable interpretation of the program by the support tools.

The debugging and profiling options include:

`-cxdb`

Generates symbolic debugging information for CXdb, the CONVEX visual debugger. You can use CXdb to debug programs compiled without the `-cxdb` option, but symbolic debugging information will not be available. CXdb is capable of debugging optimized code and this option can be used with all levels of optimization.

When this option is included on the command line, a subdirectory called `.CXdb` is created in the current working directory that contains auxiliary debugging information. Refer to the "CXdb debugger" section on page 78 or to the *CONVEX CXdb User's Guide* for more information. CXdb is an optional product.

`-cxpa`

Produces counting code for routine-level and loop-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

This option can be used with all levels of optimization and with the `-db` option. It should not be used with the CXpa block-level profiling option, `-pab`.

`-cxpab`

Produces counting code for block-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

`-cxpalib`

Instructs the compiler to link your program with the system installed libraries that are instrumented for use with CXpa. Programs profiled with `-cxpalib` may execute much more slowly. Any errors in the instrumentation of the libraries may cause the program to core dump. The amount of profiling data under this option is significantly increased.

`-cxpamon="dirname"`

Allows CXpa users to select a specific monitor instrumentation library by specifying the directory where the library resides. The full path of directory dirname is specified surrounded by quotation marks and indicates the directory path where the file `cxpamon.o` exists. The `-cxpamon` option facilitates using multiple versions of CXpa on a system.

`-cxpar`

Includes instrumentation for routine-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

C Series only

`-metrics`

Writes the CXmetrics data file *sourcefile*`.met` in the same directory as the source file. *sourcefile* is the name of your original C source file without the `.c` extension. *sourcefile*`.met` is a human-readable ASCII file used by CXmetrics to generate reports containing analytical data about the relative complexity of the program. Refer to the *CONVEX CXmetrics User's Guide* or to the metrics(1) man page for more information. CXmetrics is an optional product.

`-p`

Produces code that counts the number of times each routine is called. Profiled versions of system libraries are used instead of default libraries. If linking takes place, this option replaces the standard start-up routine with one that automatically calls monitor(3) at the start and arranges to write out the file `mon.out` when the object program terminates. You can then generate an execution profile using prof. The prof profiler is part of the optional CONVEX Consultant package.

C Series only

`-pb`

Includes instrumentation that produces an execution profile named `bmon.out` at normal termination. You can then obtain listings of source-level execution counts using `bprof`. The `bprof` profiler is part of the optional CONVEX Consultant package.

`-pg`

Includes instrumentation in the manner of `-p`, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a file named `gmon.out` at normal

termination. You can then generate an execution profile using gprof. The gprof profiler is part of the optional CONVEX Consultant package.

Optimization options

Several compiler options affect optimization. For more detailed discussion of optimization, refer to the *C Optimization Guide* for C Series information or the *Exemplar Programming Guide* for SPP Series information.

CONVEX produces two versions of CONVEX C for C Series machines: a scalar optimizing version and a vectorizing/parallelizing version, which is an optional product.

The scalar optimizing version of the compiler does not perform vector and parallel optimizations; it has a maximum optimization level of `-O1`. The scalar compiler ignores optimizing compiler options that are not supported.

The SPP Series C compiler is a single version which allows scalar and parallel optimizations.

Below are optimization options for CONVEX C.

`-alias array_args`

Assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but can allow greater optimization to occur, particularly vectorization on C Series machines. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to the elements of the formal parameter (for example, `formalParameter[10] = x[10]`).

The following source code illustrates a situation in which the `-alias array_args` option can be used:

```
void vaf( char [] );
char global = 'A';

int main()
{
   char b[10];

   vaf( b );
   return(1);
}

void vaf( char array[] )
{
   int i;

   for( i=0; i<10; i++ )
      array[i] += global;
}
```

C Series only

The loop in the `vaf` function is vectorized when the `-alias array_args` option is specified because the compiler can assume that the address of `global` is not the address of an element of `array`.

`-alias {cautious | standard | worst}`

The `-alias` option affects the assumptions the compiler makes regarding overlapping of variables, as when referenced by a pointer.

`cautious` Tries to compile with `-alias standard`, but if inappropriate constructs are found, compiles with `-alias worst` instead. This is the default in the ANSI C-compatible modes.

`standard` Performs aliasing based on assumptions permitted in ANSI C; pointers of one object type can only reference objects of that same type. For example, a pointer to an object of type `float` cannot reference an object of type `int`. Such assumptions permit additional optimizations. *The compiler generates incorrect code when these assumptions are not true.*

`worst` Performs pointer aliasing based on the assumption that pointers can modify objects of

any type. This is the default in the backward-compatible mode.

The following command line creates an executable program that uses the worst-case aliasing algorithm for compiling a program:

```
cc -alias worst file.c
```

The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`.

`-alias {no_global | global}`

When you specify `-alias no_global`, the compiler assumes that globals (external variables) are not accessed via pointers. The default is `global`.

The following would be an incorrect use of the option because the value of the global is accessed through `pointer` and `global`.

```
int global;

int *pointer;

bar()
{
   pointer = &global;
}

main()
{
   int tmp;

   bar();
   global = 1;
   *pointer = 2;
   tmp = global;
   printf("%d\n", tmp);
}
```

The program prints `1`.

`-alias {no_addr | addr}`

`-alias no_addr` indicates that address operands do not introduce aliases. Useful when objects are passed by reference but no aliases are produced. The default is `addr`.

The optimization report sometimes gives a "`no induction variable`" message.

The example below illustrates the result of using `-alias no_addr`.

```
#include <math.h>

extern int *p;
foo(float *a, float *b)
{
   int n, i;

   n = 100;

   /* Bar may change n but not assign the
      address to a pointer */
   bar(&n);

   for (i = 0; i < n; i++) {
   a[i] = sin(b[i]);
   *p++ = 1;
   }
}
```

`-alias ptr_args`

Assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that the function uses (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This feature conflicts with the language definition, but allows greater optimization to occur, particularly vectorization. You cannot use this option if the formal parameter itself is assigned by the function:

```
formalParameter = &x[10];
```

You can assign values to variables identified by the formal parameter:

```
formalParameter[10] = x[10];
```

`-alias restrict_args`

This option tells the compiler to treat the code as though it applied a `restrict` qualifier to each pointer parameter. The `restrict` qualifier tells the compiler that a pointer provides exclusive access to the data object at a given

memory location. When you use the `restrict` qualifier, you must access the object pointed to by a `restrict` pointer only through that pointer. If you reference the object by some other means, the compiler may incorrectly optimize code.

For more information on aliasing, refer to the *CONVEX C Optimization Guide* and the *Exemplar Programming Guide.*

SPP Series only

`-blockloop` *n*

Instructs the compiler to block *m-1* loops in an *m* loop nest. The compiler uses a block factor of *n* and automatically blocks only loops which will benefit from blocking. (For detailed information about loop blocking, consult the *Exemplar Programming Guide.*) Loop blocking is provided at optimization levels `-O2` and `-O3` unless you specify the `-noblock` option.

SPP Series only

`-br`

Enables base register optimizations. This option is the default on SPP Series machines. To disable base register optimizations use the `-nbr` compiler option.

SPP Series only

`-cache` *n*

Compile for a processor data cache size of *n* kilobytes. The size must be an integer >0.

C Series only

`-ds`

Causes the compiler to perform dynamic selection on loops the compiler selects on the basis of profitability. Multiple copies of the loop running sequential, vector, or parallel modes are generated; the optimal version is executed based on the trip count of the loop. This option is effective only at levels `-O2` and `-O3`.

C Series only

`-ep` *n*

Optimizes the code for *n* processors, but does not prevent the code from running on other system configurations. Single-program performance increases at the expense of system throughput. *n* must be in the range 1 through 8. It is effective only when you specify the `-O3` option. Refer to the *CONVEX C Optimization Guide* for additional information on using this option.

The following command line optimizes the C program `file.c` for use on a C240 machine:

```
cc -O3 -ep 4 -tm C2 file.c
```

The executable program created by this command line takes advantage of the instruction set and the availability of four processors of the C240 machine to increase program execution speed.

C2, C3, C4 Series

`-except {precise | default}`

The `-except` options concern the treatment of arithmetic exceptions generated within functions. The command line options are `-except precise` and `-except default`.

The `-except precise` option generates code that ensures that the program receives any arithmetic exceptions generated within functions before the return. Without `-except precise`, there is a small possibility that the location of an arithmetic exception might be reported incorrectly or lost.

The code generated under `-except precise` is specific to the target architecture for which you are compiling and is only guaranteed to work for that architecture. The target architecture is determined by the `-tm` option, or, in absence of `-tm`, defaults to the machine on which you are compiling.

Use `-except precise` only when absolutely necessary as it causes additional instructions to be inserted before every return, and this will degrade performance.

`-except default` cancels the effects of `-except precise`.

On SPP Series systems,

```
cc -except precise filename.c
```

causes a warning message:

```
Unknown flag '-except' ignored
```

The `precise` keyword is passed to the linker.

SPP Series only

`-mo`

Enables the generation of multi-op instructions. The `-mo` option is the default on SPP Series machines. To prevent the generation of multi-op instructions specify the `-nmo` option.

SPP Series only

`-nbr`

Disables base register optimizations. By default, CONVEX C on SPP Series machines provides base register optimizations.

SPP Series only

`-nga`

Disables global register allocation for arguments passed by reference. Global register allocation is enabled by default.

SPP Series only

`-ngs`

Disables global register allocation for shared memory variables which are visible to multiple threads. Global register allocation is enabled by default.

SPP Series only

`-nmo`

Disables the generation of multi-op instructions. Multi-op instructions are generated by default on SPP Series machines (and via the `-mo` option).

`-no`

No machine-independent optimization is performed; this is the default, but you can override it with the `-O`*n* options.

`-noautopar`

Parallelizes loops only if a parallelizing pragma (such as `prefer_parallel` or `begin_tasks`) precedes them to request parallelization. All other loops are treated as if the `no_parallel` pragma were specified for them. This option affects optimization level `-O3`.

Under `-noautopar`, loop interchanges for parallelization occur only for loops that are affected by parallelization pragmas and are parallelized. This option does not interfere with loop blocking and other level `-O2` interchanges.

C Series only

`-noautovec`

Vectorize loops only if a vectorizing pragma (for example, `prefer_vector` or `force_vector`) precedes them to request vectorization. This option is effective at optimization levels `-O2` and `-O3`.

The compiler still interchanges loops that otherwise would have been fully (100%) vectorizable because of pragma specifications or dependency deductions. Interchanges are not performed for partial vectorization. Any loop that is interchanged is vectorized only if a vectorization pragma applies to the loop.

SPP Series only

`-noblock`

Prevents the compiler from blocking loops in the specified source files. Because loop blocking occurs at optimization levels `-O2` and `-O3`, the `-noblock` option is effective only at

these levels. This command is the same as specifying the `no_block_loop` pragma on all loops.

`-nopeel`

Disallows loop boundary value peeling, which is on by default at optimization levels `-O2` and `-O3`. Refer to `-peel` and `-peelall` below.

C Series only

`-nopm`

This option disables the pattern matching which the compiler uses to aid in vectorization. Normally, the compiler uses pattern matching to vectorize loops not otherwise vectorizable.

`-noptst`

Disallows test promotion, which defaults to on at optimization levels `-O2` and `-O3`. Refer to `-ptst` and `-ptstall` below.

`-nore`

Specifies that the code is not reentrant. This option overrides the default (`-re`) on SPP Series systems.

Refer to the *Exemplar Programming Guide* for more details on reentrant procedures on SPP Series machines.

The `-nore` option is the default on C Series computers.

`-nptr`

Disable procedural pointer tracking. Pointer tracking reduces aliasing on pointers by keeping track of the possible objects that each pointer could reference.

Pointer tracking is performed at optimization levels `-O1` and above.

`-nsr`

Disables scalar replacement. Scalar replacement is enabled by default. Scalar replacement can occur at optimization levels `-O2` and `-O3`.

`-nuj`

Disables the loop unroll and jam transformation. On SPP Series machines, unroll and jam is enabled by default. When enabled, this transformation affects optimization levels `-O2` and higher. This option is the default on C Series machines.

`-nur`

Disables loop unrolling. On SPP Series machines, loop unrolling is enabled by default. Unrolling affects optimization levels `-O2` and `-O3`.

`-O`

Performs the highest scalar optimization level available. On C Series machines, selects the `-O1` optimization level. On SPP Series machines selects optimization level `-O2`.

`-On`

Performs machine-independent optimizations, level *n* (see Table 4).

Table 4 Optimization options for C Series and SPP Series machines

Option	C Series	SPP Series
n=0	basic block scalar optimization	basic block level scalar optimizations
n=1	`-O0` plus global scalar optimization	`-O0` plus program unit level optimizations and global register allocation
n=2	`-O1` plus vector optimizations	`-O1` plus global instruction scheduling, software pipelining, and data localization optimizations
n=3	`-O2` plus parallel optimizations	`-O2` plus parallel optimizations

Note

The `-O3` option is not available on the C1. `-O3` can degrade performance if the program is executed on a single CPU.

`-or {all | array | private | loop | none}`

Specifies the contents of the optimization report:

`all`	Display all reports.
`array`	Display the array table.
`private`	Report which user variables were privatized for parallelization and the type of privatization.
`loop`	Display the loop table.
`none`	Display no reports.

`-peel`

Causes removal of the first and/or last iterations of a loop when doing so removes conditional tests from the loop. This

is typically possible when the loop contains a test that always evaluates to zero or a nonzero value for the first and/or last iteration. By default, the compiler peels boundary values and expands code up to a limit. With the `-peel` option, this limit is increased, and code expansion may become significant. The `-peel` option works only with the `-O2` and `-O3` options. Refer to the *C Optimization Guide*, Chapter 3 for more information on `-peel`.

`-peelall`

Same as `-peel`, but allows code expansion without bound. For code containing large numbers of boundary value operations, this can increase the size of the code enough to flag compile-time errors. The `-peelall` option is effective only with the `-O2` and `-O3` options.

`-ptst`

Causes a test to be promoted out of the loop that encloses it by replicating the containing loop for each branch of the test. By default, the compiler replicates code up to a limit. With the `-ptst` option, this limit is increased and code expansion may become significant. The `-ptst` option is effective only with the `-O2` and `-O3` options.

`-ptstall`

Same as `-ptst`, but allows code replication without bound. For loops containing large numbers of tests, this can increase the size of the code enough to cause compile-time errors. The `-ptstall` option is effective only with the `-O2` and `-O3` options.

`-re`

On both C Series and SPP Series systems, `-re` generates reentrant code by creating both a scalar and parallel version of parallel loops. The default, at optimization level `-O3`, generates nonreentrant code for functions with parallel regions. This option has no effect on functions without parallel code. Use this option to compile a function called by parallelized regions of code. It is always safe to use `-re`; however, the text segment can be unnecessarily large if you use `-re` when it is not required.

The `-re` option is the default on SPP Series systems. SPP Series users should consider the following:

- The compiler generates re-entrant code for recursive parallel invocation of subprograms.
- You should not use static declarations in your code.

- You must initialize all local variables that are not shared.
- Each invocation of the subprogram contains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary values.
- If loops are parallel, each loop has a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary values.

`-rl`

This option is equivalent to `-ds -ur`.

`-sr`

Enables scalar replacement, which involves storing loop-invariant array references in registers throughout a loop, thus avoiding slower main memory accesses. This option is enabled by default. When enabled, scalar replacement occurs at optimization levels `-O2` and `-O3`.

`-uj`

Enables the loop unroll and jam transformation, whose goal is to exploit the use of registers and thus decrease the number of slower main memory accesses. On SPP Series machines, loop unroll and jam is effective by default. Unroll and jam affects optimization levels `-O2` and `-O3`. This transformation is disabled via the `-nuj` option. To specify the loop unrolling factor for this transformation use the `-ujn` *n* option. For more information consult the *Exemplar Programming Guide*.

`-ujn` *n*

Enables the loop unroll and jam transformation and specifies the desired loop unrolling factor (*n*, where *n* is the number of times to replicate the body of the loop). This option is effective at optimization levels `-O2` and `-O3`.

`-uo`

Performs potentially unsafe optimizations. For example, the `-uo` option can move the evaluation of common subexpressions and invariant code from within conditionally executed code. The code is then executed unconditionally. For more information, see the *C Optimization Guide*, Appendix A and the *Exemplar Programming Guide*, Chapter 3.

`-ur`

Causes the compiler to perform loop unrolling on loops the compiler selects on the basis of profitability. On SPP Series

machines, this option is the default. Loop unrolling occurs at optimization levels `-O2` and `-O3`.

`-urn` *n*

Causes the compiler to automatically select and unroll loops, and use a loop unrolling factor of *n*, where *n* is the number of times to replicate the body of the loop. This option is effective at optimization levels `-O2` and `-O3`.

`-va`

This option is a synonym for `-alias array_args`.

Miscellaneous options

`-altcc` *path*

This option invokes an alternate compiler driver. For example,

```
cc -altcc /usr/lib/newcc/cc bar.c
```

is equivalent to

```
/usr/lib/newcc/cc bar.c
```

`-B`*dir*

Looks for a substitute compiler in the directory *dir*. If *dir* is not specified, the backup compiler in the directory /usr/lib/oldcc is used. For example, the command

```
/usr/new/cc -B/usr/new
```

invokes the compiler in /usr/new instead of the default, /usr/lib/cc.

`-o` *name*

Specifies that *name* is the name of the executable file produced by `ld`. If this option is not specified, the default name is `a.out`. If `-c` is specified and there is only one file to compile or assemble, *name* is the name of the object file produced.

C Series only

`-tl` *time*

Sets the maximum CPU time limit on compilations to *time* minutes. If the CPU exceeds the allotted time, the compilation is aborted.

`-vn`

Identifies the compiler version to standard error.

`-W`*subproc*`,`*arg1*`,`*arg2*`, . . .`

Pass the option(s) to subprocess *subproc*. The arguments take the form `-`*opt*`[ ,` *argvalue*`]`, where *opt* is the name of an option recognized by the subprocess and *argvalue* is a separate argument to *opt* as needed.

The `-W` option specification allows additional, implementation-specific options to be recognized by the compiler driver.

The following values are recognized for *subproc*:

`p`	preprocessor
`c`	compiler
`0`	compiler
`a`	assembler
`l`	linker

Compatibility with options of other compilers

To improve portability of `make` files and to enhance compatibility with C compilers supported by other compiler vendors, CONVEX C recognizes several additional compiler options. Table 5 lists these options and how the CONVEX C compiler interprets them.

Table 5 Option compatibility

Option	Interpretation
`-g`	A synonym for the `-db` option
`-n`	Ignored with a warning
`-O`	On C Series, a synonym for the `-O1` option. On SPP Series, a synonym for the `-O2` option.
`-OL`	A synonym for the `-O1` option
`-V`	A synonym for the `-vn` option
`-w`	A synonym for the `-nw` option

In Table 6, the flags listed in the first column generates a diagnostic on CONVEX SPP Series systems. Flags listed in the second column generate a diagnostic on C Series machines.

Table 6 Flags that generate diagnostics

C Series	SPP Series
`-except *`	
`-compat rrf=*`	
`-tm[c1, c2, c3, c32, c34, c34j, c38, c4, p53, p54, p55, p56]`	`-tm spp1`
`-ep` *n*	
`-O2`	
`-fn, -fi`	
`-mi` *n*	
`-str`	
`-db, -g`	
`-pb`	
`-tl` *n*	

Predefined symbols

The macros listed in this section are predefined and have special meaning to the CONVEX C preprocessor.

Note

"__" indicates two adjacent underscore characters. There is no space between these characters. If a space is added the compiler will not recognize the variable as a predefined symbol.

`__convex__`

This symbol is always defined when using CONVEX compilers.

`__convexc__`

This symbol is always defined. It is used to identify the CONVEX C compiler.

__convex_c1
__convex_c2
__convex_c32
__convex_c34
__convex_c38
__convex_c4
__convex_spp
__convex_spp1

These symbols identify the target architecture. __convex_spp symbolizes the entire SPP Series family, while __convex_spp1 specifically refers to the SPP1000 (Exemplar) machine.

_CONVEX_EXT
_CONVEX_PCC
_CONVEX_STD
_CONVEX_STR

These symbols identify the mode of the compiler.

C Series only

_CONVEX_FLOAT

The compiler defines _CONVEX_FLOAT when the compiler operates in native floating-point mode. For more information, refer to the description of the -fi and -fn options in the "Code generation options" section on page 16.

_CONVEX_SOURCE

This symbol is predefined in extended mode. In the conforming mode (-std) you will usually want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.

__DATE__

The date of translation of the source file. It is a character string literal of the form *"mmm dd yyyy"*. This symbol is not available in the backward-compatible mode.

__FILE__

The name of the source file being compiled. This symbol is predefined in all modes. You can modify this macro with the #line preprocessor directive, but not with the #define preprocessor directive or -D command-line option.

SPP Series only

__hppa

The compiler defines __hppa when running on the HP Performance Architecture.

SPP Series only

__hpux

This symbol is defined for HP-UX compatible OS.

SPP Series only

_hp9000s800

This symbol is defined for HP compatibility.

SPP Series only

_HPUX_SOURCE

This symbol is defined to enable HP extensions. This symbol is predefined only in the extended mode.

_IEEE_FLOAT_

The compiler defines _IEEE_FLOAT_ when the compiler operates in IEEE mode. Refer to the description of the `-fi` and `-fn` options in the "Code generation options" section on page 16 for more information. This is the only mode available for SPP Series.

__LINE__

The line number of the current source line. This is predefined in all modes. You can modify this macro with the `#line` preprocessor directive, but not with the `#define` preprocessor directive or `-D` command-line option.

__NO_INLINE_MATH

The symbol __NO_INLINE_MATH is defined in `-str` and `-std` modes to suppress recognition of macros which define certain library functions. This is necessary to get conforming implementations of some functions which would otherwise not set `errno` in a standard conforming manner. Refer to intro(3m) for more details.

_PA_RISC1_1

This symbol identifies a specific version of the PA_RISC architecture for CONVEX SPP Series and HP machines.

_POSIX_SOURCE

This symbol is predefined in extended mode. In the conforming mode (`-std`) you will usually want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.

SPP Series only

_REENTRANT

This symbol is predefined for use by the include files. When it is predefined, reentrant versions of libc routines are called. When REENTRANT is not predefined, some libc routines that are not reentrant will be called. Calling a non-reentrant routine from within a parallel region is an error.

`__STDC__` or `__stdc__`

This symbol is defined when you compile in an ANSI C mode (`-str`, `-std`, or `-ext`). It indicates that the compiler is an ANSI C-style compiler, but that it is not conforming, due to CONVEX extensions.

You can only remove the `__STDC__` definition with the `-U` option or `#undef` in the `-ext` mode.

`__TIME__`

The time of translation of the source file. It is a character string literal in the form *"hh:mm:ss"*. This symbol is not available in the backward-compatible mode.

`__unix__`
`__unix`

These symbols are defined in all compatibility modes.

The compiler can predefine other names beginning with a double underbar (__). Such names are reserved to CONVEX; their use or availability can change in subsequent releases. Avoid writing applications that depend on the *presence* or *absence* of names beginning with "__" (except those defined above).

Table 7 shows where predefined symbols are defined. In this table,

C	means the symbol is defined by the compiler driver
P	means the symbol is defined by the preprocessor
L	means the symbol is defined by the `lint` driver

Table 7 Where predefined symbols are defined

Symbol	Compilation mode		
	`-pcc`	`-ext`	`-std`
`__convex__`	C L	C L	C L
`__convexc__`	C L	C L	C L
`__convex_c1`	P	P	P
`__convex_c2`	P	P	P
`__convex_c32`	P	P	P
`__convex_c34`	P	P	P

Table 7 **(continued)** Where predefined symbols are defined

Symbol	Compilation mode		
	-pcc	-ext	-std
__convex_c38	P	P	P
__convex_c4	P	P	P
__convex_spp	P	P	P
__convex_spp1	P	P	P
__CONVEX_EXT		P	
__CONVEX_FLOAT	CL	CL	CL
__CONVEX_PCC	P		
_CONVEX_SOURCE		CL	
__CONVEX_STD			P
__DATE__		P	P
__FILE__	P	P	P
__hp9000s800	CLP	CLP	CLP
__hppa	CLP	CL	CL
__hpux	CL	CL	CL
_HPUX_SOURCE		CL	
_IEEE_FLOAT	CL	CL	CL
__LINE__	P	P	P
__lint	L	L	L
__NO_INLINE	L	L	L
__NO_INLINE_MATH			C
_PA_RISC1_1	CL	CL	
_POSIX_SOURCE		CL	
_REENTRANT	P	P	P
__STDC__, __stdc__		P	P
__TIME__		P	P
__unix	P	P	P
__unix__	P	P	P

Linker use

The `cc` command invokes the linker. On C Series machines, `cc` invokes the linker directly. On SPP Series systems, `cc` generates symbolic (`.s`) code and passes it to the assembler (`as`) to generate object code (`.o`), which then goes to the linker.

`cc` passes some compiler options directly to the linker. It also lets you specify additional linker options.

Compiler options that affect linking

Several compiler options translate into linker options.

- On C Series machines, the compiler always passes the flags `-X`, `-NL`, and `-L/usr/lib` to the linker. On SPP Series, the compiler only passes the `-L/usr/lib` flag.

C Series only

- On C Series machines, the compiler passes the flags `-fi` and `-fn` to the linker.

C Series only

- The compiler passes `-Eposix` to the linker except when `-pcc` is given, in which case it passes `-Enoposix`.
- The compiler compatibility mode controls the libraries searched by the linker; you normally control this by passing one or more options to the linker via the `-W` compiler option.

Note

CONVEX recommends that you always use the appropriate compiler to invoke the linker, rather than invoking it directly, to insulate the program from changes in library structure when new compiler releases are installed.

Additional linker options

You can also include additional linker options on the `cc` command line. The compiler accepts the following flags and passes them to `ld`:

```
-L -l -s
```

Remember the following when using these flags:

- Do not insert a space between `-l` and its argument.
- For the `-l` option to be effective, you must specify it after all object files on the `cc` command line.
- For C Series machines, do not use `-lc` and `-pcc` on the same command line; they are incompatible.

Specify the other linker flags (below) using the construction `-Wl,`*arg* where *arg* is the flag.

```
-A -D -E -F -H -M -NL -R -T -X
-e -m -r -t -u -x -y
```

For instance, to use the `-D` option, you should pass `cc` the flag `-Wl,-D`. If you include one of the flags above without using the `-Wl,`*arg* construction, both the C Series and SPP Series compilers generate warning diagnostics similar to the one below:

```
*The -ld flag -D is obsolete; use -Wl,-D instead*
```

The *CONVEX Compiler Utilities User's Guide* and the ld man page describe the meaning of these flags.

Notes

The following flags are valid only for C Series compilers:

```
-F -H -M -NL
```

The following flags have different meanings on C Series and SPP Series systems:

```
-A -m -R -T -X -E
```

Though CONVEX recommends that you use the `-Wl,`*arg* construction, you may also use the `-link` *arg* construction to pass arguments to the linker. In this construction, *arg* is a single argument to pass to `ld`. If the argument does not start with `-` or starts with `-l`, it is added to the linker's file list; otherwise it is added to the linker's flag list.

Environment variables

This section describes the following environment variables.

- `CCOPTIONS`
- `CPPOPTIONS`
- `LINTOPTIONS`
- `CCLIBS`
- `TMPDIR`

You can use the `CCOPTIONS` and `CCLIBS` environment variables to preset any of the CONVEX C compiler options.

The options are a space-separated list. Any options that appear before an optional vertical bar (`|`) are processed before the

command-line arguments. Any options that appear after the vertical bar are processed after the command-line arguments. If there is no vertical bar, all options are processed before the command-line arguments.

If there is a conflict, the command line options take highest priority, and options specified with CCOPTIONS take least priority (except when using the vertical bar).

For example,

```
% setenv CCOPTIONS "-O2 | -va"
% cc foo.c
```

has the same effect as

```
% cc -O2 foo.c -va
```

The stand-alone preprocessor (cpp) has a similar variable, CPPOPTIONS, that can be used to affect its behavior. However, the CPPOPTIONS environment variable does not have any effect when the compiler invokes its internal C preprocessor.

With the previous value of CCOPTIONS, the following command compiles the file my_prog.c with an optimization level of -O1 and causes a warning message to appear:

```
% cc -O1 my_prog.c
Contradictory optimization level specifications given --
believed '-O1'.
```

You can specify an alternate directory for the compiler to use for its temporary files through the TMPDIR environment variable.

Compiler messages

This section describes messages the compiler displays during compilation or runtime. These messages are grouped into the following categories:

- Diagnostic messages
- Optimization report
- Runtime messages produced by the backward-compatible mode

Diagnostic messages

If the compiler detects an error or a condition that requires an advisory, a suitable message is directed to standard error

(stderr). You can redirect such messages to a file of your choice by using the standard shell redirection commands (refer to the csh(1) and sh(1) man pages). A compiler diagnostic message consists of the following components:

- the compiler name, `cc`;
- the line number where the error occurred;
- the path name of the source file containing the line in error; and
- a brief description of the error.

A user error message is generated in the following format:

```
cc: error on line 1 of foo.c: array element
type has unknown size
```

If the compiler detects an internal error, it generates a message in the following format:

```
>>>>>  C O M P I L E R  E R R O R      <<<<<
>>>>> See your system manager for help <<<<<
cc : Compiler error on line num of filename.
```

The preceding lines are followed by a line that describes the nature of the error. Report such errors using the `contact` utility.

Optimization report

If a program is compiled with the `-O2` or `-O3` option, the compiler generates an optimization report for each program unit. This report consists of a loop table, an array table, or both. You can specify the tables to be included in the optimization report with the `-or` compiler option, as explained in the "Optimization options" section of this chapter.

For more details, refer to the *CONVEX C Optimization Guide* for C Series information or the *Exemplar Programming Guide* for SPP Series information.

Runtime messages

Runtime error messages in the backward-compatible mode (`-pcc`) are directed to stderr. The C Series and SPP Series compilers handle these messages somewhat differently. However, the compiler always sets `errno`, no matter what option you use when you compile.

C Series only

On C Series machines, a math-routine error message has the form:

routine_name: [*error_number*] *description*

Suppose that you compile the code below with the `-pcc` flag.

```
#include <math.h>

main()
{
  float f=(-1);
  f=(float) asin(f,2);
}
```

In backward-compatible mode, the compiler does not detect that the first argument to `asin()` is negative. It compiles the program. When you run the program, it sends the following error message to stderr:

```
mth$d_asin: [312] argument out of range for
arc sin
```

SPP Series only

On SPP Series machines, a math-routine error message has the form:

routine_name: *error_description*

Suppose that you compile the code above with the SPP Series compiler using the `-pcc` flag. The compiler does not detect that the argument to `asin()` is negative. It compiles the program. When you run the program, it sends the following error message to stderr:

```
asin: DOMAIN error
```

Mixed compatibility modes

The CONVEX C compiler supports four compatibility modes for C Series machines and three for SPP Series machines:

- extended
- conforming
- strict (C Series only)
- backward-compatible

Each of these modes is accessed using methods described in the "Compatibility modes" section on page 13. However, it is

possible to create a program that is compatible with a mixture of these modes. A two-step compilation process is necessary to obtain mixed mode programs. For example, to compile a program on a C Series machine that has strict ANSI C language features but also accesses the POSIX function library, use the following command lines:

```
cc -c -str -D_POSIX_SOURCE file.c
cc -std file.o
```

The first command line requires the file to be ANSI C compatible and to include POSIX header file information. The second command links the resulting object file with the library containing ANSI C and POSIX1003.1 functions.

Refer to Chapter 3, "Compatibility modes," for more information on the subject of mixed compatibility modes.

HP Compatibility

For the CONVEX C compiler, the following considerations apply:

- A left-shift (<<) is a logical shift. A right-shift (>>) on a C Series machine is logical. On an SPP Series system, a right-shift is arithmetic.
- Multiple character constants are detected and reported as a warning. (These character constant constructs are not portable.)
- HP requires 8 byte alignment and padding for `double` and `long long` within structures and unions. C Series machines require only 4-byte alignment for these.

Storage class extensions (SPP only)

The C compiler for SPP Series implements syntax extensions to the C language to support association of user-defined data objects to specific classes of the SPP Series memory model. The various memory classes differ in their latency and caching strategies.

You may specify only one memory class for each object:

`thread_private`—the object is private to its thread of the process.

`node_private`—the object is private to threads running on a given hypernode.

`near_shared`—the object is accessible by any thread running on any hypernode of a subcomplex, but is

physically stored entirely within the subcomplex-global memory of a particular hypernode.

`far_shared`—the object is accessible by any thread running on any hypernode.

`block_shared`—the object has a unique virtual and a unique physical address, and can be accessed by any thread running on any hypernode in the subcomplex.

Note

The `block_shared` memory class can only be assigned dynamically. Use the `memory_class_malloc` function to assign block-shared memory to an object.

Include the file /usr/include/spp_prog_model.h when using the memory extensions for SPP computers. This header file maps user symbols into the implementation reserved space.

To assign any memory class but `block_shared` to a static variable, follow the form

class_name type_specifier namelist

where

class_name is the name of one of the four memory classes above,

type_specifier is a standard C data type, and

namelist is a comma-delimited list of variables and/or arrays of *type_specifier.*

The code below declares integers `i` and `j` and integer array `k[128]` as members of the `thread_private` memory class.

```
#include <spp_prog_model>
.
.
.
thread_private int i, j, k[128];
```

This header file also provides a memory allocation function, `memory_class_malloc`, which allows you to allocate memory dynamically by class. This function has the following form:

```
void *memory_class_malloc(size_t bytes,
        int class_name);
```

where

memcls_ptr is a previously-declared pointer to a variable of the desired memory class,

bytes is the requested number of bytes, and

class_name is one of the class names below:

- `THREAD_PRIVATE_MEM`
- `NODE_PRIVATE_MEM`
- `NEAR_SHARED_MEM`
- `FAR_SHARED_MEM`
- `BLOCK_SHARED_MEM`

The code below declares a pointer of type `double`, then uses `memory_class_malloc` to allocate eight bytes of `thread_private` memory class to it.

```
thread_private double *x;
.
.
.
x = (double *) memory_class_malloc(8,
                    THREAD_PRIVATE_MEM);
```

Refer to the *Exemplar Programming Guide* for more information on SPP Series storage class extensions.

Compatibility modes

3

This chapter describes:

- Compatibility modes of the compiler
- How to specify each compatibility mode on the command line
- How to convert an application to a compatibility mode
- Differences among the compatibility modes

Modes defined

CONVEX C has four compatibility modes, as shown in Table 8.

Table 8 Compatibility modes

Mode	Option	Language	Default functions
Extended	`-ext`	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	`-std`	ANSI C	ANSI C, POSIX
Strict (C Series only)	`-str`	ANSI C	ANSI C
Backward-compatible	`-pcc`	Non-ANSI C, CONVEX	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first POSIX standard to be adopted. It represents a standard system call interface and environment based on the UNIX operating system. It supports application portability at the source-code level.

The four compatibility modes differ in two areas: language specification and library functions. For example, an application that is compiled in the strict mode uses only ANSI C language features and the only library functions that are automatically linked are those in the ANSI C library.

One language feature that can be used in the extended mode but not in the strict or conforming mode is the `asm` statement. This is a CONVEX extension that directs the compiler to insert in-line assembly-language statements into object code.

Another CONVEX extension that is not permitted in either the conforming or strict modes is the data type `long long int`. This data type is a 64-bit integer. Extended mode is the default compatibility mode of the compiler. Some portability is sacrificed when CONVEX-specific language features and library functions are used.

The major difference between the conforming mode and the strict mode is the functions that can be automatically linked into the application program: the conforming mode can link in POSIX functions while the strict mode can only link in ANSI C functions. These two modes use the same language specification because POSIX only defines an interface to system functions.

The backward-compatible mode differs considerably from the other three modes with respect to language features and functions. It does not recognize the keywords `const`, `signed`, and `volatile`. Further, it does not use ANSI C or POSIX functions. The only benefit of using this mode is that it is closely compatible with CONVEX C compilers prior to CONVEX C V4.0.

Specifying compatibility modes

Specifying a compatibility mode with the compiler is straightforward; the suitable compiler option is included on the command line. The following command-line examples show how to compile the source file `applic.c` in each of the four modes:

- Extended:

  ```
  cc applic.c or cc -ext applic.c
  ```

- Conforming:

  ```
  cc -std -D_POSIX_SOURCE applic.c
  ```

C Series only

- Strict:

  ```
  cc -str applic.c
  ```

- Backward-compatible:

  ```
  cc -pcc applic.c
  ```

In the second example, a macro constant is used.

```
_POSIX_SOURCE
```

This macro provides access to POSIX function prototypes in ANSI C header files. Applications that conform to the POSIX standard must define the _POSIX_SOURCE macro on the first line of every source file. Applications that do not need to conform to the POSIX standard can define it on the command line, as in the example. This constant is not automatically defined in the conforming mode because conforming POSIX applications *must* define it in the source code.

Another macro constant that has a similar purpose is:

_CONVEX_SOURCE

This macro provides access to the CONVEX function prototypes in the ANSI C header files.

Both _POSIX_SOURCE and _CONVEX_SOURCE are automatically defined when a source file is compiled in the default mode. The _CONVEX_SOURCE macro can be used *only* if the _POSIX_SOURCE macro is also defined.

Single-mode compilation examples

Examples contained in this section demonstrate how to compile a program for each of the four modes of the compiler. For all the examples, assume that the CCOPTIONS environment variable is null. The following command line compiles an application for the extended mode, which is the default compatibility mode:

```
cc applic.c
```

Because _CONVEX_SOURCE and _POSIX_SOURCE are defined automatically, function prototypes for CONVEX and POSIX functions are accessible in the ANSI C header files.

For CONVEX SPP Series systems, _HPUX_SOURCE is also defined.

The following command line compiles an application in the conforming mode:

```
cc -D_POSIX_SOURCE -std applic.c
```

No CONVEX extensions are used in this example; only ANSI C language features are permitted. The -std compiler option indicates that only the POSIX and ANSI C system functions are automatically linked into the executable program.

The following command line compiles an application in the strict mode (C Series) that conforms to the ANSI C specification:

```
cc -str applic.c
```

The following command line compiles a program in the backward-compatible mode of the compiler:

```
cc -pcc applic.c
```

Use of either the _POSIX_SOURCE macro or the _CONVEX_SOURCE macro in the backward-compatible mode is undefined because these macros provide access to constructs that the backward-compatible modes does not recognize. Several compiler options are incompatible with this mode.

Refer to Chapter 2, "Compiler fundamentals," for more information.

Mixed compatibility modes

Each of the ANSI C compatibility modes has specific advantages, as indicated below:

- **Extended mode**—Permits CONVEX extensions to the language. Functions are optimized for CONVEX hardware.
- **Conforming mode**—Allows POSIX functions.
- **Strict mode**—Allows increased portability between ANSI C compilers.

These compatibility modes can be combined to tailor the compilation environment of an application.

Two command lines are required to compile and link an application in a mixed compatibility mode; both command lines use the cc command. The first command line translates the application into object code, specifying the language features and providing access to information in the appropriate header files. The second command line indicates which libraries are linked into the executable program.

Three language specifications can be used:

- Strict ANSI C (-str)
- ANSI C with CONVEX extensions (-ext)
- Backward-compatible C (-pcc)

These language specifications are obtained by compiling an application with the strict, extended, and backward-compatible compatibility modes, respectively.

Similarly, there are four library systems:

- ANSI C functions
- ANSI C and POSIX functions
- ANSI C, POSIX, and CONVEX functions
- Backward-compatible CONVEX C functions

Note

SPP Series C has only one library. The include files have different modes. Refer to the *Exemplar Programming Guide* for more information.

Linking with the strict (C Series only), conforming, extended, and backward-compatible compatibility modes, respectively, provides access to these library systems. For example, you might want to develop a highly portable CONVEX program. Such a program uses functions that are specific to CONVEX hardware, but is strict in its interpretation of the language.

Use the following command lines:

```
cc -D_POSIX_SOURCE -D_CONVEX_SOURCE -str -c applic.c
cc applic.o
```

The first command line provides function prototypes to CONVEX functions, while maintaining strict interpretation of the language. The `_CONVEX_SOURCE` macro can be used only if the `_POSIX_SOURCE` macro is defined, *even if no POSIX functions are used*. The second command line specifies that the ANSI C, POSIX, and CONVEX extension libraries are used.

Mode conversion

This section lists the steps required to convert programs to the backward-compatible or the extended mode of the CONVEX C compiler.

Porting to backward-compatible mode

Three steps are required to port applications compiled with the CONVEX C V3.0 compiler or the Common C compiler to the backward-compatible mode of CONVEX C:

1. Compile under ConvexOS.
2. Use `lint` to remove errors.

3. Recompile in backward-compatible mode.

Step 1 Compile under ConvexOS.

You must compile the application under ConvexOS V8.1 or later. If the application was originally compiled using the Common C compiler, it is necessary to use the `pcc` command because CONVEX C V4.0 and later is invoked with the `cc` command line. Any problems found in this step are probably caused by incompatibility between ConvexOS and the environment in which the application was previously compiled and executed.

Step 2 Use `lint` to remove errors.

The second step removes system-dependent code and code that can produce errors. This increases the portability of the application to CONVEX C. Refer to the "lint utility" section on page 70 for more information about the lint utility. The suggested command line for this step is:

```
lint -c -h *.c -pcc
```

where

`*.c`	represents names of source files.
`-c`	detects errors that occur in casting.
`-h`	applies heuristic tests to discover errors.
`-pcc`	runs `lint` in the backward-compatible mode.

Removing system-dependent code reduces chances for an error when CONVEX C is used, or when libraries that are linked with the program are modified.

Step 3 Recompile in backward-compatible mode.

The third step exposes errors caused by differences between the compiler used in the first step and the CONVEX C compiler. The command line for this step is:

```
cc -pcc *.c
```

where

`*.c`	represents names of source files.
`-pcc`	directs the compiler to execute in backward-compatible mode.

Some changes that can be required for programs compiled with the Common C compiler are listed in the "Incompatibilities with Common C" section on page 65.

Porting to extended mode

After an application has been ported to the backward-compatible mode of CONVEX C, it can be ported to the extended mode. There are three steps:

1. Link with extended libraries.
2. Use lint to remove errors.
3. Compile in extended mode.

Step 1 Link with extended libraries.

Linking with ANSI C libraries removes the dependence on the old system libraries. The two command lines for this step are:

```
cc -pcc -c *.c
cc *.o
```

where

`*.c`	represents names of source files.
`*.o`	represents names of object files.
`-c`	prevents the object files from being linked.
`-pcc`	directs the compiler to execute in backward-compatible mode.

The first command line compiles source files with the backward-compatible mode of the compiler. The second command line links resulting object files with functions available in the extended mode of the compiler.

Failures at this stage are probably caused by differences between standard library functions and backward-compatible library functions, or by the application's dependence on undocumented behavior of old library routines.

Incompatibilities between backward-compatible mode libraries and ANSI C libraries are listed in the "Extended mode differences" section on page 58.

Step 2 Use lint to remove errors.

The second step invokes the lint utility to remove system-dependent code and code that can produce errors. The suggested command line for this step is:

```
lint -c -h *.c -ext
```

where

`*.c`	represents names of source files.
`-c`	detects errors that occur in casting.
`-h`	applies heuristic tests to discover errors.
`-ext`	runs lint in the mode that accepts CONVEX extensions to the language.

Removing system-dependent code reduces chances for an error when CONVEX C is used, or when libraries that are linked with the program are changed.

Step 3 Compile in extended mode.

The extended mode of the ANSI C compiler provides the ANSI C features, POSIX functions, and CONVEX extensions. The command line used at this step is:

```
cc *.c
```

where

`*.c`	represents names of source files.

Extended mode differences

The extended mode is the default compatibility mode of the compiler. This section discusses problems that can be encountered when porting programs from compilers that do not support the ANSI standard:

- Language features that prevent a non-ANSI C program from being compiled
- Changes in the semantics of the language
- Changes in header file organization
- Future directions in ANSI C
- Error return codes of C intrinsic functions

Most of these problems concern the translation of existing code to ANSI C. Information provided in this document can be used to assess difficulties in porting programs between systems. The incompatibilities between the Common C compiler and the backward-compatible mode of CONVEX C are discussed in the "Incompatibilities with Common C" section on page 65.

Changes that ANSI C imposes on existing applications can be grouped into two categories: changes that prevent compilation and changes that alter semantics of a construct without inhibiting compilation. Semantic changes are potentially more

disruptive because different code can be produced without warning.

Chapter 5, "CONVEX C intrinsics," addresses the problem of using C intrinsic functions.

Changes that prevent compilation

Some changes introduced by the ANSI C standard can prevent compilation.

- Five keywords have been added: `const`, `enum`, `signed`, `void`, and `volatile`. Use of these words in the wrong context will generate an error message.
- Declaring an identifier that is common between two or more compilation units must obey the following rule: only one compilation unit can contain a definition of the identifier; the remaining compilation units must declare the identifier using the `extern` storage class.
- Predefined macro names, such as `__FILE__` and `__LINE__`, cannot be redefined or undefined with the `#define` or `#undef` preprocessor directives.
- The `entry` and `asm` statements are not accepted in ANSI C. The `asm` statement is available as a CONVEX extension; `entry` is not.
- The numerals 8 and 9 are no longer available in octal constants.
- String literals cannot be modified.
- Converting a pointer of any object type (other than `void`) to a pointer of a different object type without an explicit type cast is not permitted.
- `long float` is not in the ANSI C standard. Its use generates a warning in all ANSI C modes of CONVEX C.
- Accessing a nonexistent member of a structure or union is an error.
- Empty declarations are invalid except for mutually referencing `struct` and `union` structures.
- Declaring zero-length arrays is invalid.
- No type specifiers can be added to a type that was defined using `typedef`.
- Function pointers cannot be converted to nonfunction pointers, and vice versa.

- Pointers to functions that have different parameter-type information are different types.
- Functions called with a variable number of parameters must have a function prototype.
- Formal parameters of a function cannot be `typedef` names.

Semantic changes

Below is a list of semantic changes introduced by the ANSI C standard. These semantic changes do not cause the compiler to generate error messages; they can cause a non-ANSI C program to generate incorrect output. Most of these changes are cited in the *American National Standard for Information Systems — Programming Language C* document. The changes involve general operations, expressions, declarations, constants, literals, functions, macros, operators, and operands.

Operations

- A program that depends on preserving unsigned arithmetic conversions will behave differently, probably without complaint. This semantic change is considered the most serious one resulting from the current ANSI C standard.
- Code that relies on a bottom-up parsing of aggregate initializers with partially elided braces does not yield the expected initialized object.
- Results of floating-point operations might not be the same due to differences in casting and rounding.

Expressions and declarations

- The compiler cannot reorder expressions that contain successive identical commutative or associative operators if the reordering can produce different results.
- A program relying on file scope rules for external declarations might be valid under block scope rules but behave differently when compiled in an ANSI C compatibility mode.
- The empty declaration

  ```
  struct x;
  ```

  is no longer innocuous because it can now be used to hide a definition of x that exists in an outer block.

Constants and literals

- Unsuffixed integer constants can have different types. Their type depends on the smallest integral type required to represent them.
- A constant of the form `'\\078'` is valid, but denotes a character constant whose value is the combination of the values of `'\07'` and `'8'`.
- Because the escape sequences `'\a'` and `'\x'` have been added to ANSI C, a constant of the form `'\a'` or `'\x'` might have different meaning.
- Comments in ANSI C are replaced with a blank; in Common C they are removed. If your code relies on `/**/` to paste tokens together in Common C, you must now use `##` instead. `##` is the paste operator in the ANSI C preprocessor.
- It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme might not behave the same as with a previous C compiler.
- Character sequences that are trigraphs, such as `??!`, in string constants, character constants, or header names, are replaced by the corresponding character representation of such a trigraph. Trigraphs and characters they represent are listed in Table 9.

Table 9 Trigraph representations

Trigraph symbol	Represents
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

Functions and macros

- Calculations in `#if` expressions might simulate either the translation environment or the execution environment. Consequently, a program that depends on properties of one particular environment might now give different answers.
- Functions that depend on `char` or `short` parameter types being widened to `int`, or `float` to `double`, might behave differently.
- A macro that relies on formal parameter substitution within a string literal produces different results.
- Function-like macros can be recursive, but they are not recursively expanded.

Operators and operands

- To eliminate ambiguity, the following assignment operators no longer exist:

  `=>>, =<<, =&, =^, =|, =+, =-, =*, =/, =%`

  Further, for the assignment operators of the form +=, no space is permitted between the two characters + and =.
- Expressions with `float` operands can now be computed at lower precision. Compilers prior to the ANSI C standard performed all floating-point operations in `double`.
- Shifting by a `long` count no longer converts the shifted operand to `long`.

Header file changes

The following is a list of changes in organization of header files. Differences are noted when a function or macro that is present in the CONVEX C V3.0 compiler or Common C compiler is not located in the same header file in an ANSI C mode of CONVEX C.

This section does not list differences between the backward-compatible mode of CONVEX C and the CONVEX C V3.0 compiler. These differences are noted in the "Incompatibilities with Common C" section on page 65.

The section refers to CONVEX extensions and POSIX functions. These extensions and functions are available in the appropriate compatibility modes of CONVEX C discussed in the "Modes defined" section on page 51.

Changes in organization of header files are:

`ctype.h`

The four function-like macros `isascii`, `toascii`, `_toupper`, and `_tolower` do not exist in the ANSI C standard. However, they do exist as CONVEX extensions.

`math.h`

Single-precision math functions (`sfabs`, `ssqrt`, `shypot`, `scabs`, `ssin`, `scos`, `stan`, `sasin`, `sacos`, `satan`, `satan2`, `sexp`, `slog`, `spow`, `ssinh`, `scosh`, `stanh`) are available only in the backward-compatible mode of the compiler. These math functions are available as function-like macros in the extended compatibility mode.

Macro definitions `HUGE` and `HUGEI`, while present in the backward-compatible mode, have been replaced by the macro definition `DBL_MAX` in the ANSI C modes.

`signal.h`

Macro names for the signals `SIGCLD`, `SIG_CATCH`, and `SIG_HOLD` are no longer available. The function-like macro `SIGNALS_IN_PROG` does not exist. These four macros are available only in the backward-compatible mode of CONVEX C.

`stdio.h`

Functions `fileno` and `fdopen` are POSIX functions.

`strings.h`

This header file no longer exists. All its functions are now contained in the `string.h` header file.

The functions `creat`, `close`, `lseek`, `open`, `read`, `write`, and `unlink` are available as POSIX extensions.

Signal handlers must execute `signal(` *signal*, `SIG_DFL )` before they process a signal.

Future directions

Expected changes to the C language are listed below. While some of these changes might not occur, you should be aware of them to avoid possible incompatibilities in the future. These are taken from *American National Standard for Information Systems — Programming Language C* document.

Changes concerning the C language are listed below:

- Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolete feature that is a concession to existing C compilers.
- Lowercase letters as escape sequences in character and string literals are reserved for future standardization.
- Old-style function declarations that have empty parentheses will be obsolete.
- Two parameters declared with an array type (prior to their adjustment to pointer type) cannot refer to the same or overlapping objects. The `-alias array_args` option of the compiler enforces a stronger version of this requirement. Its use reduces the necessity for the `no_recurrence` optimization directive. Refer to Chapter 2, "Compiler fundamentals," for more information on the `-alias array_args` compiler option.

Changes to library specifications that can be expected in a future standardization of ANSI C are listed below. These changes are taken from the *American National Standard for Information Systems — Programming Language C* document.

`errno.h`

Macros that begin with "`E`" and a digit or "`E`" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to the declarations in the `errno.h` header.

`ctype.h`

Function names that begin with either "`is`" or "`to`", and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to the declarations in the `ctype.h` header.

`locale.h`

Macros that begin with "`LC_`" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to definitions in the `locale.h` header.

`math.h`

Names of all existing functions declared in the `math.h` header, suffixed with "`f`" or "`l`", might be used for corresponding functions with `float` and `long double` arguments and return values.

`signal.h`

Macros that begin with either "`SIG`" and an uppercase letter or "`SIG_`" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to definitions in the `signal.h` header.

`stdio.h`

Lowercase letters might be added to the conversion specifiers in `fprintf` and `fscanf`. Other characters may be used in extensions.

`stdlib.h`

Function names that begin with "`str`" and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to declarations in the `stdlib.h` header.

`string.h`

Function names that begin with "`str`", "`mem`", or "`wcs`" and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to declarations in the `string.h` header.

Incompatibilities with Common C

This section lists the incompatibilities between the backward-compatible mode of CONVEX C and the Common C compiler.

The incompatibilities are divided into two sections:

- Language definition
- Command line

Most of the incompatibilities between the two compilers result from illegal code that is accepted by the Common C compiler. CONVEX C does not accept illegal C code in any compatibility mode.

Language definition

Below are some incompatibilities between the language definitions of CONVEX C and the Common C compiler.

Type specifiers

The Common C compiler permits type specifiers to modify a data type created with `typedef`. For example:

```
typedef int my_int;
typedef unsigned my_int u_my_int;
```

This is illegal C code; such specifiers are not allowed in any compatibility mode of CONVEX C.

`static` and `extern`

Mixing `static` and `extern` in a declaration is undefined in the C language; such declarations are not portable between compilers.

Multiple initializers

The Common C compiler erroneously permits simple variables to have multiple initializers. For example:

```
{
    extern int init1(), init2(), init3();
    int x = { init1(), init2(), init3() };
}
```

When compiled with Common C, three functions are executed, with the last one assigning its return value to `x`. Multiple initializations are not permitted with the CONVEX C compiler in any of its compatibility modes.

Casts

The Common C compiler permits objects on the left side of an assignment operation to be cast. For example:

```
{
    int x;
    (int)x = 10;
}
```

This is allowed in Common C only when the cast is the same type as the variable or is a pointer to the same type as the variable. This is an error in the Common C compiler. CONVEX C detects this error.

Order of evaluation

The order of expression evaluation used by the two compilers is not the same. This impacts numerical computations and may change the order in which side effects occur. The changes can cause a program that produced correct results with the Common C compiler to produce incorrect results with the CONVEX C compiler. The program is not a valid C program because it depends on the order of evaluation in ways not specified by the language. For example, given:

```
#include <stdio.h>


int a[2] = {0, 1};
int b[2] = {3, 4};

main()
{
   int i = 0;
   a[i++] = b[i]; /* order of eval.
                     undefined */
   printf("a[0] = %d\n", a[0] );
}
```

If compiled with `/bin/pcc file.c`, this program displays `a[0] = 3`, but if it is compiled with `cc -pcc file.c`, it displays `a[0] = 4`. The `lint` program can detect this type of error. Refer to Chapter 4 for a brief introduction to the `lint` program.

Uninitialized variables

Applications that depend on uninitialized variables, dangling pointers, and other poor programming practices might not produce the same results on both compilers.

Negative bit shifts

The Common C compiler allows bit shifts to use negative operands. For example, a right shift with a negative right operand is equivalent to a left shift with a positive right operand. Negative operands of bit shift operators generate errors in the CONVEX C compiler if the shift length is a constant.

`switch` statements with pointers

The Common C compiler allows the expression in a `switch` statement to have a pointer value. The CONVEX C compiler reports an error if the expression does not have an integral type.

Undefined functions

The Common C compiler silently converts functions to the `extern` storage class if they are declared `static` and used in a compilation unit but not defined. The CONVEX C compiler produces a warning when it performs this conversion.

Functions returning `short int` or `char`

Undeclared functions that return a `short int` or `char` have their return value automatically converted to an `int` in the calling routine when compiled by the Common C compiler; this matches the default declaration of a function. CONVEX C does not perform this automatic conversion. Programs that depend on this

behavior produce erroneous results. However, `lint` is capable of finding these errors.

Command line differences

CONVEX C and the Common C compiler have two command-line differences:

- CONVEX C silently ignores the `-t` compiler option provided by the Common C compiler.
- The semantics of the `-B` compiler option are slightly different between the two compilers.

Development tools

4

This chapter describes some tools that assist in program development, utility programs you can use to find errors in a program, and tools you can use to increase the performance of a program.

Some of the programs discussed are optional products. If you are unsure whether a program exists on your system, ask your system manager.

Utilities, which make the development process easier, include:

- lint—Checks for errors the compiler does not detect.
- make—Makes program compilation easier and eliminates redundant compilations.
- indent—Formats C source files for easier reading (C Series only).
- error—Inserts error messages into source files (C Series only).

Debugging utilities include the following:

- CXdb—a window-oriented symbolic debugger
- cref—a cross-reference generator
- adb—an assembly-language debugger
- CXmetrics—a tool to analyze program complexity (C Series only).

Profilers analyze programs to detect code that uses the greatest time. CONVEX provides the Consultant package which includes three profilers:

- prof
- bprof
- gprof

It also provides two standalone profilers:

- CXpa
- CXtrace

Program development utilities

The utilities described in this section help you in creating and managing C programs.

lint utility

The lint utility is a program that checks C source files for code that can cause bugs or reduce the portability of a program. It also performs more rigorous type-checking than most C compilers. Types of errors that can be detected using lint include unreachable statements, loops not entered at the top, and automatic variables declared and not used. Consistency in the use of functions is also checked to find functions that return values in some places and not in others, and functions called with varying numbers of parameters. Similarly, each use of an object is compared with its declaration for consistency.

Although the ANSI C standard introduces more stringent type checking, it does not perform type checking across compilation units. In contrast, lint performs type checking across compilation units.

For example, lint detects the error in the following code, but the C compiler does not:

```
/* file: one.c */
#include <stdio.h>
extern short int x;
int main()
{
   (void) printf("val = %d\n", x );
   return(0);
}

/* file: two.c */
long int x = 3;

lint one.c two.c
x value declared inconsistently two.c(2)::one.c(3)
```

Note that the above code is in two separate files. The problem is that in file `one.c`, `x` is declared to be a `short int` whereas in file `two.c`, `x` is declared to be a `long int`.

lint also detects unreachable ("dead") code. For example, the following code contains a statement that is never accessed when the program is executed:

```
#include <stdio.h>

int main()
{
    goto skip;
    (void) printf("this is never printed");
    skip:return(0);
}
```

If lint is used, the `printf` statement in the preceding example is recognized as unreachable. Dead code that lint discovers can be the result of a logic error in the program design.

Compatibility modes

The same modes that apply to the compiler are used by lint:

`-ext`

Run lint in a mode compatible with cc's default compatibility mode.

`-pcc`

Run lint in a mode compatible with cc's backward-compatible mode.

`-std`

Run lint in a mode compatible with cc's conforming mode.

C Series only

`-str`

Run lint in a mode compatible with cc's strict mode.

Preprocessor control options

The lint utility predefines the symbol `__lint`. In backward compatible mode, the symbol `lint` is also predefined. The options available with the lint utility are:

`-D`*name*

Defines the macro *name* with a default value of 1.

`-D`*name=def*

Defines the macro *name* as if by the `#define` preprocessor directive.

`-I`*dir*

Names an alternate directory to search for include files. The compiler searches alternate directories in the order specified on the command line.

`-I-`

Inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include`. There is no way to override the effect of `-I-`.

`-U`*name*

Removes any initial preprocessor definition of *name*. Predefined identifiers of CONVEX C are described in the "Predefined symbols" section on page 37.

Control options

`-a`

Report assignments of long values to shorter variables.

`-altlint` *pathname*

Use substitute lint specified as *pathname*. (You must specify the location of the lint components using `-B` as well.)

`-b`

Report `break` statements that cannot be reached. (This is not the default because most `lex` and many `yacc` outputs produce dozens of such statements.)

`-B`*dir*

Use substitute lint components in directory *dir*.

`-c`

Complain about casts that have questionable portability.

`-C` *mylib*

Create a lint library `llib-l`*mylib.ln*.

`-d` *name*`[=(w|e)]`

This option provides improved control over lint's error messages. Refer to the "Compiler diagnostic options" section on page 209 for more information on how to use this option.

C2,3,4 Series only

`-fi`

Use IEEE floating-point mode. (Requires IEEE floating-point hardware.)

`-float dp_const`

Treat unsuffixed floating-point constants as double precision. This is the default.

`-float dp_ops`

Convert `float` operands to double-precision and use 64-bit arithmetic. This is the default in `-pcc` mode.

`-float sp_const`

Use 32-bit arithmetic with `float` operands. This is the default, except in `-pcc` mode.

`-float sp_ops`

Treat unsuffixed floating-point constants as single precision. (Some loss of accuracy may result.)

C Series only

`-fn`

Use native floating-point mode.

`-h`

Apply heuristic tests to try to find bugs and improve style.

`-n`

Do not check compatibility with the standard library.

`-sso`

Treat the `sizeof` operator's result as a signed integer. (Available only in `-pcc` mode.)

C Series only

`-tl` *t*

Set time limit to *t* minutes.

`-u`

Ignore functions and variables used and undefined, or defined and unused. This setting is suitable for running lint on a subset of files of a larger program.

`-v`

Suppress complaints about unused arguments in functions.

`-vn`

Show version number.

`-x`

Report variables that external declarations refer to, but never use.

`-z`

Do not complain about structures that are never defined.

Source level control

You can include directives in the source code to control lint's output.

`/*NOTREACHED*/`

At appropriate points, suppresses comments about unreachable code.

`/*VARARGS`*n*`*/`

Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

`/*ARGSUSED*/`

Turns on the `-v` option for the next function.

`/*LINTLIBRARY*/`

Used at the beginning of a file; shuts off complaints about unused functions in the file.

In the example below, the `/*VARARGS2*/` directive tells lint to check only the first two arguments in the following function declaration.

```
/*VARARGS2*/
printf("%d\t%d\t%d\t%d\n", v1, v2, v3, v4);
```

For more information on this utility, refer to the online lint(1) man page.

make utility

You should use make when you have several files that compose a program and you do not need to recompile all of them. The make utility uses the time and date stamps of source and object files to decide whether a file must be compiled: it compiles only

- those files that have been modified since you last compiled the program and
- any files that depend on the changed files.

Thus, if a program depends on twelve separate compilation units but only two of them have been modified, make compiles only those two before the entire program is linked.

Consider the sample make file shown in Figure 8.

Figure 8 Sample make file

```
#                                                    1
#Make file for a short example.                      2
#                                                    3
myprog: myprog.o second.o                    #       4
   cc myprog.o second.o -o myprog            #       5
                                             #       6
myprog.o: myprog.c local.h                   #       7
   cc -c myprog.c                            #       8
                                             #       9
second.o: second.c local.h pivot.h           #       10
   cc -c second.c                            #       11
```

If this file is stored in a file named `makefile`, the executable program `myprog` can be created by entering the command **make**.

The first three lines contain comments; comment lines start with a "#" symbol. Line 7 states that `myprog.o` is dependent on the two files `myprog.c` and `local.h`. If one or both of these are modified after `myprog.o` is created, the command on the line 8 is executed. This line creates an up-to-date version of `myprog.o`. The remaining lines follow the same format. Consequently, if `pivot.h` is changed, commands on line 11 and line 5 are executed to create an executable file, `myprog`.

This example, while useful enough for small programs, uses only a few of the features that the make utility provides. You can find information on the additional features in the make(1) man page. The `-k` option of the C compiler generates file dependencies that can be used in a makefile.

C Series only

indent utility

The indent utility formats C source files. Numerous options specify various printing styles, such as placement of comments and location of curly braces { } after `if` statements. This utility is useful because it converts different programming styles into one more consistent style, which can make programs easier to read.

The current version of indent does not recognize ANSI C keywords.

The following code contained in the file `indentin.c` is poorly-formatted.

```
#include <stdio.h>
int main(){int
j,i;j=5;i=6;printf("j=%d,i=%d",j,i);return(i*j);}
```

This small program is difficult to read: it has no white space to help identify the hierarchy of the program. The following command line helps to untangle this program:

```
indent indentin.c indentout.c -bad -bc -di8
```

where

`-bad`

inserts a blank line after every block of declarations.

`-bc`

inserts a newline character after each comma in a declaration.

`-di`*n*

specifies the indentation, in *n* character positions, from a declaration keyword to its following identifier.

The contents of the file `indentout.c` are as follows:

```
#include <stdio.h>
int
main()
{
        int   j,
              i;
        j = 5;
        i = 6;
        printf("j=%d,i=%d", j, i);
        return (i * j);
}
```

Thus, you can convert poorly-indented code to a readable form.

The indent utility provides many more options that you can use to format a program. For example, the `-troff` option formats the input file for the troff(1) utility. Consult the indent(1) man page for more information on the indent utility.

C Series only

error utility

The error utility analyzes and optionally inserts the diagnostic error messages compilers and language processors produce, such as cc and lint, into the source file at the line where the error

occurred. It replaces the painful, traditional methods of scribbling abbreviations of errors on paper, and permits you to view error messages and source code simultaneously without manipulating multiple windows in a screen editor.

error looks at the error messages, either from the specified file name or from the standard input, and attempts to determine the following:

- Language processor that produced each error message
- Source file and line number to which the error message refers
- Whether to ignore the error message

After it has read all input, error then inserts the error message, which might be slightly modified, into the source file as a comment on the line preceding the line to which the error message refers.

If error produces error messages that it cannot categorize by language processor or content, it sends those messages to the standard output. You can use several options with this utility:

`-q`

Confirm any potentially dangerous action (such as modifying a file) or verbose action.

`-t` *suffixlist*

Do not touch files whose suffixes appear in the suffix list. The suffix list is separated by dots, and "*" wildcards are accepted.

`-v`

After inserting error messages into all the files, initiate the vi editor to edit the first file containing an error.

`-T`

Produce terse output.

The following csh command line checks the syntax of a C source file, inserts any error messages into the file, and places the file in the vi editor if errors are detected.

```
cc -sc file.c |& error -v
```

The equivalent command for the Bourne shell (`sh`) is as follows:

```
cc -sc file.c 2| error -v
```

Debugging utilities

The following section describes the utility programs you can use to analyze a program.

CXdb debugger

CONVEX CXdb, the visual debugger, is an optional debugger that has all the debugging functions found in traditional debuggers. To generate the information necessary for using CXdb, you must compile your program with the `-cxdb` option on the command line.

Like the debugger provided with the CONVEX Consultant, CXdb can perform these functions:

- Debug program source code or disassembled code
- Debug programs containing multiple source modules
- Access program variables by name

In addition to these capabilities, CXdb performs these functions:

- Provide debugging contexts for source code and disassembled code in a windowing environment
- Attach CXdb to a running process
- Execute debugger commands while your process is running
- Create aliases and macros to simplify debugger commands
- Debug optimized code

CXdb's windowing environment supports line-oriented terminals and workstations capable of displaying CX/Motif. This windowing environment eases the task of debugging programs that contain multiple threads of execution. Refer to *CONVEX CXdb Concepts* and the *CONVEX CXdb User's Guide* for additional information on CONVEX CXdb.

Cross-reference generator

The cross-reference generator (cref) produces a table of references to each object in a C source program. These objects include all user-defined C structures and variables. The format of the table is controlled by options specified on the cref command line.

Consider the following C source program, `crefex.c`:

```
1  #define loop_incr 200
2  #define array_size 1000
3
4  extern void sub1(int);
5  float a[array_size];
6
7  struct some_struct {
8    int a;
9    int b;
10 };
11
12 struct some_struct var_struct = {
13   10,
14   20
15 };
16
17 void main()
18 {
19   int i;
20
21   for( i=1; i<=array_size; i+=loop_incr )
22     sub1(a[i]);
23   var_struct.a = loop_incr;
24   return(0);
25 }
```

The cross-reference generator produces a listing in the format shown in Figure 9 when the following command is executed:

```
cref crefex.c
```

Figure 9 Sample cross-reference listing

```
% cref crefex.c

a             5#     8#    22  23
array_size    2#     5     21
b             9#
i            19#    21     22
loop_incr     1#    21     23
main         17#
some_struct   7#    12#
sub1          4#    22
var_struct   12#    23
Symbols = 9       Hash table size = 20      Density = 0.450000
```

The first column of the output contains the name of the object. The numbers following each object state where that object is

used. Pound signs (#) indicate lines on which an object is modified.

The cref output in the previous example illustrates some limitations of the cref utility that you should be aware of.

- First, the object `some_struct` is shown by cref to be defined twice, on line 7 and on line 12. Actually, the object is defined on line 7, but creates a variable on line 12. However, the cref utility interprets both uses of object `some_struct` to be a definition of that object.
- Second, consider the a object. cref indicates that this object is also defined twice. However, the source code contains two objects named a . One is an array; the other is a member of a structure. Consequently, to avoid confusion, do not use the same name for several objects.

cref is useful as long as its limitations are recognized. If you execute cref on a program that is syntactically incorrect, results are unpredictable. For a complete description of the cross-reference generator, refer to the cref(1) man page.

Assembly-language debugger

The assembly-language debugger, adb, is an object-code debugger that requires no recompilation or special compiler options. The CONVEX adb debugger allows you to examine core dumps from failed programs and to interactively debug programs at the assembly-language level.

Because programs are run under adb, it is always aware of the state of the program and values of all variables. Using adb, you can perform the following functions:

- Display the assembly-language instructions of the program
- Stop program execution at any point
- Examine values of program variables
- Modify the value of any program variable
- Execute a program one instruction at a time
- Display values of machine registers
- Modify values of machine registers

You can use the adb debugger to debug programs at all optimization levels, including vector code and programs running on multiple processors. For a detailed description of adb and complete instructions about its use, refer to the *CONVEX adb Debugger User's Guide.*

C Series only

CXmetrics

The CXmetrics tool, `metrics`, provides analytical data about the relative complexity of C and FORTRAN programs. This data can help you develop, test, revise, and maintain your source code by reducing software complexity and improving quality at each phase. To use CXmetrics, you must compile your code with the `-metrics` option.

CXmetrics measures four different types of metrics:

- **Size metrics**—CXmetrics measures the number of lines in the source code.
- **Control-flow metrics**—CXmetrics measures the number of paths, usually defined in terms of the number of nodes and edges in a directed graph of the program control flow.
- **Cross-reference metrics**—CXmetrics indicates the number of references to external functions and global variables; lists specific details of those cross-references.
- **Documentation metrics**—CXmetrics counts the number of comments or comment lines in the code.

To use CXmetrics, you should follow three steps:

1. Compile your source code with the `-metrics` option.
2. Invoke CXmetrics from the command line.
3. View the output reports generated by CXmetrics.

You can generate and view CXmetrics reports in two ways:

- With the shell command-line interface
- With CXmetrics' graphical user interface (GUI)

The command-line interface lets you view only the basic CXmetrics reports. The GUI interface allows you to view the basic reports plus bar charts (histograms) of the metrics data and a hierarchical graph of your program's calling structure. For more information about these interfaces, see the *CONVEX CXmetrics User's Guide.*

Profilers

Profilers allow you to debug and analyze the performance of your programs. Before using a profiler, you need to use one of the compiler options in Table 10 to tell the compiler to add *instrumentation*, or additional code for a profiler to read. The profiling tools described in the next sections use this instrumentation in different ways.

CONVEX provides separate packages for CONVEX C Series and SPP Series systems.

CONVEX Consultant profilers

The CONVEX Consultant package is an optional product that includes a package of routines you can use to debug and analyze the performance of C or Fortran programs. This package offers three profilers that allow you to monitor the performance of your program: prof, bprof, and gprof.

You can use the information obtained from a profiler to improve the efficiency and speed of a program. To use a specific profiler, you must first compile your program using one of the options described in Table 10.

Table 10 Compiler options for profiling

Compiler option	Description
`-p`	Produces code that counts the number of times each utility is called. If the program completes normally, a profile file (`mon.out`) is produced. This file can be processed by the prof profiler to generate an execution profile. When you specify the `-p` option, the linker searches profiling libraries instead of the standard libraries.
`-pb` (C Series only)	Produces code that counts the number of times each statement is executed. If the program completes normally, a basic block profile file (`bmon.out`) is produced. This file can be processed by the bprof profiler to display the source-level execution counts.
`-pg`	Produces instrumentation similar to the `-p` option, and invokes a runtime recording mechanism that keeps more extensive statistics. If the program completes normally, a call graph profile file (`gmon.out`) is created. This file can then be processed by the gprof profiler to produce a comprehensive execution profile.

For further information about these optional profilers, refer to the *CONVEX Consultant User's Guide*.

CXpa profiler

The CONVEX C compiler supports the CXpa profiler on CONVEX C Series (C200 and greater) and SPP Series machines. CXpa is an optional product.

CXpa provides a more accurate profiler than prof and bprof. This performance analyzer is described in detail in the *CONVEX CXpa User's Guide.*

CXpa is an interactive tool that can monitor program activity at the routine level, the loop level, or the block level:

- Routine-level profiling produces summary information about routines that are called during profiled execution of the program. This information includes:
  - Number of times the routine is called
  - Wall-clock time spent in the routine and percentage of program total wall-clock time
  - CPU time spent in the routine and percentage of program total CPU time
  - Net CPU time spent in the routine and percentage of program net CPU time
- Loop-level profiling produces summary information about individual loops in the program. Certain loop optimizations affect the way a loop is profiled. For example, if loop distribution, partial vectorization, or dynamic selection has been performed, CXpa profiles each replicated copy of the loop separately. Information provided for a loop includes:
  - Type of loop (scalar, vectorized, or parallelized)
  - Number of times the loop is executed and the vector length
  - Total CPU time in the loop
- Block-level profiling shows how many times each basic block in your code is executed. A basic block is a set of sequential assembly-language statements, the last of which changes the flow of control.

Table 11 shows events that CXpa can capture for different platforms. For more information about these events, see the *CONVEX CXpa Reference.*

Table 11 Events that CXpa captures for different platforms.

C2 and C3	C4	SPP Series
Wall Clock Time	Data Cache Misses	Wall Clock Time
Call Graph	Data Cache Accesses	Call Graph
	Instruction Cache Misses	Event Accesses
	Page Table Misses	Memory Events
	Floating Point Operations	CTIcache Events
	Vector Loads/Stores	

CXtrace profiler

The CONVEX trace-based tool, CXtrace, operates in a parallel system environment. This tool is also optional.

CXtrace's main software components—an instrumentator, a runtime performance, and a set of analysis tools—measure and display your program's performance.

- The source-code instrumentator, `xinstrument`, inserts performance monitoring routines into the application's source code. You select files and constructs that should be instrumented.
- The runtime performance monitoring library, or *monitor*, provides a set of monitoring routines that measure and record various aspects of program performance, such as message-passing overhead, synchronization overhead, and time spent in different subroutines.
- Two tools process and display the execution data.
  - Trace view (`tv`) provides a static view of the entire trace file that you can scroll and zoom both horizontally and vertically.
  - `tally` provides performance statistics for the entire program. These statistics provide insights into the general behavior of the program. You can also use the data from `tally` as input into statistical drawing packages.

CONVEX C intrinsics

5

This chapter defines intrinsic functions and intrinsic instructions and identifies behavior they can introduce.

This chapter discusses the following:

- Intrinsic instructions and functions
- Intrinsic function behavior
- The procedures for disabling intrinsics

What are intrinsics?

In CONVEX C, there are two types of intrinsics: intrinsic instructions and intrinsic functions.

Intrinsic instructions are commands in a CONVEX instruction set. For example, an instruction in a CONVEX C100 Series architecture is `add`; an instruction in a CONVEX C200 Series architecture is `sqrt`. While the `add` instruction is in the instruction set for a C200, the `sqrt` instruction is not implemented in the C100.

Intrinsic functions are C-callable functions that may use intrinsic instructions. Intrinsic functions are used by default in the extended compatibility mode of the compiler. You can obtain access to these functions in the strict and standard compatibility modes by including `-U__NO_INLINE_MATH` on the `cc` command line. If you want to use intrinsic functions in the backward-compatible mode, include `-D__INLINE_MATH` on the `cc` command line or, on C Series systems, include the `fastmath.h` include file in the source code.

Intrinsic functions are useful because they:

- Require less function overhead
- Execute faster than non-intrinsic functions
- Do not inhibit vectorization of loops
- May be folded at compile time

The disadvantages of intrinsic functions are:

- Math intrinsic functions do not modify `errno` when an error occurs.
- The compiler does not recognize a dependency between `errno` and the math intrinsic functions.

The following example is a C program that calls `sqrt`, an ANSI C function:

```
#include <math.h>
#include <stdio.h>
int main()
{
   double x = 4.0;
   double y;
   y = sqrt(x);
   void) printf("%lf\n", y );
   return(0);
}
```

If this program is compiled and executed on a C1 machine or compiled with the `-tm C1` command line option, the compiler uses an intrinsic `sqrt` function with the program: since the C1 instruction set does not have a `sqrt` instruction, the compiler must implement the `sqrt` function using a software square-root algorithm. But if it is compiled and executed on a C2 or SPP Series machine, or compiled with the `-tm C2` or `-tm c2` or `-tm spp1` command line options, the compiler uses the intrinsic `sqrt` instruction.

You can see which functions are implemented as intrinsics by searching the include files for the `__NO_INLINE` macro. Intrinsic functions are implemented as function-like macros defined with CONVEX reserved function names.

For example, the `math.h` include file contains the following code fragments:

```
#if !defined(__NO_INLINE) &&
    !defined(__NO_INLINE_MATH)
/* fast implementations of the routines
defined by ANSI C */
#define acos(x) _mth$d_acos((double)(x))
#define asin(x) _mth$d_asin((double)(x))
      .
      .
      .
#endif
```

In this example, the acos function is a function-like macro that calls _mth$d_acos.

Note

Do not call functions that define intrinsic functions directly in your programs. These function names are subject to change.

Intrinsic function behavior

The following section describes problems associated with intrinsic functions.

Generation of signals

Intrinsic functions may not modify the errno variable when an error occurs. On C Series machines, when an intrinsic instruction detects an error it does not generate a signal because when a C program begins execution, the bit in the program status word register that controls the generation of intrinsic error signals is set to 0.

For example, when the following program is compiled in the extended compatibility mode and executed on a C2 machine, it prints an incorrect result:

```
#include <math.h>
#include <stdio.h>
#include <errno.h>
int main()
{
   double x = -4.0;
   double y;

   errno = 0;
   y = sqrt(x);
   if( errno == EDOM )
      (void) printf("domain error\n");
   else
      (void) printf("%lf\n", y );
   return(0);
}
```

When this program is compiled for a C1 machine using the -tm c1 command line option and executed, it catches the domain error because the implementation of the sqrt intrinsic function on a C1 modifies the errno variable when an error occurs. However, on a C2 or later C Series system, or on an SPP Series machine, the sqrt intrinsic function *does not* modify errno and errno will not be set.

On SPP Series systems, if the compiler determines that the arguments to an intrinsic function have a constant value, the value of the intrinsic function will be computed at compile time. If the intrinsic function is `sqrt` or `abs`, the hardware instruction will be used. In all other cases, the call will be made to the normal library routine and `errno` will be set as usual.

`errno` and optimization

Another problem associated with using intrinsic functions is that at levels of optimization higher than `-no`, the compiler removes redundant code or replaces code with faster pieces of code. This feature can cause problems with some intrinsic functions.

For example, the compiler removes the conditional statement in the following code because it does not realize that the `acos` function can modify `errno`:

```
errno = 0;
a = acos(x);
if(errno == EDOM) {
   .
   .
   .
}
```

At optimization level `-O2`, the compiler optimizes this code to the following:

```
a = acos(x);
```

How to disable intrinsics

In the strict and standard compatibility modes, the compiler uses all intrinsic functions except for math functions by default. (ANSI C programs expect `errno` to be modified when certain function errors occur, but the math intrinsics do not do this.) The compiler also does not use math intrinsic functions by default in the `-pcc` mode, to maintain backward compatibility with previous compilers.

There are two ways to prevent the compiler from using intrinsic functions in your program. The first way is to define the `__NO_INLINE` macro on your command line using the `-D` command line option. This prevents the compiler from accessing all intrinsic functions. However, this option might be too stringent: there are several different types of intrinsic functions,

enumerated in the following list, and you may want to disable only one type.

- `__NO_INLINE_BINT`
- `__NO_INLINE_CTYPE` **(C Series only)**
- `__NO_INLINE_MATH`
- `__NO_INLINE_SIGNAL` **(C Series only)**
- `__NO_INLINE_STDIO` **(C Series only)**
- `__NO_INLINE_STDLIB` **(C Series only)**
- `__NO_INLINE_STRING` **(C Series only)**
- `__NO_INLINE_TIME` **(C Series only)**

Each of these macros is named after the include file in which it can be found.

For example, the `stdio.h` include file, which contains some I/O functions, declares function-like macros defined with intrinsic functions. You can disable these intrinsic functions by including the `-D__NO_INLINE_STDIO` option on the command line. You might want to link your program with the intrinsic functions when you have completely debugged it.

The second way to avoid the `errno` problem on C Series machines is to set the intrinsic error enable bit of the processor status word when your program is started. You must include a signal handler that determines what caused the signal and then take appropriate actions. Further detail of this approach is beyond the scope of this chapter. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 3, "General registers," for more information on the processor status word and the bits associated with intrinsic instruction errors on C Series machines.

Input and output

6

This chapter describes methods that a program uses to receive input and generate output. It first explains basic concepts of input and output functions, then presents an example for each. The chapter briefly defines functions used for input and output; refer to the man page for each function for additional information.

File input and output concepts

This section of the chapter defines concepts used in file input and output (I/O). These concepts are generalized later to include program I/O.

File manipulation

You access a file in three phases:

1. Opening access to the file
2. Performing input and/or output on the file
3. Closing access to the file

When you open a file, the compiler creates a data structure of type `FILE` to transfer data to or from a file. Many routines use this data structure to transfer data between files and a program. After you have processed the file, you must close the file to flush all buffers containing data for that file.

Caution

If you do not close a file after processing it, you can lose data.

You can open a file with the `fopen` function. This function returns a pointer to a `FILE` data structure, as in the example below.

```
#include <stdio.h>

main()
{
   FILE *fp;
   fp = fopen("some_file", "r" );
   .
   .
   .
}
```

This statement opens the file `some_file` to read data. The include file, `stdio.h`, contains standard input and output function prototypes. The `FILE` data type declares a file pointer that other I/O routines use.

After opening the file, you can write text to it using the `fprintf` function (and many other functions).

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

main()
{
   FILE *fp = fopen( "output.data", "w" );
   if( fp == NULL ){
      (void) fprintf( stderr,
      "Can't open file: output.data\n" );
      exit( EXIT_FAILURE );
   }
   (void) fprintf( fp, "This is text.\n" );
   (void) fclose( fp );
}
```

The data structure pointed to by `fp` contains status information about the file, such as whether an error occurred when the file was last accessed. All information contained in the data structure is accessible by library functions. One such function is `ferror`, as shown in the following example.

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

main()
{
   FILE *fp = fopen( "output.data", "w" );
   if( fp == NULL ){
     (void) fprintf( stderr,"Can't open file:"
                    "output.data\n");
     exit( EXIT_FAILURE );
   }
   (void) fprintf( fp,"This is some text.\n");
   if( ferror( fp ) ){
     (void) fprintf( stderr,"Error writing to"
                    "file.\n" );
     (void) fclose( fp );
     exit( EXIT_FAILURE );
   }
   (void) fclose( fp );
}
```

The `ferror` function checks for an error when the file associated with structure `fp` was accessed. The error condition exists until you clear it with the `clearerr` function. You can also use many other functions to manipulate files.

Although `FILE` is defined in the `stdio.h` header file, its contents must be accessed only with standard functions, not as a `struct`.

Finally, the `fclose` function closes access to and flushes the buffers of a file.

File types and access modes

ANSI C defines two file types: binary and text. On a CONVEX computer system and all POSIX-compliant systems, these two file types are identical. You do not need to be concerned with these two types unless you intend to port the program to another computer system. If that is the case, use the binary or text types as required by the other computer system.

You define the access mode for a file when you open the file. The three basic modes of operation for files are reading, writing, and appending.

- Reading examines the contents of a file without modifying them.
- Writing modifies the contents of a file.

- Appending data to a file adds data at the end of a file without changing the original data.

You may create other modes by combining these basic modes. For example, the read/write mode permits data in a file to be read as well as written. See the fopen(3) man page for the method used to obtain each of the access modes.

A program can access files only if the program has access permission for the file. For example, if the file only has read-only permission, an error occurs if the program tries to open that file for writing, appending, or reading and writing. For more information on file permissions, refer to the chmod(1) man page.

System functions and stream functions

CONVEX C includes two groups of functions: system I/O functions and stream I/O functions. The stream I/O functions use the system I/O functions to perform more complicated tasks.

System I/O functions provide simple, basic file I/O functions. They are:

- `close`
- `creat`
- `lseek`
- `open`
- `read`
- `write`

Stream I/O functions are much more numerous. The stream I/O functions are described in the "stdio.h" section on page 124. Some of these functions can manipulate the size of a file buffer to enhance performance, while others can obtain specific pieces of data in a file. For example, the `fscanf` function can extract a number from the middle of a line of text.

Thus, the group of I/O functions used in a program depends on the sophistication required to manipulate data files. Also, system I/O functions limit portability of a program because they are not available in the ANSI C standard.

Program input and output

You can perform program I/O on files as well as devices. Every C program has three files that are automatically available: `stdin`, `stdout`, and `stderr`. These files are used for standard input (keyboard), standard output (display), and standard error output (display), respectively. Some device names can be found in the /dev directory; the devices that begin with the `tty` prefix are terminals.

Access the three files, `stdin`, `stdout`, and `stderr`, using stream functions only. For example, to print a sentence on a display, use the following code:

```
#include <stdio.h>
.
.
.
(void) fprintf( stdout,"Hello, world.\n" );
```

You can access many other devices from a program. Names of these devices are contained in the directory /dev on ConvexOS. Refer to the *ConvexOS Primer* for more information on these devices.

ConvexOS allows you to redirect input and output between files and C programs. The line below shows how to redirect an input file to the program `myprog`.

```
% myprog < myprog.in
```

This command forces all input for the executable file `myprog` to come from `myprog.in`. In this case, the device `stdin` is associated with the file `myprog.in` instead of the keyboard. Similar associations occur when you redirect files by piping, which permits the output of one program to be used as the input of another program. Refer to *ConvexOS Man Pages for Users* for more information on this technique.

`freopen` reassigns I/O from a device to a file, as in the code fragment below.

```
#include <stdio.h>
.
.
.
(void) freopen( "stderr.file","w", stderr );
```

This code stores all succeeding output to the device `stderr` in the file `stderr.file`.

Program input and output example

Figure 10 shows the use of stream I/O functions. It includes a short description of the program followed by the program source code. A line-by-line description of the program follows the example. Line numbers in the program are provided for reference only; they are not part of the source code.

This program shows one way to transfer data structures between a file and a program. It reads some information from a file into an array of structures, then appends it to the file in reverse order. Key routines that make the I/O of data structures possible are `fread` and `fwrite`.

Figure 10 Sample stream I/O usage

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #define NUM_PHONES 5
4   int main()
5   {
6    FILE *fp;
7    int i,j;
8    struct {
9      char name[8];
10     char phone[7];
11   }
12   phone_array[NUM_PHONES];
13
14   if( (fp = fopen( "phones", "a+")) == NULL) {
15    ·(void) fprintf(stderr,"Can't open file: phones\n ");
16     exit( EXIT_FAILURE );
17   }
18   if( fseek( fp, 0L, SEEK_SET ) != 0 ) {
19     (void) fprintf( stderr, "Error seeking file origin\n" );
20     exit( EXIT_FAILURE );
21   }
22   j = fread( (void *)phone_array, sizeof( phone_array ), 1, fp );
23
24   if( j != 1 ) {
25     (void) fprintf( stderr, "Error reading phone array\n" );
26     exit( EXIT_FAILURE );
27   }
28
29  .if( fseek( fp, 0L, SEEK_END ) != 0) {
30     (void) fprintf( stderr, "error seeking end of file\n" );
31     exit( EXIT_FAILURE );
32   }
33
34   for( j=0, i=NUM_PHONES-1; i>=0; i— )
35     j+=fwrite((void *)&phone_array[i],sizeof(phone_array[i]),1,fp);
36
37   if( j != NUM_PHONES ) {
38     (void) fprintf( stderr, "Error writing phone array\n");
39     exit( EXIT_FAILURE );
40   }
41   (void) fclose( fp );
42   return(0);
43  }
```

The descriptions below refer to the line numbers in Figure 10.

1

The first line includes necessary functions and data structures for input and output. The `stdlib.h` header file includes the function prototype for the `exit` function.

2

The `stdio.h` header file includes the definition of the `FILE` structure.

3

Declares macro constant `NUM_PHONES`.

4

Starts the definition of the `main` function.

6

`fp` is a data structure that retains information on a file.

7

Declares two integers `i` and `j`.

8-11

These lines define the data structure that associates a phone number with a name.

12

`phone_array` is the array of data structures to use for input and output in this program.

14

This line tries to open the file that contains names and phone numbers. The access mode is `a+` because the program initially reads, then appends data to the file. This access mode does not allow the program to overwrite any data. If `fopen` cannot open the file, perhaps due to incorrect access permissions, it returns `NULL` and the program halts.

18

The `fseek` function in this line positions the file position pointer to the beginning of the file. This positioning is necessary because the append mode positions this pointer to the end of the file when `fopen` opens the file. `fseek` returns a zero if it is successful; otherwise, a nonzero return value indicates an error occurred. Consequently, the program verifies that no error occurred.

22

The function on this line reads in the five sets of phone numbers. `fread` requires four parameters:

- the address of a structure that receives the data,
- the number of bytes for each structure,
- the number of times the structure is read, and
- the stream pointer.

The `fread` function returns 1 if reading was successful.

24

This conditional ensures that `fread` reads the required data.

29

The `fseek` function in this line returns the file position pointer to the end of the file. Again, the program verifies the return value to ensure that no error occurred.

34-36

The program needs a `for` loop to write the data structures because they are appended to the contents of the file in reverse order. Consequently, the program must write data structures one at a time. Every time the loop calls the `fwrite` function, the first parameter contains the address of each data structure in the array.

37

Like the `fread` function, the `fwrite` function returns the number of elements sent to the designated file. This line verifies that all structures elements are written and prints an error otherwise.

41

This line flushes the buffer associated with the `fp` data structure and closes the connection with the file.

`fread` and `fwrite` transfer blocks of data of known size. Since all they require is a pointer to the data structure and its size, I/O is fast. But this approach has disadvantages. For example, the data structure must be of a constant size. In the previous program, the name field was limited to 8 characters. Strings of variable length cause problems; they must be read using other more explicit methods that do not use a storage area with a fixed size. One such function is `fgets`.

Data files created by the functions `fread` and `fwrite` are not portable to other computer systems. One cause of this

nonportability is the structure padding referred to in Appendix A, "Data types and representations." Structure padding occurs when the compiler aligns data types on boundaries in memory. This can cause blank spaces to appear inside a data structure.

These blank spaces accompany the data structure when it is written out using `fwrite`. For example, computer A may insert one blank byte between the `name` array and the `phone` array in the previous program, while computer B may insert no spaces. When computer B attempts to read computer A's output, it loses data.

Consequently, do not use these functions when you plan to transfer the data files to other computer systems.

Runtime library

7

This chapter provides a brief overview of functions that are available in the C libraries provided with CONVEX C; ANSI C, POSIX, and CONVEX functions are included. The information is organized by header files.

Note

The runtime libraries used on SPP Series systems are different than the ones used on C series machines. This chapter currently only describes the C Series header files and libraries.

Refer to local man pages for more information.

Under each header file, the functions are listed with a short description and some common macro definitions. The header files discussed in this chapter are:

- `assert.h`
- `ctype.h`
- `signal.h`
- `errno.h`
- `float.h`
- `limits.h`
- `locale.h`
- `math.h`
- `setjmp.h`
- `stdarg.h`
- `stddef.h`
- `stdio.h`
- `stdlib.h`
- `string.h`
- `time.h`

Note

The libraries used by the CONVEX C Compiler for SPP Series systems using SPP-UX are different. Refer to the online man pages on the machine you are using for more details on the library functions and include files.

Each section describes the ANSI C functions, associated structures, identifiers, and macros. Where applicable, the CONVEX extensions for C Series machines and POSIX extensions that are accessible with these ANSI C header files are also described. The compatibility mode of the compiler determines which extensions are available.

CONVEX C provides four compatibility modes as shown in Table 12.

Table 12 Compatibility modes

Mode	Option	Language	Default functions
Extended	`-ext`	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	`-std`	ANSI C	ANSI C, POSIX
Strict (C Series only)	`-str`	ANSI C	ANSI C
Backward-compatible	`-pcc`	Non-ANSI C, CONVEX	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first of the POSIX standards to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at the source-code level.

By avoiding nonstandard features of an operating system, an application has a greater chance of being ported to another computer system without major modifications. Refer to Chapter 3, "Compatibility modes," for more information on compatibility modes.

Functions versus function-like macros

Function-like macros are very similar to functions; a function-like macro is called in the same manner as a function. However, several differences exist:

- Function-like macros may increase the size of the code.
- Function-like macros are often faster; there is no function call overhead such as register saving.
- Function calls may reduce optimization.

- Contents of functions have local scope.
- Functions have an address.
- Functions evaluate their arguments only once.

Function-like macros are most suitable for routines that are used often but require little code; such routines include character input and output functions. Several of the header files, `ctype.h` and `stdio.h` in particular, include both functions and function-like macros for some of the routines, permitting the programmer to choose which version should be used. When a library function can be accessed by both functions and function-like macros, all function calls access the function-like macro by default.

Two ways to access the function instead of the macro are:

- Undefine the function-like macro using `#undef`.
- Surround the function name with parentheses.

For example, in the header file `ctype.h`, `isalpha` is declared as a function prototype and defined as a function-like macro. Two ways of calling the function are

```
#include <ctype.h>
#undef isalpha
int ch = isalpha('A');
```

and

```
#include <ctype.h>
int ch = (isalpha)('A');
```

The macro is called as usual:

```
#include <ctype.h>
int ch = isalpha('A');
```

Thus, you can select either a function or a function-like macro when a choice exists.

Functions that have function-like macros are identified in each function description.

Calling runtime functions

You can access a library function in two ways:

- Include the associated header file.
- Implicitly declare the function if no types from the header file are required.

Each method has its advantages and disadvantages.

Including a header file is easy and portable. Any data types, structures, and macros that use the function are automatically declared and defined. When an application is ported to another system, changes to data structures in header files probably do not require changes to the source code. This method is preferred.

For example, the code below uses the `memmove` function to replace the last four elements of `arra` with the first four elements.

```
#include <string.h>

char arra[10];

memmove( arra, &arra[6], 4 );
```

In the absence of a function prototype, the compiler implicitly declares any function it encounters. It performs integral promotions on each integral argument (that is, it promotes `char` and `short` to `int`), and promotes arguments of type `float` to type `double`.

In conclusion, header files are not always required to access a runtime function, but their inclusion is easy and assists in maintaining the program.

`assert.h`

This header file contains a function-like macro that halts a program if its argument is not true. This macro is useful in debugging applications.

ANSI C

`void assert(`*expression*`)`

Aborts program if *expression* is false and `NDEBUG` is not defined. Writes file name and line number of the file with the error.

CONVEX extension

`void _assert(`*expression*`)`

Equivalent to the `assert` function.

`ctype.h`

This header file contains character handling functions that are used for classifying character-coded integer values by table lookup. Each function returns a nonzero value for true or zero for false.

ANSI C

ANSI C declares these functions:

`int isalnum(int` *ch*`)`

Returns true for any letter or digit.

`int isalpha(int` *ch*`)`

Returns true for any letter.

`int iscntrl(int` *ch*`)`

Returns true for any control (nonprinting) character.

`int isdigit(int` *ch*`)`

Returns true for any decimal-digit character.

`int isgraph(int` *ch*`)`

Returns true for any printable character except a space.

`int islower(int` *ch*`)`

Returns true for any lowercase letter.

`int isprint(int` *ch*`)`

Returns true for any printable character including space.

`int ispunct(int` *ch*`)`

Returns true for any punctuation character. These characters are defined as all printing characters except spaces, digits, or letters.

`int isspace(int` *ch*`)`

Returns true for any space, tab, carriage return, newline, vertical tab, or form feed.

`int isupper(int ch)`

Returns true for any uppercase letter.

`int isxdigit(int ch)`

Returns true for any hexadecimal digit.

`int tolower(int ch)`

Returns the corresponding lowercase letter when the argument is an uppercase letter. If the argument is not an uppercase letter, it returns the argument unchanged.

`int toupper(int ch)`

Returns the corresponding uppercase letter when the argument is a lowercase letter. If the argument is not a lowercase letter, it returns the argument unchanged.

These functions are also declared as function-like macros:

- `isalnum`
- `isalpha`
- `iscntrl`
- `isdigit`
- `isgraph`
- `islower`
- `isprint`
- `ispunct`
- `isspace`
- `isupper`
- `isxdigit`

CONVEX extensions

These functions are CONVEX extensions:

`int isascii(int ch)`

Returns true for any ASCII character.

`int toascii(int ch)`

Returns the argument with all bits that are not part of the standard ASCII character turned off.

`int _tolower(int ch)`

Returns the same data as `tolower`, except it has a restricted domain and runs faster. If the argument to `_tolower` is not an uppercase letter, its result is undefined.

`int _toupper(int ch)`

Returns the same data as `toupper`, except it has a restricted domain and runs faster. If the argument to `_toupper` is not a lowercase letter, its result is undefined.

errno.h

This header file defines macro constants for error conditions.

ANSI C

`EDOM`

Contains the value that indicates a domain error.

`ERANGE`

Contains the value that indicates a range error.

The header also declares an external identifier:

`errno`

Contains a value indicating the most recent error. This identifier is accessed globally by several library routines.

float.h

This header file defines macro constants for floating-point data types.

ANSI C

`DBL_DIG`

Number of decimal digits in type `double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

`DBL_EPSILON`

Smallest number z of type `double` such that 1.0 + z =1.0.

`DBL_MANT_DIG`

Number of digits in the floating-point mantissa of the `double` type. The base is the value of `FLT_RADIX`.

`DBL_MAX`

Largest representable number of type `double`.

`DBL_MAX_10_EXP`

Largest integer z such that 10^z can be represented by a normalized number of type `double`.

`DBL_MAX_EXP`

Largest integer z such that $FLT\_RADIX^{(z-1)}$ is a normalized floating-point number of type `double`.

`DBL_MIN`

Smallest normalized positive number of type `double`.

`DBL_MIN_10_EXP`

Smallest representable integer z such that 10^z can be represented by a normalized number of type `double`.

`DBL_MIN_EXP`

Smallest integer z such that $FLT\_RADIX^{(z-1)}$ is a normalized floating-point number of type `double`.

`FLT_DIG`

Number of decimal digits in a type `float` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

`FLT_EPSILON`

Smallest number z of type `float` such that `1.0 + z=1.0`.

`FLT_MANT_DIG`

Number of digits in the floating-point mantissa of the `float` type. The base is the value of `FLT_RADIX`.

`FLT_MAX`

Largest representable number of type `float`.

`FLT_MAX_10_EXP`

Largest integer z such that 10^z can be represented by a normalized number of type `float`.

`FLT_MAX_EXP`

Largest integer z such that $FLT\_RADIX^{(z-1)}$ is a normalized floating-point number of type `float`.

`FLT_MIN`

Smallest normalized positive number of type `float`.

FLT_MIN_10_EXP

Smallest representable integer z such that 10^z can be represented by a normalized number of type `float`.

FLT_MIN_EXP

Smallest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `float`.

FLT_RADIX

Base number system for exponent representation.

FLT_ROUNDS

Rounding mode for floating-point addition. CONVEX computers round to the nearest representable value.

LDBL_DIG

Number of decimal digits in a number of type `long double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

LDBL_EPSILON

Smallest number z of type `long double` such that `1.0 + z=1.0`.

LDBL_MANT_DIG

Number of digits in the floating-point mantissa of the `long double` type. The base is the value of `FLT_RADIX`.

LDBL_MAX

Largest representable number of type `long double`.

LDBL_MAX_10_EXP

Largest integer z such that 10^z can be represented by a normalized number of type `long double`.

LDBL_MAX_EXP

Largest integer z such that $\texttt{FLT\_RADIX}^{(z-1)}$ is a normalized floating-point number of type `long double`.

LDBL_MIN

Smallest normalized positive number of type `long double`.

LDBL_MIN_10_EXP

Smallest representable integer z such that 10^z can be represented by a normalized number of type `long double`.

LDBL_MIN_EXP

Smallest integer z such that FLT_RADIX$^{(z-1)}$ is a normalized floating-point number of type long double.

limits.h

This header file defines macro constants that indicate sizes of integral data types.

ANSI C

Macros defined are:

CHAR_BIT

Number of bits in char data type.

CHAR_MAX

Maximum value for an object of type char.

CHAR_MIN

Minimum value for an object of type char.

INT_MIN

Minimum value for int data type.

INT_MAX

Maximum value for int data type.

LONG_MAX

Maximum value for long int data type.

LONG_MIN

Minimum value for long int data type.

MB_LEN_MAX

Maximum number of bytes in a multibyte character.

SCHAR_MIN

Minimum value for signed char data type.

SCHAR_MAX

Maximum value for signed char data type.

SHRT_MAX

Maximum value for short int data type.

SHRT_MIN

Minimum value for `short int` data type.

UCHAR_MAX

Maximum value for `unsigned char` data type.

UINT_MAX

Maximum value for `unsigned int` data type.

ULONG_MAX

Maximum value for `unsigned long int` data type.

USHRT_MAX

Maximum value for `unsigned short int` data type.

POSIX extensions

Macros defined are:

_POSIX_ARG_MAX

Largest number of bytes, including environment data, for one of the `exec` functions.

_POSIX_CHILD_MAX

For each real user ID, the maximum number of simultaneous processes.

_POSIX_LINK_MAX

A file's maximum link count value.

_POSIX_MAX_CANON

Maximum size of a terminal canonical input queue in bytes.

_POSIX_MAX_INPUT

The maximum buffer size for a terminal input queue in bytes.

_POSIX_NAME_MAX

The maximum character length of a file name.

_POSIX_NGROUPS_MAX

The maximum number of simultaneous supplementary group IDs for each process.

_POSIX_OPEN_MAX

The maximum number of files that can be open for one process simultaneously.

_POSIX_PATH_MAX

The maximum character length of a path name.

_POSIX_PIPE_BUF

The maximum number of bytes that can be written to a pipe without being interrupted.

The following macro constants are undefined (with `#undef`) by this header file:

- CHILD_MAX
- LINK_MAX
- MAX_INPUT
- MAX_CANON
- NAME_MAX

locale.h

Functions in this header file tailor the international output environment for functions that control character handling, string collation, date and time formatting, and numeric editing. This header file lets you customize the output of a program for a specific nationality.

Only the "C" locale is implemented in CONVEX C.

ANSI C

The structure defined is:

`lconv`

Contains the characters that can be changed in each locale.

The macros defined in this file represent categories that are used by the `setlocale` function. They are:

LC_ALL

A composite of all the other categories.

LC_COLLATE

This category impacts the comparison function used by the functions `strcoll` and `strxfrm`.

LC_CTYPE

This category impacts the following functions:

- `isalnum`
- `isprint`

- isalpha
- ispunct
- iscntrl
- isspace
- isdigit
- isupper
- isgraph
- isxdigit
- islower

This category does not impact the function-like macros of the same name. Therefore, you must undefine a function-like macro before you can use the function that LC_CTYPE affects.

LC_MONETARY

This category impacts information returned by the localeconv function regarding monetary characters.

LC_NUMERIC

This category impacts the decimal point character used by formatted input and output functions, string conversion functions, and non-monetary formatting information returned by the localeconv function.

LC_TIME

This category impacts the strftime function.

Functions declared in this header file are:

`struct lconv *localeconv(void)`

Returns a pointer to a structure of current locale information. The structure is in the static storage class and cannot be modified by the application.

`char *setlocale(int` *category*`, const char *`*locale*`)`

Changes or queries a program's current locale. *category* is a category macro defined in this header file. The *locale* argument may have three values:

- "" — sets the category to the locale of the minimal C translation.
- "C" — sets the category to the locale of the minimal C translation.
- NULL — queries the system for the current locale of the specified category.

Math function errors are of two kinds: domain errors (EDOM) and range errors (ERANGE). Domain errors occur when the size of an argument causes significant inaccuracy in the function result. When a domain error occurs, the value of the macro EDOM is stored in errno, an external identifier declared in errno.h.

Range errors result when the computed value cannot be represented within the machine's precision. When a range error occurs, the value of the macro ERANGE is stored in errno.

Caution

In the extended (default) compatibility mode, domain and range errors may not change the errno identifier. To force these errors to change the value of errno, recompile files that contain math functions with -D__NO_INLINE_MATH on the command line. Refer to Chapter 6, "CONVEX C intrinsics," for more information.

Domain and range errors that occur in the standard (-std) and strict (-str) compatibility modes do affect the errno identifier.

ANSI C

The macro defined is:

HUGE_VAL

Largest positive double precision value. It is not a constant expression and cannot be evaluated by the preprocessor.

This header declares the following ANSI functions. If an overflow occurs, the function returns HUGE_VAL; except for the tan function, the sign is the same as that of the actual result. If an underflow returns, the function returns zero and errno is set to the value of ERANGE.

`double acos(double x)`

Returns arccosine. The argument must be in the range [1,+1].

`double asin(double x)`

Returns arcsine. The argument must be in the range [1,+1].

`double atan(double x)`

Returns arctangent.

`double atan2(double numer, double denom)`

Returns arctangent of *numer/denom*. The arguments cannot both be zero.

`double ceil(double x)`

Returns smallest integer not less than the argument, *x*.

`double cos(double x)`

Returns cosine of the argument that is measured in radians.

`double cosh(double x)`

Returns hyperbolic cosine.

`double exp(double x)`

Returns exponential, e^x.

`double fabs(double x)`

Returns absolute value, $|x|$.

`double floor(double x)`

Returns largest integer not greater than the argument, *x*.

`double fmod(double` *numer*`, double` *denom*`)`

Returns floating-point remainder of *numer/denom*.

`double frexp(double` *number*`, int *`*pow*`)`

Returns *f*, where $1/2 \geq f$ or $f = 0$ and `number` = $f x 2^{(*pow)}$. If `number` = 0, **pow* and *f* are zero.

`double ldexp(double x, int` *pow*`)`

Returns $x * 2^{pow}$

`double log(double x)`

Returns natural logarithm of *x*.

`double log10(double x)`

Returns base-10 logarithm of x.

`double modf(double` *number*`, double *`*i*`)`

Splits `number` into a fraction, *f*, and an integer **i*, where *f* + (**i*) = *number*. The function returns *f*.

`double pow(double x, double` *pow*`)`

Returns x^{pow}.

`double sin(double x)`

Returns sine of the argument that is measured in radians.

`double sinh(double x)`

Returns hyperbolic sine.

`double sqrt(double x)`

Returns positive square root of the argument, $|\sqrt{x}|$.

`double tan(double x)`

Returns tangent of the argument that is measured in radians.

`double tanh(double x)`

Returns hyperbolic tangent.

Table 13 lists values returned by math functions when a domain error occurs.

Table 13 Math function return values

Function	Domain error return value
acos	acos(1)
asin	asin(1)
atan	pi/2
atan2	pi/2
cos	cos(8)
cosh	HUGE_VAL
sin	sin(0)
sinh	-HUGE_VAL
log	log(|x|)
log10	log10(|x|)
pow	pow(|x|)
sqrt	sqrt(|x|)

CONVEX extensions

A range or domain error in these functions sets global integer `errno` to the value of a math error defined in `errno.h`. These errors are more specific than EDOM and ERANGE.

`double atof(const char * str)`

Returns the `double` representation of the first number contained in the string pointed to by *str*. This file declares `atof` for convenience; the ANSI C standard declares this function in `stdlib.h`.

`double cabs(struct { double x, y;}, z)`

Returns *z*'s Euclidean length.

`double dcvtid(double x)`

Converts native mode input, *x*, into IEEE floating-point mode.

`double gamma(double x)`

Returns the log gamma of *x*.

`double hypot(double x, double y)`

Returns the Euclidean distance of *x* and *y*.

C Series only

`double idcvtd(double x)`

Converts *x* from IEEE to native format.

`int ipow(int x, int pow)`

Returns x^{pow}.

`float ircvtr(float x)`

Converts *x* from IEEE to native floating-point format.

`double j0(double x)`

Bessel functions of the first kind (j1, order 0).

`double j1(double x)`

Bessel functions of the first kind (j1, order 1).

`double jn(int n, double x)`

Bessel functions of the first kind (j1, order *n*).

C Series only

`long long int lpow(long long x, long long pow)`

Returns x^{pow}.

C Series only

`float rcvtir(float x)`

Converts *x* from native to IEEE format.

`double y0(double x)`

Bessel functions of the second kind (y1, order 0).

`double y1(double x)`

Bessel functions of the second kind (y1, order 1).

`double yn(int n, double x)`

Bessel functions of the second kind (y1, order *n*).

For descriptions of these functions, refer to their man pages.

setjmp.h

This header file declares two functions that you can use instead of the normal function call and return paradigm.

ANSI C

The type defined is:

`jmp_buf`

Declares an identifier that retains the context of the calling environment. It is used by the two functions declared in this header file.

Functions declared are:

`int setjmp(jmp_buf` *buffer*`)`

Save the calling environment context in *buffer*.

`void longjmp(jmp_buf` *buffer*`, int` *retval*`)`

Restore the environment context saved in *buffer*. Program execution resumes with the statement following the `setjmp` function associated with *buffer*. The `setjmp` function returns *retval* if it is nonzero; if *retval* = `0`, the `setjmp` function returns 1.

`int _setjmp(jmp_buf` *buffer*`);`

`int _longjmp(jmp_buf` *buffer*`, int` *retval*`);`

`setjmp` and `longjmp` save and restore the signal mask `signalmask(2)`, while `_setjmp` and `_longjmp` manipulate only the C stack and registers.

`longjmp` may not be used to restore the environment saved by `_setjmp`, and `_longjmp` may not be used to restore the environment saved by `setjmp`. An error condition occurs in both cases.

POSIX extensions

The type defined is:

`sigjmp_buf`

Declares a location in which to store the calling environment information.

Functions declared are:

`int sigsetjmp(sigjmp_buf` *env*`, int` *savemask*`)`

Saves the calling environment in *env* and if *savemask* is nonzero, saves the process's current signal mask as part of the calling environment.

`void siglongjmp(sigjmp_buf` *env*`, int` *savemask*`)`

Restores the calling environment saved in *env* and if the signal mask is saved in the calling environment, restores that as well.

Restrictions for parallel programming

Use the `setjmp` and `longjmp` functions with care within programs that contain parallelism. You may not `longjmp` into or out of a parallel construct.

signal.h

This header file declares the functions that control the behavior of a program when signals occur. It also defines macro constants that represent various signals.

ANSI C

The type defined is:

`sig_atomic_t`

No interrupts are recognized when a value is assigned to an object of this type. This type is useful for communication between the program and its signal handlers.

The functions are:

`void (*signal(int` *sig*`, void (*`*func*`)(int)))(int)`

Define the behavior of the program when a particular signal is received.

`int raise(int` *sig*`)`

Sends a signal to the executing program.

A function-like macro is:
`int raise(`*sig*`)`

CONVEX extensions

Refer to the header file for a list of the signals. Basic categories are:

- Traps
- Floating-point exceptions
- Bus errors
- Segment errors

Structures defined are:

`sigaction`

An aggregate of a signal handler, a bit mask, and some flags.

`sigvec`

Overlaid by `struct sigaction`.

`sigcontext`

Contains information pushed on the stack when a signal is delivered. It is made available to the handler to allow it to properly restore state if a non-standard exit is performed.

A function-like macro is:

`int sigmask(`*number*`)`

Creates a mask for the bit indicated by *number*. For example, the value of `sigmask(5)` is 16.

A function is:

```
int (*signal(int sig, void (*func)(int sig,
     int subcode,
     struct sigcontext *scp)))(int)
```

Defines the behavior of the program when it receives a particular signal. The difference between this function and the ANSI C signal function is that the function *func* accepts three parameters, while the ANSI C signal handler accepts only one parameter.

POSIX extensions

The structures defined are:

`sigaction`

An aggregate of a signal handler, a bit mask, and some flags.

`sigvec`

Overlaid by `struct sigaction`.

The following three macros defined are used as the first argument to the `sigprocmask` macro. They determine how the signal set of a process is changed.

`SIG_BLOCK`

Resulting signal set is a combination of the current signal set and the signal set pointed to by another argument to the function.

`SIG_UNBLOCK`

Resulting signal set is an intersection of the current signal set and the complement of the signal set pointed to by another argument to the function.

`SIG_SETMASK`

Resulting signal set is the signal set pointed to by another argument to the function.

Functions declared are:

`int kill(pid_t` *pid*`, int` *sig*`)`

Sends signal *sig* to the process specified by *pid*. Returns 0 if the function is successful; otherwise it returns `-1`.

`int sigaction(int` *sig*`, struct sigaction *`*act*`, struct sigaction *`*oact*`)`

Customizes or examines the action associated with a particular signal. `act` is a pointer to the new action; `oact` is used to store the old action if it is not `NULL`. If `act` is `NULL`, the signal handling is not changed. Returns 0 if the function is successful; otherwise, `-1`. Returns 0 if the function is successful; otherwise, `-1`.

`int sigaddset(sigset_t *`*set*`, int` *signo*`)`

Adds the individual signal specified by the value of `signo` to the signal set pointed to by `set`. Returns 0 if the function is successful; otherwise, `-1`.

`int sigdelset(sigset_t *`*set*`, int` *signo*`)`

Removes the individual signal specified by the value of *signo* from the signal set pointed to by *set*. If the function detects no errors, it returns 0; otherwise, it returns `-1`.

`int sigemptyset(sigset_t *set)`

Excludes all POSIX signals from the initialization of the signal set pointed to by *set*. If the function detects no errors, it returns 0; otherwise, it returns `-1`.

`int sigfillset(sigset_t *set)`

Includes all POSIX signals in the initialization of the signal set pointed to by *set*. If the function detects no errors, it returns 0; otherwise, it returns `-1`.

`int sigismember(sigset_t *set, int signo)`

Tests whether the set pointed to by *set* contains the signal specified by the value of *signo*. Returns 1 if the signal is a member of the set, and 0 if it is not. If an error occurs, the function returns `-1`.

`int sigpending(sigset_t *set)`

set is the storage location for the set of signals that are blocked from delivery and pending for the calling process. Returns 0 if it successful, and `-1` if there is an error.

`int sigprocmask(int how, sigset_t *set, sigset_t *oset)`

Examines or changes the calling process's signal mask. *how* is one of the three macros described previously; *set* lists the modifications required as specified by *how*; and *oset* is the resulting signal set. If *set* is NULL no modifications take place, and oset contains the current signal set. Returns 0 if function is successful, or `-1` otherwise.

`int sigsuspend(sigset_t *sigmask)`

The set of signals pointed to by *sigmask* replaces a process's current signal mask. The process is suspended until it receives a signal that terminates it or a signal that forces it to execute a signal-catching function. Returns a `-1` if an error occurs.

stdarg.h

The contents of this file are used to access parameters of functions that may have a variable number of arguments.

ANSI C

The type defined is:

`va_list`

Declares a variable argument list structure.

Function-like macros are:

`type *va_arg(va_list` *list*`,` *type*`)`

Returns the next parameter in the argument list. The second argument is the type of the next parameter.

`void va_end(va_list` *list*`)`

Releases access to the structure defined by `va_list`.

`void va_start(va_list` *list*`,` *LastParam*`)`

Initializes the data structure list. *LastParam* is the last nonvariable parameter in the function argument list.

stddef.h

This file defines useful types and constants for pointer and size operations.

ANSI C

Types defined are:

`ptrdiff_t`

Type resulting from the subtraction of two pointers.

`size_t`

Type returned by the `sizeof` macro operator.

`wchar_t`

Data type for wide characters.

The macro defined is:

`NULL`

Null pointer constant.

Function-like macro is:

`size_t offsetof(`*structure*`,` *field*`)`

Returns the offset, in bytes, of a structure member from the base address of its structure.

stdio.h

This header file declares types, macros, and functions that are used for input and output.

ANSI C

Macros defined are:

`BUFSIZ`

Buffer size for `setbuf`.

`EOF`

Value returned by functions that indicates the end of a file.

`FILENAME_MAX`

Maximum number of characters permitted in a file name.

`FOPEN_MAX`

Maximum number of files that can be open simultaneously.

`_IOFBF`

Parameter of `setvbuf` function that indicates full buffering for input/output.

`_IOLBF`

Parameter of `setvbuf` function that indicates line buffering for input/output.

`_IONBF`

Parameter of `setvbuf` function that indicates no buffering for input/output.

`L_tmpnam`

Maximum length of a file name that is returned by the `tmpnam` function.

`NULL`

Null pointer constant.

`SEEK_CUR`

Parameter of the `fseek` function that sets the file reference point at the beginning of the file.

`SEEK_END`

Parameter of the `fseek` function that sets the file reference point at the end of the file.

`SEEK_SET`

Parameter of the `fseek` function that sets the file reference point at the current file position.

`stderr`

File pointer for standard error stream.

`stdin`

File pointer for standard input stream.

`stdout`

File pointer for standard output stream.

`TMP_MAX`

Maximum number of unique file names that can be produced by the `tmpnam` function.

The types defined are:

`FILE`

Object that records all information needed to access a file.

`fpos_t`

Object that retains all information required to uniquely specify a file position.

`size_t`

Type returned by the `sizeof` macro operator.

Operations on files

`int remove(const char *`*filename*`)`

Discards the link between a file name and its file contents. Returns a nonzero value when an error occurs.

`int rename(const char *`*old*`, const char *`*new*`)`

Associates a new file name with a file. The link between **old* and the contents of the file is removed. Returns a nonzero value when an error occurs.

`FILE *tmpfile(void)`

Creates a temporary binary file. When the program terminates, the file is removed automatically. `NULL` is returned if an error occurs.

`char *tmpnam(char *`*filename*`)`

Returns a valid, unique file name.

File access functions

`int fclose(FILE *file)`

Flushes the buffer associated with *file*, then closes it. Returns EOF if an error occurs.

`int fflush(FILE *file)`

Flushes the buffer associated with *file*. Returns EOF if a write error occurs.

`FILE *fopen(const char *filename, const char *mode)`

Initiates access to a file. *mode* determines the type of access. Returns a pointer to the file structure or NULL if an error occurs.

```
FILE *freopen(const char *filename,
    const char *mode, FILE *fp)
```

Associates a new file with an existing file pointer. Any file already associated with the stream is closed. Access to the file is specified by *mode*. Returns NULL if an error occurs.

`void setbuf(FILE *fp, char *buffer)`

Establishes full buffering for a stream with *buffer* or causes input and output with a file pointer, *fp*, to be unbuffered.

```
int setvbuf(FILE *fp, char *buffer, int buf_type,
    size_t size)
```

Tailors the type of buffering associated with a file pointer to fully buffered, line buffered, or unbuffered. The third parameter selects the buffer type. It can have one of three values: _IOFBF, _IOLBF, or _IONBF. It also permits a user-defined buffer, specified by *buffer* and *size*. Returns a nonzero value if an error occurs.

Formatted input/output functions

`int fprintf(FILE *fp, const char *format, ...)`

Formats text, as specified by *format* and writes the text to the file pointed to by *fp*. Returns the number of characters written.

`int fscanf(FILE *fp, const char *format, ...)`

Reads input from the file pointed to by *fp* using *format* and places the input data in the addresses specified by additional arguments. Returns EOF if no data is read; otherwise, it returns the number of conversions that were performed.

`int printf(const char *format, ...)`

Performs the same function as `fprintf` but writes to `stdout`.

`int scanf(const char *format, ...)`

Performs the same function as `fscanf` but reads from `stdin`.

`int sprintf(char *array, const char *format, ...)`

Performs the same function as `fprintf`, except the formatted text is written into the array pointed to by *array*. The null character is appended to the formatted text.

`int sscanf(const char *array, const char *format, ...)`

Performs the same function as `fscanf`, except the input is obtained from the array pointed to by *array*.

`int vfprintf(FILE *stream, const char *format, va_list arg)`

Performs the same function as `fprintf`. *arg* specifies the variable argument list. *arg* must be initialized with `va_start` before `vfprintf` is called. The header file `stdarg.h` must be included when this function is used.

`int vprintf(const char *format, va_list arg)`

Performs the same function as `printf`. *arg* specifies the variable argument list. *arg* must be initialized with `va_start` before `vprintf` is called. The header file `stdarg.h` must be included when this function is used.

`int vsprintf(char *array, const char *format, va_list arg)`

Performs the same function as `sprintf`. *arg* specifies the variable argument list. *arg* must be initialized with `va_start` before `vsprintf` is called. The header file `stdarg.h` must be included when this function is used.

Character input and output functions

`int fgetc(FILE *fp)`

Returns the next character from the file pointed to by *fp*. If the end of the file is encountered or an error occurs, EOF is returned.

`char *fgets(char *str, int n, FILE *fp)`

Reads *n*-1 characters, or up to a newline character, from the file pointed to by *fp* into the string pointed to by *str*.

`int fputc(int ch, FILE *fp)`

Outputs a character to the file pointed to by *fp*. `fputc` performs the same function as `putc`, except that `fputc` is a function, whereas `putc` is a macro.

`int fputs(const char *str, FILE *fp)`

Copies the null-terminated string *str* to the file pointed to by *fp*. Returns EOF if a write error occurs.

`int getc(FILE *fp)`

Returns the next character from the file pointed to by *fp*, or EOF if the end of the file is encountered or an error occurs.

`int getchar(void)`

Returns the next character from `stdin`, or EOF if the end of the file is encountered or an error occurs.

`char *gets(char *str)`

Reads a string from the standard input `stdin`. Reads up to a newline character or the end of the file. A null pointer is returned if an error occurs.

`int putc(int ch, FILE *fp)`

Outputs a character to the file pointed to by *fp*. Returns EOF if a write error occurs.

`int putchar(int ch)`

Outputs a character to the standard output, `stdout`. Returns EOF if a write error occurs.

`int puts(const char *str)`

Writes the null-terminated string pointed to by *str* to the standard output, `stdout`, and appends a newline character. Returns EOF if an error occurs.

`int ungetc(int ch, FILE *fp)`

Pushes character, *ch*, back into an input buffer associated with the file pointer *fp*. Returns EOF if an error occurs.

Direct input and output functions

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)`

Reads a block of data from the file associated with *fp*. *ptr* is the base address of the array for the data; *nmemb* is the number of elements in the array; and *size* is the number of

bytes required for each element in the array. Returns the number of array elements actually read.

`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`

Writes a block of data to the file associated with *fp*. *ptr* is the base address of the data to write; *nmemb* is the number of items to write; and *size* is the number of bytes that represents each item. Returns the number of items actually written.

File positioning functions

`int fgetpos(FILE *fp, fpos_t *pos)`

pos is set equal to the current file position indicator of the file pointed to by *fp*. If an error occurs, a nonzero value is returned, and `errno` is set equal to one of the values in Table 14.

Table 14 `errno` values of `fgetpos`, `fsetpos`, and `ftell`

Value returned	Error condition
EBADF	File is not open
ESPIPE	File name is associated with a pipe or socket
EINVAL	File position is negative

`int fseek(FILE *fp, long int offset, int ref_pt)`

Sets the file position of the file pointed to by *fp* to the position indicated by *ref_pt* and *offset*. *ref_pt* can be one of three values: `SEEK_CUR`, the current file position; `SEEK_SET`, the beginning of the file; and `SEEK_END`, the end of the file. *offset* is the number of bytes from the reference point. Returns a nonzero value when a domain error occurs.

`int fsetpos(FILE *fp, const fpos_t *pos)`

Sets file position of file pointed to by *fp* equal to value stored in *pos* that originated in a prior call to `fgetpos`. Sets the file position indicator for `stream` according to the value of the object pointed to by *pos*, which is a value obtained from an earlier call to `fgetpos` on the same file.

`long ftell(FILE *fp)`

Returns the offset of the current file position of the file pointed to by *fp* from the beginning of the file. The value is

measured in bytes. If an error occurs, it returns `-1L` and sets `errno` equal to a value listed in Table 14.

`void rewind(FILE *fp)`

Sets the file position indicator to the beginning of the file pointed to by *fp*.

Error-handling functions

`void clearerr(FILE *fp)`

Clears end-of-file and error indicators for the file associated with *fp*.

`int feof(FILE *fp)`

Returns a nonzero value if the end-of-file indicator for the file associated with *fp* is set.

`int ferror(FILE *fp)`

Returns a nonzero value if the error indicator of the file associated with *fp* is set.

`void perror(const char *str)`

Writes the error message associated with the value stored in `errno` to `stderr`. This message is preceded by the string pointed to by *str*. For more information about C Series error messages, see the intro(2) man page; for information about SPP Series messages, see the errno(2) man page.

Routines in the following list are available as function-like macros and as functions:

- `getc`
- `putchar`
- `getchar`
- `feof`
- `putc`
- `ferror`

CONVEX extensions

The functions declared are:

`FILE *popen(char *str, char *mode)`

Creates a pipe between the calling process and the command to be executed, which is pointed to by *str*. *mode* can be `"r"` for reading or `"w"` for writing. The value returned is a pointer

that can be used to write to the standard input of the command or read from its standard output. If an error occurs, NULL is returned.

`int pclose(FILE *pp)`

Waits for the process associated with the pipe pointer, *pp*, to terminate and returns the exit status of the command. Returns -1 on a domain error.

POSIX extensions

Two macros are defined:

`L_cuserid`

Maximum length of the controlling user ID, including the terminating null byte.

`L_ctermid`

Maximum length of the controlling terminal ID, including the terminating null byte.

The following functions are declared:

`int *fdopen(int fildes, const char *type)`

Associates a file with a file descriptor.

`int fileno(const FILE *fp)`

Returns the integer file descriptor associated with *fp*. This is also available as a function-like macro.

`char *cuserid(char *str)`

Returns a name associated with the effective user ID of the process.

stdlib.h

This header file contains declarations for general utilities.

ANSI C

The macros defined are:

`EXIT_FAILURE`

Argument of `exit` that indicates abnormal termination status.

`EXIT_SUCCESS`

Argument of `exit` that indicates normal termination status.

`NULL`

Null pointer constant.

`MB_CUR_MAX`

Number of bytes required to represent a multibyte character in the current locale.

`RAND_MAX`

The maximum integer returned by `rand`.

The types defined in this file are:

`size_t`

Result type of the `sizeof` operator.

`wchar_t`

Data type for wide characters.

`div_t`

Return type of the `div` function.

`ldiv_t`

Return type of the `ldiv` function.

String conversion functions

`double atof(const char *nptr)`

Returns the `double` representation of the initial portion of the string pointed to by *nptr*.

`int atoi(const char *nptr)`

Returns the `int` representation of the initial portion of the string pointed to by *nptr*.

`int atol(const char *nptr)`

Returns the `long int` representation of the initial portion of the string pointed to by *nptr*.

`double strtod(const char *nptr, char **endptr)`

- Returns the `double` representation of the initial portion of the string pointed to by *nptr*.
- Returns 0 if conversion is unsuccessful.
- Returns `HUGE_VAL` and sets `errno` to `ERANGE` if an overflow occurs.
- Returns 0 and sets `errno` to `ERANGE` if an underflow occurs.

`long strtol(const char *nptr, char **endptr, int base)`

- Returns the `long int` representation of the initial portion of the string pointed to by *nptr*.
- Returns 0 if conversion is unsuccessful.
- If a range error occurs it sets `errno` to `ERANGE` and returns `LONG_MIN` if the sign of the answer is negative and `LONG_MAX` if the sign is positive.

`unsigned long strtoul(const char *nptr, char **endptr, int base)`

Returns the `unsigned long int` representation of the initial portion of the string pointed to by *nptr*. Returns 0 if conversion is unsuccessful. If a range error occurs, `errno` is set to `ERANGE` and `ULONG_MAX` is returned.

Pseudo-random sequence generation functions

`int rand(void)`

Returns a nonnegative integer less than or equal to `RAND_MAX`.

`void srand(unsigned int seed)`

Reinitialize the sequence of pseudo-random integers.

Memory management functions

`void *calloc(size_t nmemb, size_t size)`

Allocates a block of memory for *nmemb* objects, each of size *size*.

`void free(void *pm)`

Deallocates the memory pointed to by *pm*.

`void *malloc(size_t size)`

Returns a pointer to a memory block that has *size* bytes. Returns a null pointer if sufficient memory is not available.

`void *realloc(void *ptr, size_t size)`

Changes the memory size originally allocated to *ptr*. The new memory size requirement is specified by *size*. Returns the new memory pointer or a null pointer if an error occurs.

Communication with the environment functions

`void abort(void)`

Causes program to terminate abnormally if the signal `SIGABRT` is not cleared by a signal handler. Does not return program execution to its caller.

`int atexit(void (*`*`func`*`)(void))`

The function pointed to by *func* is registered. Functions that are registered are executed when a program ends. The *func* function is called without arguments at normal program termination. The `atexit` function returns a nonzero value if the *func* function could not be registered.

`void exit(int `*`status`*`)`

Causes program to terminate normally.

`char *getenv(const char *`*`name`*`)`

Searches environment list for a string of the form `name=`*value*. Returns a pointer to the string *value* if such a string is present. If such a string is not present, `getenv` returns the `NULL` pointer.

`int system(const char *`*`string`*`)`

Issues a shell command. The current process waits until the shell has completes the command indicated by `string`, then returns the exit status of the shell.

Searching and sorting functions

```
void *bsearch(const void *key,
      const void *base,
      size_t nmemb, size_t size,
      int (*compare)(const void *, const void *))
```

Searches for *key* in the array pointed to by *base* using the function *compare*. Each array element has *size* bytes. Returns a pointer to the matching element of the array if it exists; otherwise it returns a null pointer.

```
void qsort(void *base, size_t nmemb,
      size_t size,
      int (*compare)(const void *, const void *))
```

Provides a quick-sort utility. *base* is a pointer to the base of the data; *nmemb* is the number of elements; *size* is the width of an element in bytes; *compare* is the name of the routine that compares two elements in the array.

Integer arithmetic functions

`int abs(int j)`

Returns the absolute value of *j*.

`div_t div(int numer, int denom)`

Returns the quotient and remainder of the arguments. The structure returned has two members, `quot` and `rem`.

`long labs(long j)`

Returns the `long` representation of the absolute value of *j*.

`ldiv_t ldiv(long numer, long denom)`

Returns the `long` representation of the quotient and remainder of the arguments. The return value has two members, `quot` and `rem`.

Multibyte character functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. If `*str` is `NULL` in any of the following functions, `0` is returned indicating that no state-dependent encodings exist.

`int mblen(const char *str, size_t n)`

Returns the number of bytes, up to *n*, of the multibyte character pointed to by `str`. This is always 1 unless *n* = 0 or `*str` = `NULL`.

`int mbtowc(wchar_t *pwc, const char *str, size_t n)`

Because all multibyte characters are length 1, the first character pointed to by `*str` is copied to `*pwc` and 1 is returned. However, if n = 0, no conversion is performed and -1 is returned.

`int wctomb(char *str, wchar_t wchar)`

Converts the multibyte character code stored in the object `wchar` into a multibyte character whose base address is *str*. Returns 1 because all multibyte characters are of length 1.

Multibyte string functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. Conversions between multibyte strings and wide characters are trivial because no special encodings are supplied by CONVEX. If `*str` is `NULL` in any of the following functions, `0` is returned, indicating that no state dependent encodings exist.

`size_t mbstowcs(wchar_t *pwcs, const char *str, size_t n)`

Copies *n* bytes, or until NULL is encountered, of the multibyte string pointed to by **str* to the wide character string pointed to by **pwcs*. Returns the number of multibyte characters copied.

`size_t wcstombs(char *str, const wchar_t *wcs, size_t n)`

Copies an array of *n* or fewer multibyte character encodings into a multibyte character string. Returns one less than the number of characters copied.

string.h

String-handling functions are used to manipulate character arrays and blocks of memory.

ANSI C

One macro is defined:

`NULL`

Null pointer constant.

One type is defined:

`size_t`

Type returned by the `sizeof` macro operator.

Copying functions

`void *memcpy(void *str1, const void *str2, size_t n)`

Copies the array of type `char` pointed to by *str2* into the array pointed to by *str1*. *n* or fewer elements are copied. If the arrays overlap, the behavior is undefined. Returns *str1*.

`void *memmove(void *str1, const void *str2, size_t n)`

Copies the array of type `char` pointed to by *str2* into the array pointed to by *str1*. *n* or fewer elements are copied. The arrays may overlap. Returns *str1*.

`char *strcpy(char *str1, const char *str2)`

Copies the null-terminated string pointed to by *str2* into the string pointed to by *str1*. If the strings overlap, the behavior is undefined. Returns *str1*.

```
char *strncpy(char *str1,const char *str2,size_t n)
```

Copies exactly *n* characters from string pointed to by *str2* into the string pointed to by *str1*, truncating or null-padding if necessary. If the strings overlap, the behavior is undefined. Returns *str1*.

Concatenation functions

```
char *strcat(char *str1, const char *str2)
```

Concatenates the string pointed to by *str2* at the end of the string pointed to by *str1*. The null-terminating character of *str1* is overwritten by the first character of *str2*. If the strings overlap, the behavior is undefined. Returns *str1*.

```
char *strncat(char *str1, const char *str2, size_t n)
```

Concatenates *n* or fewer characters of the string pointed to by *str2* at the end of the string pointed to by *str1*. The null-terminating character of *str1* is overwritten by the first character of *str2*. If the strings overlap, the behavior is undefined. Returns *str1*.

Comparison functions

```
int memcmp(const void *buf1, const void *buf2,
    size_t n)
```

Compares the first *n* bytes of *buf1* and *buf2*. Each byte is considered to be an `unsigned char`. Returns:

- A positive integer if *buf1* is greater than *buf2*.
- Zero if the two buffers are the same.
- A negative integer if *buf1* is less than *buf2*.

```
int strcmp(const char *str1, const char *str2)
```

Compares characters in two strings and returns:

- A positive integer if the string pointed to by *str1* is greater than the string pointed to by *str2*.
- Zero if the two strings are the same.
- A negative integer if the string pointed to by *str1* is less than the string pointed to by *str2*.

```
int strcoll(const char *str1, const char *str2)
```

Compares characters in two strings. The comparison function is affected by the current locale category `LC_COLLATE`. Returns:

- A positive integer if the string pointed to by *str1* is greater than the string pointed to by *str2*.
- Zero if the two strings are the same.
- A negative integer if the string pointed to by *str1* is less than the string pointed to by *str2*.

```
int strncmp(const char *str1, const char *str2,
    size_t n)
```

Compares not more than *n* characters of two strings (characters after a NULL are not compared) and returns:

- A positive integer if the string pointed to by *str1* is greater than the string pointed to by *str2*.
- Zero if the first *n* characters of the two strings are the same.
- A negative integer if the string pointed to by *str1* is less than the string pointed to by *str2*.

```
size_t strxfrm(char *str1, const char *str2,
    size_t n)
```

Transforms sufficient characters in the string pointed to by *str2* such that the string pointed to by *str1* will have *n* or fewer characters. The transformation does not change the expected result of the `strcoll` function. Returns the number of characters in the transformed string.

Search functions

```
void *memchr(const void *str, int ch, size_t n)
```

Returns a pointer to the first location of the character *ch* (interpreted as `unsigned char`) in the first *n* characters of the string pointed to by *str*. If no character is found, it returns a NULL pointer.

```
char *strchr(const char *str, int ch)
```

Returns a pointer to the first location of the character *ch* (interpreted as `char`) in the string pointed to by *str*. If no character is found, it returns NULL.

```
size_t strcspn(const char *str1, const char *nset)
```

Returns the length of the largest initial substring of the string pointed to by *str1*. The substring cannot contain any characters in the string pointed to by *nset*.

```
char *strpbrk(const char *str1, const char *set)
```

Returns a pointer to the first occurrence in the string pointed to by *str1* of any character in the string pointed to by *set*.

Returns a pointer to such a character or a NULL pointer if a matching character is not found.

`char *strrchr(const char *str, int ch)`

Returns a pointer to the last location of the character *ch* (interpreted as `char`) in the string pointed to by *str*. If no character is found, it returns NULL.

`size_t strspn(const char *str1, const char *set)`

Returns the length of the largest initial substring of the string pointed to by *str1* consisting only of characters contained in the string pointed to by *set*.

`char *strstr(const char *str1, const char *pattern)`

Returns a pointer to the first substring that matches the string pointed to by *pattern* in the string pointed to by *str1*. Returns NULL if no such substring is found.

`char *strtok(char *str, const char *delimiters)`

Divides the string pointed to by *str* into tokens delimited by the characters in the string pointed to by *delimiters*. The first function call returns a pointer to the first token in *str*. Delimiters are overwritten by the NULL character. Subsequent calls to `strtok` must have NULL as the first argument; a pointer to the next token is returned. If a token is not found, the return value is NULL.

Miscellaneous functions

`void *memset(void *buf, int ch, size_t n)`

Initializes the first *n* characters of the array pointed to by `buf` with the character *ch*.

`char *strerror(int errno)`

Returns a pointer to the error message associated with the error number *errno*. For a list of the error messages associated with each value of *errno* on C Series systems, see the intro(2) man page; for SPP Series messages, see the errno(2) man page. `strerror` is also available as a function-like macro.

`size_t strlen(const char *str)`

Returns the number of characters in the string pointed to by *str*.

CONVEX extensions

Functions declared are:

`char *index(char *string, int ch)`

Returns a pointer to the first occurrence of the character *ch* (interpreted as `char`) in the string pointed to by *string*. If no character is found, it returns `NULL`.

`char *rindex(char *string, int ch)`

Returns a pointer to the last occurrence of the character `ch` (interpreted as `char`) in the string pointed to by *string*. If no character is found, it returns `NULL`.

time.h

This file defines useful types and functions for manipulating time.

ANSI C

Macros defined in this header file are:

`NULL`

Null pointer constant.

`CLOCKS_PER_SEC`

The number of times the system clock increments for each second.

Types defined in this header file are:

`size_t`

The type returned by the `sizeof` macro operator.

`clock_t`

The type returned by the clock function.

`time_t`

The type returned by the time function.

One structure is defined:

`tm`

Members of this structure are:

`tm_sec`

Seconds after the minute.

`tm_min`

Minutes after the hour.

`tm_hour`

Hours since midnight.

`tm_mday`

Day of the month.

`tm_mon`

Months since January.

`tm_year`

Years since 1990.

`tm_wday`

Days since Sunday.

`tm_yday`

Days since January 1.

`tm_isdst`

Daylight savings time flag: positive if DST is in effect; 0 if DST is not in effect; negative if DST status is unknown.

Time manipulation functions

`clock_t clock(void)`

Returns the execution time for the current process.

`double difftime(time_t` *time1*`, time_t` *time0*`)`

Returns the difference in seconds between two times. This is also available as a function-like macro.

`time_t mktime(struct tm *`*timeptr*`)`

Converts the time in `tm` representation to time in `time_t` representation. Returns the converted value or `-1` if the conversion cannot be performed. The structures are described above.

`time_t time(time_t *`*timer*`)`

Returns the current time in a `time_t` representation. If *timer* is not `NULL`, the value returned is also stored in *timer*. the `time_t` structure is described above.

Time conversion functions

`char *asctime(const struct tm *timeptr)`

Converts a time in the `tm` representation into a string in the format:

```
Wed Apr 25 23:26:45 1962\n\0
```

(The `\n` and `\0` values are a carriage-return and null value, respectively.) Returns a pointer to the string.

`char *ctime(const time_t *timer)`

Converts the `time_t` representation pointed to by *timer* into an ASCII representation.

`struct tm *gmtime(const time_t *timer)`

Converts time pointed to by *timer* from the `time_t` structure to the `tm` structure.

`struct tm *localtime(const time_t *timer)`

Converts time returned by the time function, pointed to by *timer*, to a structure containing the broken-down time, correcting for time zone and if necessary, daylight savings time.

`size_t strftime(char *str, size_t n,`
`const char *format, const struct tm *time)`

Converts `time` into a string pointed to by *str* using the format specified by *format*. The number of characters produced cannot be greater than *n*. If *n* is exceeded, the contents of the string are undefined and 0 is returned; otherwise, the number of characters produced is returned.

POSIX extensions

One macro is defined:

`CLK_TCK`

Number of clock ticks for each second.

One external identifier is declared:

`tzname`

An array of time zone names.

Preprocessor directives

8

The preprocessor allows you to define macros for constants, define function-type macros, and conditionally compile sections of code.

- **Macros for constants**—You can increase program readability and simplify program maintenance by defining macros for constants or short pieces of code. For example, if some functions use π, it is better to define a macro with a meaningful name that contains the value for π rather than to use the number 3.14 throughout the source file. Thus, if you need greater precision, you only need to change the value for π in only one place in the source file.
- **Function-type macros**—These macros are shorthand for one or more lines of source code that use one or more parameters. For example, if you use a set of three statements frequently throughout a source file, you may define a macro that represents the three lines. Then, whenever you need the three lines, you only need to type the macro name. The sections below explain function-like macros in more detail.
- **Conditional compilation**—Conditional compilation is a technique used to select the source code to be compiled. For example, you may use it to remove parts of the code that produce debugging information that is not needed in the completed application. Conditional compilation also makes it easier to port source code between computer systems.

Preprocessor directives are indicated by a # symbol which may appear anywhere on a line as long as it is preceded only by white space and C comments. The directive must follow the # symbol on the same line with only blanks, horizontal tabs, or C comments separating the two. The directive only affects the line on which it appears; you cannot define multiline directives.

Preprocessor directives may have zero or more arguments.

#define

The syntax of this directive is:

```
#define name definition
```

The definition for this directive is terminated by the newline character. In its simplest form, this directive defines constants. For example, the following line provides a definition for π:

```
#define PI 3.14159265
```

After defining this constant, use only the identifier `PI` when the value of π is required.

When you need more precision, change the definition of `PI` to:

```
#include <math.h>
#define PI (4.0 * atan(1.0))
```

Because the definition of a name is terminated by a newline character, it is not limited to one word. When used in this manner, `#define` performs simple text substitution.

A multiword definition is:

```
#define UCHAR unsigned char
```

Thus, this type of definition replaces commonly used pieces of code.

The backslash or continuation character (\) placed at the end of a line permits the definition of a name to extend over multiple lines. The backslash character must be followed by a newline; it cannot be followed by blanks, tabs, or comments. Lines connected with the backslash character are recognized by the preprocessor as a single line. For example, the preprocessor considers the following definition to be on one line:

```
#define close_files fclose( file1 );\
   fclose( file2 );\
   fclose( file3 )
```

When the word `close_files` is seen in the source file in which this definition is visible, the preprocessor replaces it with the three C statements.

Finally, this directive may define a function-like macro that takes arguments. For example, the previous definition may be written as:

```
#define *close_files( file1, file2, file3 )\
fclose( file1 );\
fclose( file2 );\
fclose( file3 )
```

This feature permits macro definitions to be more flexible. A macro function that takes arguments must have the left parenthesis immediately follow the macro name; no intervening spaces are permitted.

Even though a function-like macro looks like a C function, there are differences:

- Function calls require more overhead; therefore, they may be slower.
- Function-like macros may increase the code size generated when the file is compiled.
- Parameters in a macro may have unexpected side effects.

An example of a macro with an unexpected side effect is:

```
#define MAX( a, b )  (((a)>(b))?(a):(b))
```

The macro MAX is expanded. It can produce incorrect results if its parameters are used incorrectly:

```
num1 = 21;
num2 = 1;
result = MAX( num1++, num2 );
```

This macro function produces unexpected results because the first parameter is incremented each time it is used in the expression to which MAX expands. The macro expansion of the above code is:

```
num1 = 21;
num2 = 1;
result = (((num1++)>(num2))?(num1++):(num2));
```

The expanded code increments num1 twice, producing an unexpected answer.

#include

The `#include` directive replaces the specified line with a specified file. The format of this directive is:

```
#include "filename"
```

or

```
#include <filename>
```

The directory search order for `#include` files enclosed in double quotes (" ") is:

1. The directory of the file that contains the `#include` directive. (The `-I-` compiler option suppresses this search.)
2. The directories specified by the `-I` command line option.
3. The system include file directory, /usr/include.

The directory search order for `#include` files delimited by angle brackets (< >) is:

1. The directories specified by the `-I` command line option
2. The system include file directory, /usr/include.

#undef

The `#undef` directive removes any previous definitions for *name*. The syntax of the `#undef` directive is:

```
#undef name
```

where *name* is an identifier that has been defined using the `#define` preprocessor directive or the `-D` command-line option.

This directive is often used to ensure that a routine is really a function and not a macro.

Macro operators

Two operators are useful in defining a macro: # and ##.

The # operator converts a parameter to a string. This operator can be used only with formal arguments in the definition of a function-like macro. It converts the parameter to a string literal, as though the parameter was enclosed in double quotes. If the actual parameter contains white space, only the white space embedded within text is significant, and contiguous white space is reduced to one blank. Further, if the actual parameter contains symbols that are normally preceded by a backslash in a string literal, these symbols are inserted automatically into the resulting string literal.

For example, you could use the following function-like macro to open a file:

```
#define file_open( filename )\
    fopen( #filename, "r" )
```

You could then use this macro function in the following manner:

```
FILE *fp;
fp = file_open( myfile );
```

The preprocessor expands this macro into the following lines of code:

```
FILE *fp;
fp = fopen( "myfile", "r" );
```

A second operator that you can use in macro definitions is ##, the token pasting operator. It creates one token from two tokens. For example, you may use it to perform operations on variables that are spelled similarly:

```
#define print_var( num ) \
    printf("var" #num " = %d\n", var##num )

int var1, var2, var3, var4;
print_var( 3 );
```

The macro expansion results in the following lines of code:

```
int var1, var2, var3, var4;
printf("var" "3" " = %d\n", var3 );
```

Conditional compilation

You can use many directives in conditional compilation. They are similar to their counterparts in C with one primary exception: the expression that the preprocessor evaluates must compute to a constant *prior* to program execution. Consequently, symbols in an expression are limited to macro names and integers.

The syntax of the `if` directive is:

```
#if expression
```

where *expression* is a combination of expressions constructed with C operators such as `&&`, `||`, and `!`.

The preprocessor provides and defines an operator which returns `1` if its operand has an existing macro definition. Macros are defined with the `#define` preprocessor directive or with the `-D` command-line option. Refer to Chapter 2, "Compiler fundamentals," for more information on the `-D` option.

If *expression* evaluates to a nonzero number, source code between `#if` and a following `#elif`, `#else`, or `#endif` directive remains in the source file. Below are explanations for these directives:

- The `#else` directive is the preprocessor counterpart to the `else` keyword.
- The `#elif` directive is the preprocessor counterpart to the `else if` keyword combination.
- The `#endif` directive terminates the `#if` directive.

For example, the following conditional code compiles different source code for different execution environments:

```
#define CONVEX 1
#define OTHER_COMPUTER 0
assert(!(CONVEX !=0 && OTHER_COMPUTER != 0));

#if CONVEX
   long long j;
#elif OTHER_COMPUTER
   long j;
#endif
```

This code uses extensions to C, depending on the architecture of the execution environment. It does not execute the `assert` function until the program is run, while the conditional compilation occurs before runtime. If both `CONVEX` and `OTHER_COMPUTER` macro constants are nonzero, it prints a

diagnostic message and the program halts. (Refer to the assert(3) man page for more information on this function-like macro.)

Two other conditional compilation directives are `#ifdef` and `#ifndef`. These directives are specialized forms that test whether a name is defined. For example:

```
#ifdef Exemplar
#include <spp_prog_model.h>
#endif

#ifndef CONVEX
#define CONVEX
#endif
```

#pragma

Pragmas provide additional information to the compiler; they also instruct the compiler to override conditions that automatically control vectorization or parallelization. Appendix B, "Pragmas and directives," describes pragmas recognized by the compiler.

#error

The syntax of this directive is:

```
#error token token ...
```

where *token* can be a preprocessor macro or text. This directive displays an error message when the preprocessor is executing.

Instead of using the `assert` function, you can use the following code:

```
#if OTHER_COMPUTER != 0 && CONVEX != 0
#    error "illegal environment"
#endif
```

This code has the advantage of producing error messages during compilation time instead of execution time.

`#line`

The syntax of this directive is:

```
#line number [filename]
```

The `#line` directive associates a different line number and, optionally, a file name with a source file.

This directive is useful when the source file has been generated by another program, such as a translator for another language. You can use this directive to associate errors in the C source file with lines in the Fortran source file.

The asm statement

9

The asm statement is a CONVEX extension that is available only in the extended and backward-compatible modes. The text of the asm instruction is machine-dependent.

Assembly-language statements

The CONVEX C compiler allows you to insert assembly-language statements into a C program using the asm statement. This statement has the form:

```
asm ("assembly_language_instruction");
```

The compiler turns off global optimization and global register allocation when compiling source files that contain asm statements. Therefore, you must isolate functions that use the asm statement from functions that can be optimized.

Use the asm statement to embed assembly-language statements in C code rather than writing an entire routine in assembly language. For example, on C Series computers:

```
void func(int interr[], int blast[])
{
   int i;
   asm ("dsi");   /* disable interrupts */
   for( i=0; i<10; i++ ){
      if( interr[i] )
         blast[i] = 1;
   }
   asm ("eni");   /* enable interrupts */
}
```

The available assembly-language instructions are described in the *CONVEX Compiler Utilities User's Guide* and the *CONVEX Architecture Reference Manual (C Series)*.

Data types and representations

This appendix describes data types that are supported by the CONVEX C compiler. These data types are listed below:

- Character
- Integer
- Enumeration
- Floating-point
- Pointer
- Void
- Union
- Structure
- Array

Data types that the compiler supports fall into three categories:

- those that ANSI C requires,
- those that are CONVEX extensions, and
- those that are accepted in the backward-compatible mode of the compiler.

Each discussion of a data type includes a description and a representation; examples clarify certain situations. The descriptions note data types that ANSI C compilers do not accept. For each internal data representation, numbers at the top of the figure represent bit numbers; numbers at the bottom represent byte offsets. The least significant bit in the data representation is numbered 0.

Integer types

The ANSI C compiler supports four integral data types: `char`, `short int`, `int`, and `long long int`. The bit length of each data type is shown in Table 15.

Table 15 Integral bit length

Data type	Length
`char`	8-bit integer
`short int` or `short`	16-bit integer
`long int` or `int` or `long`	32-bit integer
`long long int`	64-bit integer

By default, the compiler assumes that each data type is a signed value. Table 16 lists the range of numbers that these data types can represent

Table 16 Integral ranges

Data type	Signed range	Unsigned range
`char`	-128... 127	0 ... 255
`short int` or `short`	-32, 768 ... 32, 767	0 ... 65, 535
`long int` or `int` or `long`	-2,147,483,648 ... 2,147,483,647	0 ... 4,294,967,295
`long long int`	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807	0 ... 18,446,744,073,709,551,615

char and int

As ranges for the char data type imply, you can perform integer arithmetic on characters. The char representation contains a single character in the execution character set (ASCII).

Figure 11 illustrates the internal representation of the char data type.

Figure 11 char representation

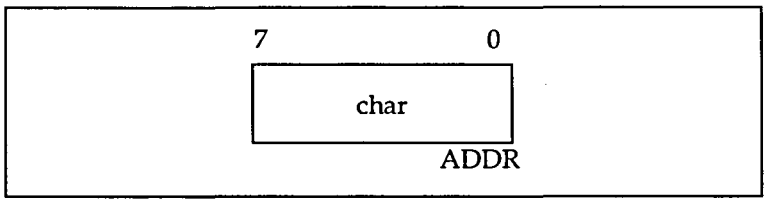
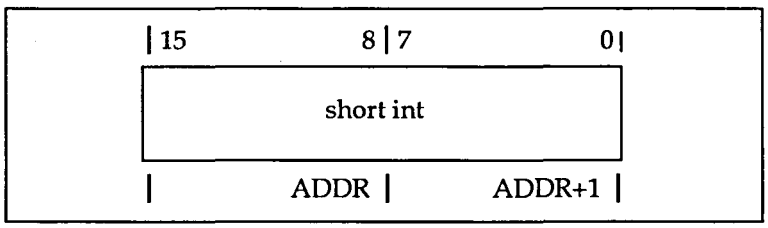


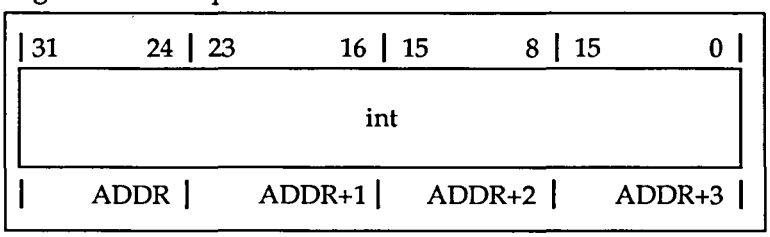
Figure 12 shows the internal representation of the short int data type.

Figure 12 short int representation



A 32-bit integer variable is declared as int (or long int) and can be unsigned or signed. Figure 13 shows the int data type representation.

Figure 13 int representation



long long int

This data type is an integer type that is implemented in 64 bits. The range for the `long long` data type is shown in Table 17.

Table 17 `long long` data type range

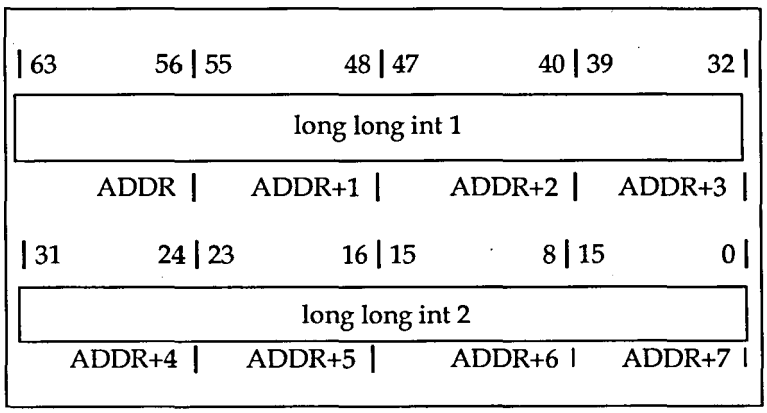
Format	`long long` or `long long int`
Signed	-9,233,372,036,854,775,808 ... 9,233,372,036,854,775,807
Unsigned	0 ... 18,446,744,073,551,615

Note

The `long long` data type is a CONVEX extension to the ANSI C standard. It should not be used by programs that will be ported to other compilers. This data type is accepted in the backward-compatible and extended modes of the compiler.

Figure 14 shows the `long long` data representation.

Figure 14 `long long` representation



Because 64-bit values are not truncated, improper stack alignment results when `long long int` values are passed to routines that expect `int` arguments. Therefore, in the backward-compatible mode, you cannot pass `long long` integers as arguments to functions that do not expect to receive them. This exception applies to `char` and `short` values (which can be passed to routines that expect `int` or `long` values).

CONVEX C converts 8- and 16-bit quantities to a 32-bit format before pushing them onto the runtime stack.

Enumerated types

An enumerated data type is a user-defined integral data type. Declare enumerated scalar data as `enum`. For example, you might declare the enumerated data type `color` as:

```
enum color { red, blue, green } hue;
```

In this example, the variable `hue` could be only one of the values `red`, `blue`, or `green` at any given time.

Internally, `enum` values are stored as integer representations. By default, the first enumerated value (`red` in the example above) is stored with the ordinal value of zero. Subsequent enumerated values are represented by sequential integer values. For instance, in the example above, the value of `blue` is 1 and `green` is 2.

Default ordinal values are overridden when they are followed by an equal sign and a new ordinal (for example: `enum color {red=10, blue=20,green=30}`). Because the `enum` data type is implemented as a `signed int`, the range of ordinal values available for use is the same as that used by a `signed int` in Table 16.

Floating-point types

The ANSI C compiler supports three floating-point data types: `float`, `double`, and `long double`. The bit length of each of these data types is listed in Table 18.

Table 18 Floating-point bit length

Data type	Length
`float`	32-bit floating-point
`double` and `long double`	64-bit floating-point

You can represent floating-point data in either CONVEX native format or IEEE format, although CONVEX native format runs only on C Series machines. To process floating-point data in IEEE mode, you must specify the correct option and your machine must have IEEE support hardware. Refer to Chapter 2, "Compiler fundamentals," for more information on options.

Note

CONVEX C Series hardware and runtime libraries support only the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

CONVEX SPP Series systems are true IEEE machines.

Table 19 shows the positive range of float values for native and IEEE formats. Floating point values may also be negative.

Table 19 Native and IEEE floating-point ranges

Format	float
Native (C Series)	$2.9387359 \times 10^{+39}$ ... $1.7014117 \times 10^{+38}$
IEEE	$1.1754944 \times 10^{-38}$ ... $3.4028235 \times 10^{+38}$

Table 20 shows the positive range of `double` and `long double` values for native and IEEE formats. `double` values may also be negative.

Table 20 Native and IEEE floating-point ranges

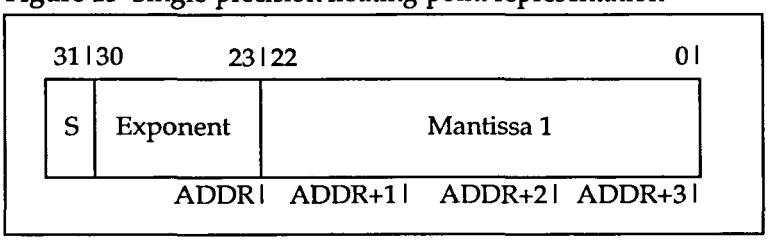
Format	double and long double
Native (C Series)	$5.562684646268003 \times 10^{-309}$ ... $8.988465674311584 \times 10^{+308}$
IEEE	$2.225073858507201 \times 10^{-308}$ ... $1.797693134862317 \times 10^{+308}$

Floating-point representation: `float`

The `float` keyword declares single-precision (32-bit) floating-point variables.

Positioning of the sign, exponent, and mantissa apply to native and IEEE formats; Figure 15 illustrates the internal representation of single-precision floating-point data.

Figure 15 Single-precision floating-point representation



C Series only

Single-precision native

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 128. That is, 128 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to

the left of bit position 22. The binary point is to the left of the implicit 1 bit.

Single-precision IEEE

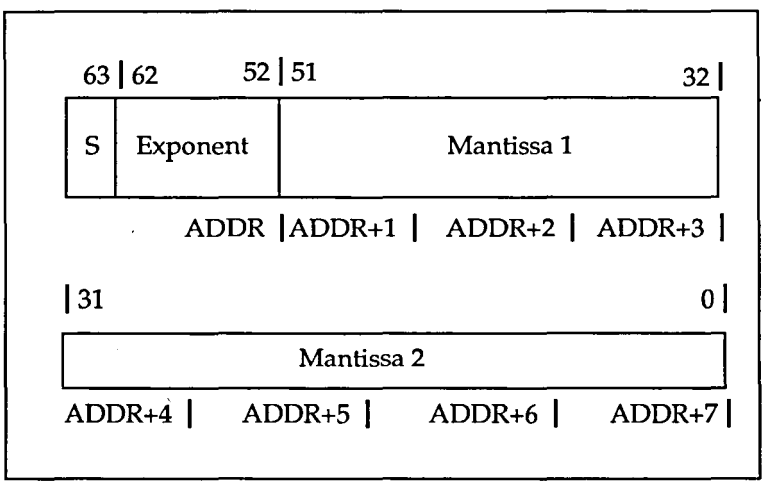
In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the right of the implicit 1 bit.

Floating-point type: `double`, `long double`

Double-precision (64-bit) floating-point variables are declared with the `double` or `long double` keyword and can be represented in either native format or IEEE format. To process floating-point data in IEEE mode, the machine must have IEEE support hardware.

Figure 16 shows the internal representation of double-precision floating-point data. The position of the sign, exponent, and mantissa apply to native and IEEE formats; particulars of each format are described in the following sections.

Figure 16 Double-precision floating representation



Double-precision native

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1024; that is, 1024 must be subtracted from the exponent to give the actual power of 2. The mantissa is

the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the left of the implicit 1 bit.

Double-precision IEEE

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the right of the implicit 1 bit.

Floating-point type: `long float`

Note

The `long float` data type is a floating-point type supported only by the backward-compatible mode of CONVEX C. It should not be used by programs that will be ported to other computer systems.

This type is synonymous with the `double` and `long double` data types. Table 21 shows the range for this type.

Table 21 Native and IEEE long float range

Format	long float
Native	$5{,}562684646268003 \times 10^{-309}$ ... $8.988465674311584 \times 10^{+307}$
IEEE	$2.225073858507201 \times 10^{-308}$ ... $1.797693134862317 \times 10^{+308}$

Pointer data type

A pointer is a variable that contains a 32-bit address. An asterisk is used to declare a pointer. For example, the declaration

```
char *cp;
```

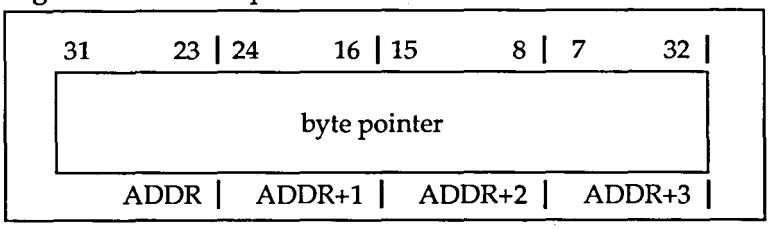
designates a pointer named *cp* to which you can assign the address of a `char` variable. All pointers defined in CONVEX C refer to the location of a byte in memory. Pointers have the same range of possible values as the `unsigned int` data type. Not all possible unsigned integer values, however, can be used as valid pointers.

While it is not an error for a pointer variable to contain the address of an invalid memory location, it is an error for the program to attempt to access the contents of the address to

which such a pointer refers. Also, it is an error for a program to reference an object pointed to by a NULL (0) pointer.

Figure 17 shows the data representation of a pointer.

Figure 17 Pointer representation



Word-aligned pointers have zeros as the two least-significant-bit positions; halfword-aligned pointers have a zero in the least-significant-bit position. Aligned addresses usually result in faster program execution because data with aligned addresses can be cached in high-speed memory by CONVEX C Series hardware. The compiler tries to keep addresses properly aligned.

Note

On CONVEX SPP Series systems, unaligned accesses cause a runtime exception.

`void` data type

The `void` data type does not represent an object in the way other data types do. Use this data type to specify the return type of a function that does not return a value and can also be used to discard a return value. This type has no data representation.

The `void *` data type is a generic data pointer that you can use to point to any data type. It is frequently used in function declarations when the actual return type or parameter is unknown. For example, the function declaration of `malloc` is:

```
extern void *malloc(size_t);
```

To obtain a pointer to a `char` object, the return type must be cast:

```
char *cp;
cp = (char *) malloc(4);
```

The representation of the `void *` data type is the same as that for a pointer (shown in Figure 17).

union data type

The union data type permits you to assign different data types to the same storage location. For example, a 32-bit integer can occupy the same memory location as a 32-bit floating-point number.

One common use of the union data type is to provide access to individual bytes in a variable. For example:

```
union {
   unsigned short word_a;
   struct {
      unsigned char hi_byte;
      unsigned char lo_byte;
   } a;
} reg;
```

You can manipulate high and low bytes as individual quantities. If reg.a.lo_byte is 255, adding 1 to its value does not affect the high byte. In contrast, adding 1 to the value of reg.word_a may change the value of both the high and the low bytes.

The representation of the union data type depends on the size of the largest member in the union. If the largest member of a union type is a long int, the union type occupies 4 bytes of storage. The address of each member of the union is the same as the address of the union type identifier.

struct data type and representation

A structure is a named collection of one or more variables (called members), possibly of different types, grouped together under a single name. The sections below discuss assignment of structures, structures in function calls and returns, and data storage and alignment.

Assignment of structures

ANSI C allows assigning of one structure to another via the "=" operator. Given the declarations

```
struct employee {
   char name[40];
   int age;
   char sex;
};
struct employee new_emp =
   {"john smith", 31, 'm'}, old_emp;
```

You can assign old_emp the values from new_emp with the single assignment statement

```
old_emp = new_emp;
```

rather than the three statements:

```
strcpy( &old_emp.name, &new_emp.name );
old_emp.age = new_emp.age;
old_emp.sex = new_emp.sex;
```

Structures in function calls and returns

ANSI C also supports the use of structures in function calls and returns. Structures are passed to functions by value. The entire structure is pushed onto the stack. If new_emp is passed to a function update_emp, 48 bytes are pushed on the stack. The extra three bytes maintain stack alignment for performance reasons.

You may also declare functions to return structure values. For example:

```
struct employee update_emp(struct employee);
new_emp = update_emp( old_emp );
```

Data storage and alignment

Data storage refers to the size of data types. *Data alignment* refers to the way a system or language aligns data structures in memory. Data type alignment and storage differences can cause problems when moving data between systems that have different alignment and storage schemes. These differences become apparent when data within a structure is exchanged between systems using files or inter-process communication.

Data type alignments

CONVEX SPP Series systems use a NATURAL (as opposed to WORD) alignment mode. That is, alignment of an object is related to the size of the object. Therefore, an `int` is aligned on a 2-byte boundary, a `float` on a 4-byte boundary, and so forth.

Note

On C Series systems, alignment for `double` and `long double` is on a 4-byte boundary. On SPP Series systems, alignment for `double` and `long double` is on an 8-byte boundary. All other data type alignments are the same on the two systems.

Structure alignment and padding

The alignment of members within a structure depends on the data types of the members.

An `int` member does not cross a 32-bit aligned boundary. However, this fact does not imply that all `int` members are aligned on 32-bit boundaries. For example, two 16-bit `int` members fit in the same 32-bit package.

Boundaries for members within structures are the same as the alignment values for variables on the runtime stack.

The list below summarizes how the compiler aligns and pads members of structures:

- Arrays are aligned on a boundary corresponding to their data type. For example, a float array within a `struct` is aligned on a 4-byte boundary.
- Structures and unions are aligned as required by the most restrictive member.
- Padding is done to a multiple of the alignment size.

Table 22 shows alignment for structure members.

Table 22 Alignment of structure members

Structure member	Alignment boundaries	
	C Series	SPP Series
`char`, `unsigned char`, and `signed char`	byte	byte
`short`, `unsigned short`, and `signed short`	even byte	2-byte
`int`, `unsigned int`, and `signed int`	4-byte	4-byte
`long`, `unsigned long`, and `signed long`	4-byte	4-byte
`long long`, `unsigned long long`, and `signed long long`	4-byte	8-byte
`float`	4-byte	4-byte
`double` and `long double`	4-byte	8-byte

Bit-field alignment

- Bit fields are allocated in blocks of size `int`. They begin at the high-order location of storage.
- Bit fields that span an `int` boundary are placed in the following `int` storage location.

- Bit fields of type `short`, `char`, and `long long` are also permitted in the extended and backward-compatible modes.
- An `int` bit field is treated as an `unsigned int` bit field on C Series machines. On CONVEX SPP Series systems, an `int` bit field is treated as an `signed int` bit field.
- The order of allocation of bit fields within an `int` is from the most-significant bit position to the least significant bit position.
- A bit field cannot straddle a word (32-bit) boundary, but can straddle a byte (8-bit) or half-word (16-bit) boundary.
- An enumerated data type is represented as a `signed int`.
- Non bit field members pack into unallocated portions of bit field blocks, provided alignment restrictions are met.

Array data type and representation

An array is an aggregation of data in which all items are of the same data type. The items are arranged in contiguous memory locations.

Arrays have between 1 and 14 dimensions. For example

```
int four_dim [4][5][6][7];
```

is a declaration of a four-dimensional array that contains 840 items. Arrays are stored in row-major order. All arrays are zero based: the first element of array `A` is `A[0]`.

Array addresses

Like more primitive data types, the address of an individual element in an array is specified using the `&` operator. For example, the address of the third item in a one-dimensional array named `eigen` is:

```
&eigen[2];
```

Consequently, you can specify the base address of the array as `&eigen[0]`. However, you can also specify the base address of the array as `eigen` (or `&eigen` in ANSI C).

When arrays are passed to a function, the array name specifies the array:

```
int some_array[12];          /* array */
void use_array( int [] );    /* function */
```

```
use_array( some_array );   /* pass array */
```

C does not check boundary values on arrays; you must ensure that you do not make illegal array references in `use_array`.

Pointers to arrays

As noted in the previous section, the name of the array is a constant pointer to the base address of an array. Unlike a pointer, the address of the array cannot be modified. For example, the statement

```
array_name += 5;
```

is incorrect. The name of an array can never be the recipient of an assignment.

However, modifying a pointer to an array is legal. If a pointer is declared as pointing to the same type of data that is contained in the array, that pointer can point to individual elements in the array, and you can make assignments to that pointer. If the pointer is declared to be a different type from the elements in the array, arithmetic operations with the pointer probably will not produce the desired results.

String representation

A special case of an array representation is the *string* data type. For example,

```
char filename[] = "main.c";
```

defines an array of type `char` that contains the file name `main.c`. Each byte in the array contains an ASCII character code. The NULL byte is automatically appended to the string. The NULL byte indicates the end of the string; it is expected by many system functions that have parameters that are strings.

If a string is created by hand, as in

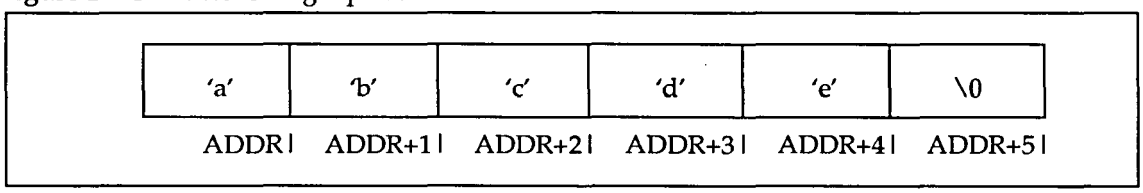
```
char filename[7];
filename[0] = 'm';
filename[1] = 'a';
filename[2] = 'i';
filename[3] = 'n';
filename[4] = '.';
filename[5] = 'c';
filename[6] = NULL;
```

then you *must* append the NULL character to the string by hand. This character is automatically appended only when arrays are initialized to a string or in some functions such as `strcat` and `strcpy`.

Single-character constants in C are delimited by apostrophes. ASCII codes are specified as `char` variables when you place the one- to three-digit octal number representing the desired character code preceded by a backslash (\) character between apostrophes, as in `'\64'`.

Arrays of character data are stored in ascending memory addresses, regardless of 32-bit word boundaries. Figure 18 shows the representation of the string "abcde" in memory.

Figure 18 Character string representation



Pragmas and directives

B

This appendix describes CONVEX C pragmas and compiler directives. Pragmas and compiler directives are functionally equivalent. The use of directives is being phased out; CONVEX C supports compiler directives for backward compatibility only. Use the `pragma` syntax instead of directives for new programs.

For additional information on SPP Series pragmas, see the *Exemplar Programming Guide*. For more information on C Series pragmas, refer to the *C Optimization Guide*.

Pragma syntax

A pragma of the form

```
#pragma _cnx pragma[, pragma]
```

is equivalent to the following compiler directive format

```
/*$dir pragma[, pragma]*/
```

Use commas to separate multiple pragmas specified on one line. When specifying a pragma, you can optionally specify a target machine for which the pragma will apply.

```
#pragma _cnx [SPP | CSERIES] pragma[, pragma]
```

Including either the `SPP` or `CSERIES` argument applies the pragma only when the code is compiled for the specified target machine. This option is useful if you are writing code that may be compiled on either architecture and you wish to include pragmas for both, while avoiding seeing compiler warning messages for those that do not apply to the current machine. If the target machine argument is not specified, the pragma applies in all cases.

For a description of each pragma, see the corresponding pragma entry in the next section.

Note

Pragmas are case-insensitive; they can be specified in uppercase, lowercase, or any combination. However, "`#pragma`" must be entirely in lowercase letters. "`_cnx`" must be entirely uppercase or entirely lowercase (either "`_cnx`" or "`_CNX`").

Vector-specific pragmas are ignored on CONVEX SPP Series systems.

Table 23 lists the pragmas available on CONVEX C Series and CONVEX SPP Series systems.

Table 23 Compiler pragmas available on C Series and SPP Series systems

C Series and SPP Series	C Series only	SPP Series only
`begin_tasks, next_task, end_tasks`	`force_parallel`	`block_loop`
`loop_private`	`force_parallel_ext`	`critical_section, end_critical_section`
`no_parallel`	`force_vector`	`loop_parallel`
`no_peel`	`max_trips`	`no_block_loop`
`no_promote_test`	`no_recurrence`	`no_loop_dependence`
`no_side_effects`	`no_vector`	`ordered_section, end_ordered_section`
`opt_level`	`prefer_parallel_ext`	`save_last`
`peel`	`prefer_vector`	
`peel_all`	`pstrip`	
`prefer_parallel`	`select`	
`promote_test`	`synch_parallel`	
`promote_test_all`	`vstrip`	
`returns_unique_pointer`		
`scalar`		
`task_private`		
`unroll`		
`unroll_and_jam, no_unroll_and_jam`		

Pragmas

Some pragmas listed here provide the compiler with information that it cannot deduce on its own, while others instruct the compiler to override default conditions that control optimization, vectorization, or parallelization. Remember the following guidelines when using pragmas:

- You may specify more than one pragma on each line.
- The compiler issues a diagnostic message for pragmas that it does not recognize.
- You can use pragmas in all the compatibility modes of the compiler.
- A pragma associated with a loop affects the loop that immediately follows the pragma. It does not affect nested loops.

The remaining sections in this appendix describe the pragmas. A pragma's format is shown when it has associated options.

`begin_tasks`, `next_task`, `end_tasks`

A *task* is a sequence of linear code that can be executed in parallel with other tasks. The `begin_tasks` pragma identifies a sequence of tasks for independent, parallel execution.

The `next_task` pragma precedes each task except the first.

The sequence of tasks ends with `end_tasks`.

The following code shows the use of the tasking pragmas:

```
#pragma _cnx begin_tasks
    statement
#pragma _cnx next_task
    statement
#pragma _cnx next_task
    statement
#pragma _cnx end_tasks
```

You can specify a maximum of 255 tasks between a `begin_tasks` and an `end_tasks` pragma.

`block_loop` (SPP Series only)

Indicates a specific loop to block, and optionally, the block factor that will be used in the compiler's internal computation of loop nest based data reuse. In absence of the `block_factor`

argument, this directive is useful for indicating which loop in a nest the compiler should block.

```
#pragma _cnx block_loop [(block_factor=n)]
```

critical_section, end_critical_section (SPP Series only)

critical_section defines the beginning of a code block in which only one thread can be executing at a time; end_critical_section defines the end of the block.

```
#pragma _cnx critical_section[(gate)]
    statement block
#pragma _cnx end_critical_section
```

The critical_section pragma optionally accepts *gate*, a gate variable, which can be used to differentiate between parallel tasks.

force_parallel (C Series only)

The force_parallel pragma is effective only if you specify the -O3 compiler option. The pragma tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma on a loop whether or not the loop contains calls, but it may not be safe to use it for a loop with calls.

Certain actual dependencies, such as from one scalar to another, cause the compiler to ignore the force_parallel pragma.

This pragma does not allow the compiler to interchange or distribute loops outside the force_parallel loop for vectorization. To enable those optimizations, use force_parallel_ext.

Caution

This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop containing true dependencies, you may not get correct results. Check answers generated by the parallelized code.

Using the force_parallel pragma with scalar or no_recurrence causes a warning. Using force_parallel and another parallelizing pragma in the same loop nest causes a warning also.

The following code shows how to use force_parallel:

```
#pragma _cnx force_parallel
for( i=0; i<n; i++ )
   sub(a, i);     /* compiled with -re; sub
                     contains no parallelization
                     inhibitors */
```

`force_parallel_ext` (C Series only)

The `force_parallel_ext` pragma is effective only if you specify the `-O3` compiler option. This pragma forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma on a loop whether or not the loop contains calls.

If you specify `force_parallel_ext` and `force_vector` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip mine loop.

Unlike `force_parallel`, the `force_parallel_ext` pragma allows the compiler to interchange outer loops for vectorization.

Caution

This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you use this pragma with `scalar` or `no_recurrence`, a warning is issued. Also, an error occurs when you use `force_parallel_ext` and another parallelizing pragma in the same loop nest.

`force_vector` (C Series only)

The `force_vector` pragma forces the compiler to vectorize the loop that follows, regardless of apparent recurrences. It is possible to use `force_vector` on a loop that the compiler would not fully vectorize without the pragma and get incorrect answers because the pragma causes the compiler to ignore dependencies. This pragma is effective only in code compiled at optimization level `-O2` or higher.

Use this pragma only with fully vectorizable loops. If you specify `force_vector` and `force_parallel_ext` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip mine loop.

Caution

This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the vectorized code.

If you use `force_vector` with `no_recurrence` or `scalar`, the compiler issues a warning. It also issues a warning when you try to use `force_vector` and another vectorizing pragma in the same loop nest.

`loop_parallel` (SPP Series only)

The `loop_parallel` pragma specifies that the compiler should run the immediately following loop in parallel. This pragma is effective only in code compiled at optimization level `-O3`.

Caution

This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the parallelized code by comparing them to the results generated by the non-parallelized code.

A loop marked with a `loop_parallel` pragma must have a known number of iterations at loop invocation time (runtime), which you must specify by setting the induction variable (`ivar`). It cannot vary as a function of the loop. For this reason, `loop_parallel` cannot apply to any loop that exits by abnormal termination. A `loop_parallel` loop will not be interchanged with any other loop in a loop nest.

This pragma differs from the `prefer_parallel` pragma in that `prefer_parallel` will parallelize the loop that follows only if it contains no loop-carried dependencies.

The format of the `loop_parallel` pragma is:

```
#pragma _cnx loop_parallel (ivar= var [,attribute-list])
```

where

`ivar=`*var* specifies the primary loop induction variable.

attribute-list contains one or more of the following attributes:

- `threads`—causes thread-level parallelism.
- `nodes`—causes node-level parallelism.
- `chunk_size=`*n*—divides the loop into chunks of *n* iterations and distributes the chunks round-robin to the processors. You can either set the `chunk_size` or choose `ordered`, but not both.

- `max_threads=`*m*—allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.
- `ordered`—causes ordered execution of the loop; provides no automatic synchronization. You can either choose `ordered` or set the `chunk_size`, but not both.
- `ordered, nodes`—causes ordered execution across hypernodes.
- `ordered, threads`—causes ordered execution across threads.
- `nodes, chunk_size=`*n*—causes node-level parallelism by chunks.
- `threads, chunk_size=`*m*—causes thread-level parallelism by chunks.
- `chunk_size= n, max_threads=`*m*—causes chunk parallelism on no more than *m* threads.
- `ordered, max_threads=`*m*—causes ordered parallelism on no more than *m* threads.
- `nodes, max_threads=`*m*—causes node-level parallelism on no more than *m* hypernodes.
- `threads, max_threads=`*m*—causes thread-level parallelism on no more than *m* threads.
- `ordered, nodes, max_threads=`*m*—causes ordered node-level parallelism on no more than *m* hypernodes.
- `ordered, threads, max_threads=`*m*—causes ordered thread-level parallelism on no more than *m* threads.
- `nodes, chunk_size= n, max_threads=`*m*— causes node-level parallelism by chunks of size *n* on no more than *m* hypernodes
- `threads, chunk_size= n, max_threads=`*m*— causes thread-level parallelism by chunks of size *n* on no more than *m* threads.

The sections below discuss and furnish examples of `ivar` and of the `chunk_size` and `ordered` attributes.

`ivar`

Since a loop marked with a `loop_parallel` pragma must have a known number of iterations at loop invocation time (runtime), the qualifier (`ivar=`*var*) specifies the induction loop variable:

```
#pragma _cnx spp loop_parallel (ivar=i)
for (i=0; i<n; i++) {
   a[i] = c[i] * sin(b[i]);
}
```

In C, secondary induction variables are sometimes included in `for` statements, as shown in the following example.

```
#pragma _CNX loop_parallel(ivar = i)
   for(i=j=0; i<n;i++,j+=2) {
      a[i] = ...;
      .
      .
      .
   }
}
```

Because of the `loop_parallel` directive, secondary induction variables will not be recognized, even when present. To manually parallelize this loop, you must remove `j` from the `for` statement and either privatize it and make it a function of `i`, or declare `j` to be shared (which is the default storage class) and increment it within a critical section inside the loop.

The following example demonstrates how to restructure the loop so that `j` is a valid secondary induction variable:

```
#pragma _CNX loop_parallel(ivar = i)
#pragma _CNX loop_private(j)
   for(i=0; i<n; i++) {
      j = 2*i
      a[i] = ...;
      .
      .
      .
   }
}
```

This method runs faster than placing `j` in a critical section because it requires no synchronization overhead, and the private copy of `j` used here can typically be more quickly accessed than a shared variable.

If you identify a variable with `ivar` which is *not* an induction variable, the compiler issues a warning and ignores the pragma.

chunk_size

Specifying an integer constant *n* for `chunk_size` forces the compiler to cyclically execute *n* or fewer iterations until it has executed all iterations. For example, the following code uses the `loop_parallel` pragma to distribute the loop's work among all threads in chunks of 8 iterations:

```
#pragma _cnx loop_parallel(ivar=i,chunk_size=8)
for (i=0; i<750; i++)
   a[i] = c[i] * sin(b[i]);
return(0);
```

By default, the work of the loop is split evenly among the threads in partitions of `chunk_size` iterations. In loops where the chunk size does not evenly divide among the threads, such as in the previous example, some threads execute one less iteration. In the example, the loop's 750 iterations are partitioned into 94 chunks; these chunks are distributed among the threads in 92 chunks of 8 iterations and 2 chunks of 7 iterations.

ordered

`loop_parallel` does not normally restrict the order in which the loop's iterations start or finish. If a loop's iterations must execute in order because of synchronization requirements, you should use the `ordered` quantifier.

Specifying the `ordered` qualifier forces the compiler to initiate the loop's iterations in order.

```
#pragma _cnx loop_parallel(ivar=i, ordered)
for (i=1; i<=40; i++)
   .
   .   /* Code containing ordered section */
   .
return(0);
```

While the `loop_parallel(ordered)` pragma guarantees starting order, it does not guarantee ending order and provides no synchronization. To guarantee proper synchronization of a loop, you must use ordered sections or synchronization functions.

Refer to the *Exemplar Programing Guide* for more information.

loop_private

The `loop_private` pragma declares a list of variables and/or arrays private to the immediately following loop. The compiler assumes that variables declared `loop_private` have no loop-carried dependencies. No starting or ending values can be assumed for these variables. This pragma is effective only in code compiled at optimization level `-O3`.

The `loop_private` pragma has the form

```
#pragma _cnx loop_private(varlist)
```

where

varlist is a list of scalar variables or arrays, separated by commas, that are to be private to the immediately following loop.

Only scalar variables and statically-sized arrays can be declared private. Dynamic, allocatable, and automatic arrays are not allowed. Structures are not allowed. Including primary induction variables (such as `for` loop indices) in *varlist* will yield wrong answers. Secondary induction variables must be `loop_private`, and they must be a function of the primary loop induction variable.

If a variable that appears in *varlist* is referenced in an iteration of the loop, it must have been assigned a value previously on that iteration of the loop. Values assigned outside the loop or in previous iterations will not be available.

If the variable is referenced after the loop, it must have been assigned a value after the loop. Values assigned inside the loop or before the loop are not available, unless the `save_last` directive is specified.

The following example uses the `loop_private` pragma.

```
#pragma _cnx prefer_parallel
#pragma _cnx loop_private(s)
for (i=0; i < n; i++){
   if (i > i_limit)
      s = 3.0;
   else if (i <= i_limit)
      s = 2.0;
   a[i] = s;
}
```

In this example, s must have a value for the assignment to a[i] at the end of the loop. Without the loop_private pragma, the compiler cannot tell that s is always assigned. It therefore assumes that the value of s from a previous iteration might be needed, and fails to parallelize the loop.

The presence of loop_private(s) tells the compiler to ignore the possible dependency on s, so that the prefer_parallel pragma can be honored and the loop can be parallelized.

max_trips (C Series only)

The max_trips pragma tells the compiler that the loop will execute no more than the specified number of times. This pragma is effective only in code compiled at optimization level -O3.

The format of this pragma is:

```
#pragma _cnx max_trips(n)
```

where

n is less than or equal to the vector register length of 128.

Use this pragma to prevent the compiler from strip mining the loop. Eliminating strip mining results in more efficient code generation when the maximum trip count is less than or equal to 128.

no_block_loop (SPP Series only)

The no_block_loop pragma informs the compiler to perform no blocking on the immediately following loop. Generally the compiler will attempt to automatically block a nested loop to achieve optimal cache data reuse between loop iterations.

```
#pragma _cnx no_block_loop
```

no_loop_dependence (SPP Series only)

The no_loop_dependence pragma instructs the compiler that it can safely ignore any potential loop-carried dependencies (LCDs) it detects in specified arrays. The no_loop_dependence pragma is effective only in code compiled at optimization level -O3. This pragma has the form

```
#pragma _cnx no_loop_dependence (namelist)
```

where

namelist is a list of arrays whose potential LCDs will be ignored by the compiler.

If *namelist* is not specified, the compiler assumes that there are no LCDs involving any of the loop's arrays.

no_parallel

The `no_parallel` pragma tells the compiler not to parallelize the loop immediately following the pragma. The pragma does not prevent vectorization of the loop (on C Series). This pragma is effective only in code compiled at optimization level `-O3`.

If `no_parallel` and `no_vector` both precede the same loop on a C Series machine, that loop will run in scalar mode.

You may perform loop transformations (such as blocking, interchange, and distribution) but not parallelization.

Use the `no_parallel` pragma when a loop is parallelizable but a short trip count makes it desirable to run it scalar.

no_peel

The `no_peel` pragma stops loop peeling, an optimization that can increase code size and compile time.

This pragma overrides command-line options (`-peel` and `-peelall`). This pragma is effective only in code compiled at optimization level `-O2` and higher. Refer to the *CONVEX C Optimization Guide* for more information.

no_promote_test

The `no_promote_test` pragma stops test promotion, an optimization that can increase code size and compile time.

This pragma prevents the compiler from applying test promotion to the loop that immediately follows. This pragma overrides the command-line options (`-ptst` and `-ptstall`). This pragma is effective only in code compiled at optimization level `-O2` and higher. Refer to the *CONVEX C Optimization Guide* for more information.

no_recurrence (C Series only)

The `no_recurrence` pragma instructs the compiler to disregard any recurrences in a loop. If nothing else impedes vectorization,

the compiler vectorizes the loop. This pragma is effective only in code compiled at optimization level `-O2` or higher.

The `no_recurrence` pragma does not affect recurrences caused by a nested `for` loop. You can, however, use the pragma on each loop in a nest to give the vectorizer maximum opportunity to improve the nest's performance.

When you use `no_recurrence` and the compiler finds a recurrence, the compiler breaks the recurrence by removing one or more dependencies of the cycle. In the following code, if `j` is positive, no recurrence exists.

```
#pragma _cnx no_recurrence
for( i=0; i<n; i++ )
   a[i] = a[i + j];
```

The compiler always accepts a `no_recurrence` pragma on an apparent recurrence involving an array element; the compiler always ignores a `no_recurrence` pragma on an apparent recurrence involving a scalar. In the latter case, the compiler knows that a recurrence exists.

Caution

Incorrect results can occur if you mistake a real recurrence for an apparent one. When in doubt, test vector results against scalar results to determine whether a recurrence is real or apparent.

For more information about recurrence, refer to the *CONVEX C Optimization Guide.*

`no_side_effects`

The `no_side_effects` pragma tells the compiler that the specified function does not modify the value of an argument or global variable, perform input or output, or call another subprogram. The pragma applies only to the specified function and not to calls that the function might make.

The format of this pragma is:

```
#pragma _cnx no_side_effects(func[,func])
```

where

func specifies a user-defined function.

This pragma allows the compiler to remove a function call during scalar optimization if the call occurs in an expression assigned to an unused scalar variable. The compiler removes the function call because the function has no side effects.

Such optimization opportunities usually arise after the compiler performs other optimizations. These opportunities rarely occur in the original source text.

Place the pragma before the call to the named function but after its declaration. If *func* has not been declared, its use in the pragma implies a declaration of `extern int` *func*`()`. The following code shows how to use this pragma.

```
int f1(int, int);
#pragma _cnx no_side_effects(f1)
x = y * f1(5, Z) - w;
```

A function call with no side effects is invariant with respect to a loop when:

- Its arguments do not vary within the loop and the function call can be moved out of the loop.
- It does not modify a nonlocal variable.
- It does not perform I/O.
- It does not call a function that has side effects.

`no_vector` (C Series only)

The `no_vector` pragma tells the compiler not to vectorize the loop immediately following the pragma. This pragma does not prevent parallelization.

If `no_parallel` and `no_vector` both precede the same loop, that loop will run in scalar mode.

`opt_level`

Use the `opt_level` pragma to force the optimization level to be the minimum setting of the command line and pragma argument.

```
#pragma _cnx opt_level([-no,-O0,-O1,-O2,-O3])
```

The effect of the `opt_level` pragma is to set the effective optimization level to the minimum of the optimization level specified on the command line and the level specified within the `opt_level` pragma. The `opt_level` pragma should appear at file scope. It remains in effect until the next `opt_level` pragma or the end of the file.

ordered_section, end_ordered_section (SPP Series only)

The ordered_section and end_ordered_section compiler pragmas force all threads to execute a designated section of code one thread at a time in iteration order. These pragmas are useful only in programs compiled at optimization level -O3 and must be used with loop_parallel loops.

The end_ordered_section pragma specifies the point where the ordered section ends. This pragma does not accept the *gate* parameter.

These pragmas have the following form:

```
#pragma _cnx ordered_section (gate)

#pragma _cnx end_ordered_section
```

where

gate specifies a gate variable to be used to control entry into the ordered section.

peel

The peel pragma increases the compiler's internal limit on loop peeling. Conditionally executed first and/or last iterations of a loop are moved to outside the loop body. This optimization can increase the size of your code.

peel_all

The peel_all pragma disregards the compiler's internal limit on loop peeling. This optimization can increase the size of your code substantially.

prefer_parallel

The prefer_parallel pragma tells the compiler to parallelize the loop immediately following the pragma only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler does not find any dependencies, it parallelizes the loop.

On C Series machines, this pragma prevents the compiler from interchanging and distributing loops outside the prefer_parallel loop for vectorization, whereas prefer_parallel_ext does not.

`prefer_parallel_ext` (C Series only)

The `prefer_parallel_ext` pragma tells the compiler to parallelize the loop immediately following the pragma only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler finds no dependencies, it parallelizes the loop.

This pragma allows the compiler to interchange loops outside the `prefer_parallel_ext` loop for vectorization. To vectorize a loop and parallelize the resulting strip mine loop, use `prefer_parallel_ext` and `prefer_vector` at optimization level `-O3`.

`prefer_vector` (C Series only)

The `prefer_vector` pragma tells the compiler to vectorize the loop immediately following the pragma only if it appears safe. The compiler checks for actual recurrences. If the compiler finds no recurrences, the compiler tries to interchange the loop so that it is the innermost loop and then it tries to vectorize the interchanged loop.

`promote_test`

The `promote_test` pragma increases the compiler's internal limit on test promotion. Test promotion replicates code. Using this optimization can increase compile time and the size of your code.

`promote_test_all`

The `promote_test_all` pragma disregards the compiler's internal limit on test promotion to its maximum value. Test promotion replicates code. Using this optimization can increase compile time and the size of your code substantially.

`pstrip` (C Series only)

The `pstrip` pragma tells the compiler to strip mine the parallel loop immediately following the pragma. The compiler strip mines the loop according to the strip mine length you specify. The format of this pragma is:

```
#pragma _cnx pstrip(integer_constant)
```

where

integer_constant specifies the strip mine length.

The default action of parallel strip mining combines loop iterations into groups of $n/(2*ep)$, where n is the actual loop trip count, and ep is the number of processors (specified with the `-ep` compiler option) when the number of processors is greater than one. If the number of expected processors is one, the number of loop iterations in a group is always one. To override the default, use `pstrip` to specify with *integer_constant* the number of iterations to group.

A single thread executes each group. Parallel strip mining occurs only at optimization level `-O3`. If you do not use `pstrip`, the compiler selects a default strip mine length appropriate for the architecture of the machine for which you are compiling. When the number of iterations is small (less than 32), a `pstrip` value of one usually gives the best results.

You cannot use `pstrip` with vector loops.

`returns_unique_pointer`

The `returns_unique_pointer` pragma tells the compiler that the pointer returned by a function points to a memory location that no other pointer references. This pragma aids the compiler's alias analysis and pointer tracking.

Use this pragma as shown in the following example:

```
#pragma _cnx returns_unique_pointer(bar)
foo=bar()
```

Function `bar` must return a pointer type. `bar` must return a different value every time it is called.

Caution

Using the `returns_unique_pointer` pragma on a function that does not return a unique value each time it is called can lead to incorrect alias analysis resulting in incorrect optimizations.

`save_last` (SPP Series only)

The `save_last` pragma specifies that all variables named on an associated `loop_private` pragma have their last value saved into a shared variable of the same name at loop termination. If `save_last` is not specified, then the values in privatized

variables or arrays are indeterminate at loop termination. The format is:

```
#pragma _cnx save_last
```

You can only rely on this pragma for variables that are unconditionally assigned on the final iteration of the loop.

scalar

The `scalar` pragma prevents the loop following the pragma from being vectorized or parallelized. This pragma does not prevent other loops from being vectorized or parallelized.

This pragma does not perform loop transformations with the exception of unrolling: it performs unrolling for C Series machines but prevents it for SPP Series.

Use the `scalar` pragma:

- when the loop's iteration count is too low to benefit from vectorization,
- when you must obtain numerical results identical to those of a scalar loop, or
- when you want to prevent the compiler from interchanging or distributing loops.

The results of a vectorized loop can differ from its scalar version. Floating-point sum and product reduction operators can give different answers when the hardware does not process the operands sequentially.

select (C Series only)

The `select` pragma tells the compiler to generate multiple versions of a loop that select runtime code based on specified trip, or iteration, counts. The compiler can generate up to four versions of a loop: scalar, vector, parallel, and parallel-vector. The format of this pragma is:

```
#pragma _cnx select(vtrip,ptrip,pvtrip)
```

where

vtrip specifies the trip count at which to select parallel execution for the loop following the pragma.

ptrip specifies the trip count at which to select parallel execution for the loop following the pragma.

pvtrip specifies the trip count at which to select parallel-vector execution for the loop following the pragma. Parallel-vector execution implies that the loop is vectorized and the resulting pragma strip mine loop is parallelized.

If you omit a trip count by using two adjacent commas, the compiler uses a default value. If you replace a trip count with an asterisk, the compiler does not generate code for the corresponding mode.

If the actual trip count is less than or equal to the smallest specified trip count in the pragma, the loop runs scalar. If the actual trip count is greater than the largest trip count, the loop runs in the mode of the largest trip count. For example, suppose you precede a loop with this statement:

```
#pragma _cnx select(10,4,200)
```

The loop runs scalar if the actual trip count is 1 to 4, and parallel if the trip count is 5 to 10. The loop runs vector if the trip count is 11 to 200, and parallel-vector if the trip count is greater than 200.

The statement

```
#pragma _cnx select(*,*,*)
```

causes the loop to run in scalar mode.

`synch_parallel` (C Series only)

The `synch_parallel` pragma is effective only if you specify the `-O3` compiler option. This pragma tells the compiler to generate code that executes the loop that follows in parallel. However, instead of ignoring dependencies, the compiler inserts synchronization code that causes the dependencies to be honored at runtime.

Without specific pragmas, the compiler vectorizes any dependency-free part of the loop; this normally produces superior results. However, if a loop contains much code that is conditionally executed, you might want to parallelize the loop with the `synch_parallel` pragma, particularly if all the dependencies are in seldom executed branches.

On a machine with four processors, the loop in the following code might run faster parallelized and synchronized than if it is

partially vectorized and the recurrence placed in a scalar, nonparallel loop:

```
float a[100], b[100], d[100], e[100], f[100];

int calc(int n, int m)
{
   int i;

   #pragma _cnx synch_parallel
   for(i=0; i<32; i++)
      if( a[i] < 0 ){
            a[i] = a[i] + b[i];
            d[i] = e[i] * f[i];
      }
}
```

task_private

The `task_private` pragma declares a list of variables and/or arrays private to the immediately following *task*, or sequence of code that can be executed in parallel with other tasks. In CONVEX C, you can define tasks using the `begin_tasks`, `next_task`, and `end_tasks` pragmas. The `task_private` pragma is effective only in code compiled at optimization level `-O3`.

The `task_private` pragma must immediately precede or appear on the same line as its corresponding `begin_tasks` pragma. The compiler assumes that variables declared `task_private` have no dependencies between the following tasks; therefore, you cannot assume a starting or ending value for the `task_private` variable within a task.

The `task_private` pragma has the form

```
#pragma _cnx task_private(varlist)
```

where

varlist is a list of variables or arrays, separated by commas, that are to be private to each following task.

The following tasks are defined by a `begin_tasks` pragma and one or more `next_task` pragmas. The scope of the `task_private` variables is terminated along with the task list when the compiler encounters an `end_tasks` pragma.

Only variables and statically-sized arrays can be declared private. Structures are not allowed. Including induction variables in *varlist* will yield wrong answers.

If a variable that appears in *varlist* is referenced within a task, you must have assigned it a value previously within that task. Values assigned outside the task list or in other tasks are not available unless you have saved them with the `save_last` pragma.

If the variable is referenced after the task list, you must have assigned it a value after the task list. Values assigned inside or before the task list are not available.

For more information about tasks in CONVEX C, see the "begin_tasks, next_task, end_tasks" section on page 171.

unroll

The `unroll` pragma is effective only if you specify the `-O2` or `-O3` compiler option. The `unroll` pragma reduces loop overhead by replicating the body of the loop following the pragma.

```
#pragma _cnx unroll [(unroll_factor=n)]
```

where, if the `unroll_factor` argument is included, its value *n* specifies number of times to replicate the body of the loop.

To be eligible for unrolling, a loop must contain no internal branching and have an iteration count that the compiler can determine. The compiler unrolls a loop completely only if it knows that the iteration count is less than five; otherwise, the compiler partially unrolls the loop.

For C Series machines, the compiler completely unrolls only loops that can be vectorized. It partially unrolls loops that cannot be vectorized.

unroll_and_jam and no_unroll_and_jam

The `unroll_and_jam` pragma enables the loop unroll and jam transformation for the immediately following loop. The goal is to exploit the use of registers and thus decrease the number of slower main memory accesses. This pragma is effective at optimization levels `-O2` and `-O3`. The `unroll_and_jam` pragma is available only on CONVEX SPP Series machines; it takes the following form:

```
#pragma _cnx unroll_and_jam[(unroll_factor=n)]
```

where, if the `unroll_factor` argument is included, its value *n* specifies the number of times to replicate the body of the loop.

The `no_unroll_and_jam` pragma disallows the unroll and jam transformation for the immediately following loop only. It also is available only on SPP Series machines and takes the form:

```
#pragma _cnx no_unroll_and_jam
```

By default the loop unroll and jam transformation is disabled. For more information see coverage of the `-uj`, `-nuj`, and `-ujn` compiler options, or refer to the *Exemplar Programming Guide*.

`vstrip` (C Series only)

The `vstrip` pragma tells the compiler to strip mine the vector loops immediately following the pragma. This pragma is especially useful for automatically parallelized vector loops (loops that are vectorized and run with the outer strip parallel).

The format of this pragma is:

```
#pragma _cnx vstrip(integer_constant)
```

where

integer_constant specifies the strip mine length.

Vector strip mining executes a loop in strips of 128 elements by default, and the parallel outer loop runs iterations of the vector loop in parallel. `vstrip` overrides the compiler default and specifies a different strip mine length. A shorter strip optimizes the iterations of the strip mine loop so that the loop can be effectively parallelized. To determine the approximate maximum strip mine length when the number of expected processors (specified with the `-ep` option) is more than one, the compiler uses the formula

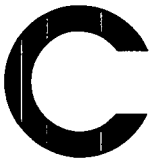
```
max(min(((n + ep - 1) / ep), 128), 8)
```

where

n is the actual loop trip count.

The actual strip mine length is the smaller of the number of iterations remaining to be processed or the maximum length of the strip determined with the formula (either the default or from the pragma).

Cray intrinsic functions



Introduction

A subset of Cray's bit intrinsic functions is available with this release of CONVEX C. Because these functions use an argument of type `long long`, they can only be used with the extended (default) and backward-compatible modes of the compiler. They are provided to increase the portability of programs from Cray Standard C to CONVEX C.

The compiler performs no domain or range checking on these functions.

Even though the functions are implemented using the same name, there are differences between the implementations. These differences include:

- **intrinsic function mapping**—Not all of the Cray bit intrinsic functions can be implemented using intrinsic instructions on a CONVEX computer. Some functions are implemented using function-like macros, while other are implemented using external functions.
- **`bint.h` include file**—You must declare the functions. The `bint.h` include file contains the function prototypes for each of the functions. The compiler uses this information to coerce arguments and to increase vectorization. bint.h is in /usr/include on C Series systems and /usr/convex/include on SPP Series systems.
- **`-lbint` command line option**—You must include `-lbint` on the `cc` command line so that the compiler can link the bit intrinsic functions with your program. The library that contains the bit intrinsic functions is /usr/lib/libbint.a on C Series systems and /usr/convex/lib/libbint.a on SPP Series systems.

The CONVEX C compiler accepts the following Cray functions:

- `_count`
- `_dshiftl`
- `_dshiftr`
- `_gbit`
- `_gbits`
- `_ldzero`
- `_leadz`
- `_mask`
- `_maskl`
- `_maskr`
- `_parity`
- `_pbit`
- `_pbits`
- `_popcnt`
- `_poppar`

Function descriptions

In each of the function descriptions below, the rightmost bit position is zero, so the sixth bit position is the seventh bit from the right.

`bint_t _count(bint_t` *target*`);`

Returns the number of "1" bits contained in *target*.

`bint_t _dshiftl(bint_t` *target*`, bint_t` *source*`, int` *length*`);;`

Replaces the *length* rightmost bits of *target* with the *length* leftmost bits of *source*. Returns the result.

`bint_t _dshiftr(bint_t source, bint_t` *target*`, int` *length*`);`

Replaces the *length* leftmost bits of *target* with the *length* rightmost bits of *source*. Returns the result.

`bint_t _gbit(bint_t` *target*`, int` *i*`);`

Returns the value in *target* of the bit located in the *i*th bit position from the right.

```
bint_t _gbits(bint_t target, int length, int i);
```

Returns the value in *target* consisting of *length* bits beginning with the *i*th bit position from the right. The bits are right-justified.

```
bint_t _ldzero(bint_t target);
```

```
bint_t _leadz(bint_t target);
```

These functions perform the same action. They return the number of leading zeros contained in *target*. Note that the argument is coerced to the `bint_t` data type, a `typedef` of `long long`. Consequently, the number of zeros may be inflated. Refer to the example at the end of this appendix.

```
bint_t _mask(bint_t length);
```

Returns a mask of *length* left-justified "1" bits if $0 \leq length \leq 63$. Otherwise, it returns a mask of 128-*length* right-justified "1" bits.

```
bint_t _maskl(bint_t length);
```

Returns a mask of *length* left-justified "1" bits.

```
bint_t _maskr(bint_t length);
```

Returns a mask of *length* right-justified "1" bits.

```
bint_t _parity(bint_t target);
```

Returns the parity of *target*: zero if it has an even number of "1" bits or one if it has an odd number of "1" bits.

```
bint_t _pbit(bint_t target, int i, bint_t source);
```

Returns the value of *target* in which the bit located in the *i*th bit position from the right is replaced by the least significant bit of *source*.

```
bint_t _pbits(bint_t target, int length,int i,
    bint_t source);
```

Returns the value of *target* in which *length* bits beginning with the *i*th bit position from the right are replaced by the value of the *length* least significant bits of *source*.

```
bint_t _popcnt(bint_t target);
```

Returns the number of "1" bits contained in *target*.

```
bint_t _poppar(bint_t target);
```

Returns the parity of *target*: zero if it has an even number of "1" bits or one if it has an odd number of "1" bits.

Example

The following example computes the population count, the leading zero count, and the parity count of a `char` data type.

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

typedef long long bint_t;

main()
{
   unsigned char arg = 0x30;
   bint_t retval;

   retval = _count( arg );
   (void) printf("pop count = %lld\n", retval );
   retval = _leadz( arg );
   (void) printf("leading 0's = %lld\n", retval-56);
   retval = _parity( arg );
   (void) printf("parity = %lld\n", retval );
}
```

Note that 56 is subtracted from the return value of the _leadz function. The number 56 is `(sizeof(long long) - sizeof(char))`. Subtracting 56 accounts for the extra zeros introduced by the coercion to the `bint_t` data type

Implementation-defined features

This chapter states the implementation-defined features of the strict compatibility (`-str`) and conforming (`-std`) modes of CONVEX C. Using these features may cause problems when a program is being ported to or from another computer system. When such a transfer takes place, implementation-defined features of the two compilers must be compared to determine what changes are required in the program.

Implementation-defined features in this chapter are organized according to the ISO/IEC 9899:1990(E) document *International Organization for Standardization and the International Electrotechnical Commission —Programming Languages C*, Appendix G.3, "Implementation-defined behavior."

All these features pertain only to the strict and conforming compatibility modes of CONVEX C, except where noted.

Translation

A file that contains one or more syntax errors will generate a message in the following form when it is compiled:

`cc: Error/Warning on line` *line* `of` *file*`:` *message*

where

- `cc` indicates a compiler warning or error.
- *line* is the line on which the error was detected.
- *file* is the name of the file that contains the error.
- *message* is the warning or error message that the syntax error generates.

Environment

The main function has three arguments:.

- *argc* contains the number of arguments on the program command line.
- *argv* is an array of string pointers each of which is an argument on the command line.
- *envp* is a pointer to an array of strings that constitute the environment of the program.

The last array element in *argv* and *envp* is a NULL pointer.

The interactive devices available to an executable program are in the /dev directory; file names are prefixed by tty, pty, and console.

Identifiers

Every character in an identifier with *internal linkage* (one that is not visible in another compilation unit) is significant.

An identifier with *external linkage* (one that is used in another compilation unit) has 255 significant initial characters.

Characters in an identifier with external linkage are case sensitive.

Characters

You should be aware of the following characteristics of characters:

- Source and execution character sets are ASCII characters. There are eight bits in a character in the execution character set.
- This version of CONVEX C does not include multibyte character encodings.
- Mapping of members of the source character set to members of the execution character set is one to one.
- The value of character constants containing undefined escape sequences is the same value as the character constant without the back slash. For example, "\c" is the same as "c". Also, the value of "~", "@", "$", and any control character within single quotes is the ASCII value for that character.

- If a character constant contains more than one character, the compiler obtains its value using the following pseudocode:

```
int value = 0;
while( more characters )
   value = ( value << 8 ) +
           ( value of next char );
```

- Because an integer occupies four bytes of memory, the value of a character constant that has more than four characters consists of only the last four characters (Table 24).

Table 24 Character constant representation

Constant	Representation
'a'	0x61
'abc'	0x616263
'abcdef'	0x63646566

The result has type `signed int`. In the extended and backward-compatible modes, the character constant can hold up to eight characters because `value` can be type `long long int`.

- CONVEX C defines no locale-specific implementations.
- The international code set (multi-byte characters) is not implemented.
- A `char` type has the same range of values as a `signed char` type.

Integers

Integers are represented in two's complement form. Ranges for the integral types are enumerated in the "Integer types" section on page 154.

Converting an integer to a shorter signed integer is the same as transferring the low-order *N* bits of the source integer, where *N* is equal to 8 * `sizeof`(*destination*). Converting from an unsigned quantity to a signed quantity of the same length does not change the representation; the most significant bit of the unsigned

quantity is interpreted as the sign bit. Some of these conversions are shown in Table 25.

Table 25 Integer conversions

int	cast to short
-2147483647	1
65535	-1
unsigned short	**cast to short**
65535	-1
32768	-32768

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

For CONVEX C Series:

- A bitwise operation on a signed integer has the same result as a bitwise operation on an unsigned integer of the same value.
- A right shift of a negative signed integral type results in a positive value; sign extension does not occur.

For CONVEX SPP Series systems, when the shift operand of a right shift is signed, the shift is an arithmetic shift.

Floating-point numbers

Appendix A, "Data types and representations," describes the data types and ranges of floating-point numbers.

When an integer is converted to a floating-point number, it is rounded to the nearest value that can be represented. When a number of type `float` is converted to a number of type `double`, the mantissa is zero-extended. Such a conversion does not increase the accuracy of the value converted.

When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest value that can be represented.

Arrays and pointers

The type of integer required to hold the maximum size of an array is `size_t`, defined as an `unsigned int`.

Pointers and `int` integers can be cast to each other without any changes to the underlying bit pattern.

The type of integer required to hold the difference between two pointers to elements of the same array is `ptrdiff_t` defined as an `int`.

Registers

The `register` storage class has no effect on register use except in the presence of `asm` statements.

Structures, unions, enumerations, and bit fields

If a member of a union object is accessed after a value has been stored in a different member of the object, the value of the member depends upon the types of its members.

For example, given:

```
union {
    short snum;
    int  inum;
} num;
```

The `short` representation overlays the third and fourth bytes of the `int` representation because members in a union have the same address. Consequently, the first byte of a `short` overlays the third byte of an `int`.

Similarly, if a union object contains two members, a `float` and a `char`, the `char` overlays the exponent and sign bit of the `float`. Refer to Appendix A, "Data types and representations," for the representations of data types.

Alignment for members of structures is as follows:

- The alignment of arrays is dictated by the alignment of each element in the array.
- Structures and unions are aligned as required by the most restrictive member.

Table 26 shows the alignments for members of structures.

Table 26 Alignments for members of structures

Members of structures	Alignment	
	C Series	SPP Series
`char`, `unsigned char`, and `signed char` types	byte	byte
`short`, `unsigned short`, and `signed short` types	even byte	2-byte
`int`, `unsigned int`, and `signed int` types	4-byte	4-byte
`long`, `unsigned long`, and `signed long` types	4-byte	4-byte
`long long`, `unsigned long long`, and `signed long long` types	4-byte	8-byte
`float`	4-byte	4-byte
`double` and `long double` types	4-byte	8-byte

Bit fields are allocated in blocks of size `int`. They begin at the high-order location of storage. Bit fields that span an `int` boundary are placed in the following `int` storage location. The extended compatibility mode permits bit fields to have `char`, `short`, and `long long` data types.

An `int` bit field is treated as an `unsigned int` bit field on C Series machines. On CONVEX SPP Series systems, an `int` bit field is treated as an `signed int` bit field.

The order of allocation of bit fields within an `int` is from the most-significant bit position to the least significant bit position.

A bit field cannot straddle a word (32-bit) boundary, but can straddle a byte (8-bit) or half-word (16-bit) boundary.

An enumerated data type is represented as a `signed int`.

Qualifiers

Any object that has a `volatile`-qualified type is accessed when its value is used or modified at any optimization level.

The volatile type does not address issues associated with unsynchronized access to shared data. To use shared data, you must provide a lock around a region, not just an access.

Declarators

The number of declarators that may modify an arithmetic, structure, or union type is at least 12.

Statements

The maximum number of `case` values in a `switch` statement is indefinite. Other limits in the compiler will be reached before any limitation imposed on the number of `case` values is reached.

Preprocessing directives

Character constants in conditional inclusion directives are unsigned.

The method used for locating a file in a `#include` directive depends on its delimiters. A file name in a `#include` directive is delimited by *double quotes* (`" "`) or *angle brackets* (`<>`).

The search order for a file delimited by *double quotes* is:

1. The current directory.
2. The directory specified by the `-I` command line option.
3. The system directory, /usr/include.

The search order for a file delimited by *angle brackets* is:

1. The directory specified by the `-I` command line option
2. The system directory, /usr/include.

Source file character sequences are mapped to the ASCII character set.

Appendix B, "Pragmas and directives," describes the behavior for each `#pragma` directive encountered by the compiler.

The system translations of `__DATE__` and `__TIME__` are always available.

Library functions

The following section describes library functions and any implementation-defined declarations and definitions.

The `NULL` macro expands to the integer value 0.

The diagnostic message displayed by the `assert` function is of the form:

`Assertion failed: <integer expression>, file <name>, line <number>`

After the `assert` macro prints the diagnostic message, the abort function executes, resulting in an IOT trap.

Character checks

Table 27 shows which characters cause the functions `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` to return a true value.

Table 27 Characters checked by `ctype.h` functions

Function	Returns true value for:
`isalnum`	Any letter or digit
`isalpha`	Any letter
`iscntrl`	Delete character or ASCII character with value <32
`islower`	Any lower case letter
`isprint`	ASCII code 32 through 126, inclusive

Math error return values

Table 28 indicates the values returned by math functions when a domain error occurs. (`HUGE_VAL` is defined in the `math.h` header file. It is the largest floating-point value. It may not be representable as a `float`.)

Table 28 Math function return values

Function	Domain error return value
`acos`	acos(1)
`asin`	asin(2)
`atan`	pi/2
`atan2`	pi/2
`cos`	cos(0)
`cosh`	`HUGE_VAL`
`log`	log(|x|)
`log10`	log10(|x|)
`pow`	pow(|x|)
`sin`	sin(1)
`sinh`	`-HUGE_VAL`
`sqrt`	sqrt(|x|)

- Mathematics functions do not set the integer expression `errno` to the macro `ERANGE` on underflow errors.
- When the `fmod` function has a second argument of zero, a domain error occurs and a zero is returned by the function.

Signals

Refer to the man page for signal(3c) and the `signal.h` include file for specifics on the semantics for each signal recognized by the `signal` function.

Note that these are different on C Series and SPP Series systems.

- When a signal occurs, future occurrences of that signal are blocked until you unblock the signal.
- The default handling is not reset if the SIGILL signal is received by a handler specified to the signal function.

Other

- The last line of a text stream does not require a terminating newline character.
- Space characters that are written to a text stream immediately before a newline character are not eliminated.
- No null characters are appended to the data written to a binary stream.
- When a file is opened in the append mode, the file position indicator is initially positioned at the end of the file.
- When a write occurs on a text stream, the associated file is not truncated beyond that point.

Buffering

CONVEX C supports unbuffered, fully buffered, and line-buffered file input and output. The `stdin`, `stdout`, and `stderr` streams are line buffered.

Two functions affect buffering of a file: `setbuf` and `setvbuf`. These functions replace the file input and output buffers that are automatically allocated by an application with an array. If the second argument of these functions is `NULL`, the input and output is unbuffered.

Files

- It is possible to have zero-length files.
- A file name may consist of 1 to 256 characters. The characters may not include the *slash* or the null character. File names dot (.) and double dot (..) have special meanings.
- If the `remove` function is executed on a file that is open, you can continue to read and write to that file until the file is explicitly closed. At that time, the file ceases to exist.
- If an attempt is made to rename a file to a file that already exists, the existing destination file is erased.

Printing and scanning

- The `%p` conversion specification in the `fprintf` function is used to display the address of a pointer. The address displayed is a hexadecimal number that does not have a `0x` prefix.
- When the conversion specification in the `fscanf` function is `%p`, a hexadecimal number is input. This number must not have a `0x` prefix.
- The '-' character in the scanlist of a `%[` conversion specifier has no special meaning. That is, a '-' character in the input is matched regardless of where the '-' appears in the scanlist.

Error flags

For information on error flags, please refer to the man pages. For C Series machines, refer to the intro(2) man page; for SPP Series machines, refer to the errno(2) man page.

Functions `calloc`, `malloc`, and `realloc` return a unique pointer if the size requested is zero. This pointer is not NULL.

The `abort` function closes open files and deletes temporary files. The `exit` function returns its argument if that argument is nonzero, `EXIT_SUCCESS`, or `EXIT_FAILURE`.

The `strerr` function uses an integer argument to select and return a string indicating a certain error. Therefore, you can use `strerr` to "translate" error numbers (such as those returned by the errno macro) into readable messages.

Environment

The set of environment names used by the `getenv` function includes:

```
PATH, HOME, TERM, SHELL, TERMCAP, EXINIT,
USER, PRINTER
```

You may also define other environment names. Similarly, it is possible for an environment to have no environment names.

System functions

You can use the `setenv` function to change the environment list used by `getenv`.

The `system` function uses the same arguments as the `sh` command. When `system` is executed, the current process waits until the shell has finished, then returns the exit status of the shell.

System time

The system keeps time and can provide you with the present time and other information in several different formats, as in the functions below.

`localtime` returns a pointer to the "broken-down time" (see `struct tm` below), correcting for the time zone and Daylight Savings Time.

`gmtime` converts directly to Greenwich Mean Time (GMT), which is the time UNIX uses. GMT is the time at Greenwich, England at any given time.

`timezone` returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich, England. If the second argument is 0, the function returns the standard name; otherwise, it returns the Daylight Savings Time version.

If the required name does not appear in a table built into the routine, the function returns the difference from GMT. For instance, in Afghanistan, `timezone(-(60*4+30),0)` is appropriate because it is 4:30 ahead of GMT; the function returns the string "`GMT+4:30`".

You can retrieve information about the system time through functions such as `clock` and `time`. The header file `#time.h` defines the following data types for time functions:

`size_t` The unsigned integer type of the result of the `sizeof` operator.

`clock_t` An unsigned long returned by functions such as `clock()`.

`time_t` An unsigned long returned by functions such as `time()`.

`struct tm` A structure which holds the components of "broken-down" time (Table 29).

Table 29 Components of `struct tm`

Name	Value	Range
`int tm_sec`	Number of seconds after the minute	0..61
`int tm_min`	Number of minutes after the hour	0..59
`int tm_hour`	Number of hours since midnight	0..23
`int tm_mday`	Day of the month	1..31
`int tm_mon`	Number of months since January	0..11
`int tm_year`	Number of years since 1900	
`int tm_wday`	Number of days since Sunday	0..6
`int tm_yday`	Number of days since January 1	0..365
`int tm_isdst`	Daylight savings time	

`tm_isdst` is a flag that indicates whether Daylight Savings Time is in effect:

- A positive value means that Daylight Savings Time is in effect.
- A value of zero means that Daylight Savings Time is not in effect.
- A negative value means that the status information is not available.

The `clock` function returns the approximate processor time that the currently-executing process has used: the *era*. To determine the number of seconds, divide `clock`'s return value with the value of the macro `CLOCKS_PER_SEC`. If clock cannot retrieve the processor time or cannot represent that value, it returns the value `(clock_t)-1`.

For more information about time-related data and functions, see the "time.h" section on page 140.

Error messages

This appendix describes error messages encountered when the cc or the lint programs process C source files. The first part of this appendix discusses the control of some error messages; the second part details the error messages and, in some cases, includes a short example that shows the cause of the error message.

Error message control

The compiler generates error messages specific to each compatibility mode. For example, the backward-compatible mode does not allow use of prototypes. Refer to Chapter 3, "Compatibility modes," for more information on compatibility modes.

Compiler diagnostic options

Many compiler options control the diagnostic output of the compiler. For example, `-w` suppresses all warning messages.

The `-d` option can convert a warning message to an error message. This conversion is available for only a certain number of diagnostic messages. The syntax for this option is:

```
-d name[={w|e}]
```

where

name	is the name of the diagnostic message,
`=w`	converts name to a warning message,
`=e`	converts name to an error message,

and no suffix after *name* suppresses the *name* diagnostic message.

Error messages prevent the creation of an executable program; warning messages do not. For example, to permit extra

characters after the `#endif` preprocessor directive, compile with:

```
-d pp_extra
```

or to generate an error message, use:

```
-d pp_extra=e
```

The default for this error is a warning message.

Note

Converting an error message to a warning message or removing it entirely can cause undefined behavior in the compiler. Use care when overriding error messages.

Messages that can be overridden

The diagnostic messages you can override are listed below, accompanied by a description of the condition that they control. If the condition exists, a message is generated only if the message is converted to a warning or error message. Either of these cases might be the default. *Removing a diagnostic message does not affect the behavior of the compiler; it merely prevents the message from being displayed.* Unless otherwise noted, all diagnostic options are available with both cc and lint. None of these diagnostic options are available with /lib/cpp.

`arg_ptr_qual`

Detects actual function parameters that do not have the same type qualifiers as those declared in the function prototype.

`arg_ptr_ref`

Detects actual function parameters that are not the same pointer type as those declared in the function prototype.

`assign_in_condition`

Detects assignment expressions in locations where conditional expressions are expected.

`char_cvt_truncates`

Indicates that a `char` data type is the target of a cast of an integer type larger than a `char` data type. This message is available only in lint.

`class_ignored`

Indicates that an explicit storage class is used to declare a `struct` tag, `union` tag, `enum` tag, or `enum` member.

`const_condition`

Detects constant expressions in locations where conditional expressions are expected such as `if` and `switch` expressions. This message is available only in lint.

`const_not_init`

Detects uninitialized constant variables.

`cvt_changes_sign`

Detects when an unsigned integer is assigned to a signed integer of the same size. This message is available only in lint.

`cvt_to_unsigned`

Detects when a signed integral type is assigned to an unsigned integral type. The unsigned integral type can be the same size or larger than the signed integral type. This message is available only in lint.

`division_by_zero`

Detects compile time division by zero.

`dollar_names`

Detects identifiers embedded with the `$` character.

`escape_range_sequence`

Detects when an escape sequence in an integer constant is greater than 0xff.

`eval_order`

Detects when an expression has an undefined evaluation order. This message is available only in lint.

`float_suffix`

Detects when the floating-point suffixes, d, D, f, F, l, or L, are used. This option has no effect in the strict and conforming compatibility modes.

`function_parameter`

Reports when a function is used as a function parameter; you can only use function pointers as a function parameter. This option is on by default in lint.

`function_pointer_cast`

Looks for non-function pointers that are assigned to function pointers. This option is available only in lint.

`hidden_arg`

Indicates that the parameter of a function is hidden by an identifier with the same name declared in the outermost block of that function. This diagnostic affects only the backward-compatible mode of the compiler.

`hidden_extern`

Indicates that a declaration using `extern` inside a function definition causes an identifier not in the block to be obscured. This diagnostic affects only the backward-compatible mode of the compiler.

`hides_outer`

Indicates that an identifier prevents access to an identifier of the same name in an enclosing block.

`implementation_defined=[w|e]`

Control messages about the use of language features that are implementation defined. By default, these messages are not generated. To generated these messages, use the following syntax:

```
-d implementation_defined=[w|e]
```

where

`e` generates errors

`w` generates warning messages

`implicit_decl`

Detects when an implicit declaration is used because a function has not been declared previously.

`incomplete_record`

Indicates that a `union` or a `struct` was declared without any members. This message is available only in lint.

`int_cvt_truncates`

Indicates that a `long long` variable is converted to a variable of type `int`. This message is available only in lint.

`integer_overflow`

Detects when an integer constant larger than 64 bits is used.

`long_long_suffix`

Detects when the integer literal suffixes `LL` or `ll` are used.

`misplaced_lint_directive`

Checks for a lint directive used in the wrong context. This message is available only in lint.

`neg_shift`

Checks for right operands of shift operators that are negative constants.

`negative_to_uns`

Indicates when a negative constant is assigned to an unsigned type.

`no_arg_type`

Checks for function arguments declared without a data type.

`no_external_declaration`

Detects when no declarations are accessible to other compilation units.

`non_int_bit_field`

Checks for types of bit fields other than `int` or `unsigned int`.

`nothing_declared`

Detects empty declarations such as `int`.

`null_effect_expression`

Detects intrinsic math function calls that have no effect. This option is available only with cc.

`pointer_alignment_efficiency`

Checks for operations that can result in inefficient memory alignment. This message is available only in lint.

`pp_argcount`

Indicates that the number of arguments in the actual argument list of a macro does not match the number of arguments in the formal argument list.

`pp_argsended`

Indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.

`pp_badstr`

Indicates incorrect use of `#`, the string operator that is used in macro definitions.

`pp_badtp`

Indicates that an invalid token paste was attempted—the result of combining the left and right operands of the `##` macro definition operator is not a valid token.

`pp_extra`

Indicates that a preprocessor directive has extra arguments.

`pp_idexpected`

Indicates that a `#ifdef` or `#ifndef` does not have a legal identifier as an argument, or a formal argument of a function-like macro is not a legal identifier.

`pp_line_range`

Indicates the argument for the `#line` preprocessor directive does not have a value between 1 and 32,767, inclusive.

`pp_macro_arg`

Indicates that two or more formal macro arguments have the same name.

`pp_macro_redefinition`

Indicates that the replacement list for a function-like macro is not the same as its original definition. The preprocessor uses the most recent replacement list.

`pp_macro_redefinition_cmdl`

This message indicates that a macro was redefined with a new definition on the command line. This message is available only in lint.

`pp_malformed_directive`

Indicates that the proper syntax for a preprocessor directive was not used.

`pp_old_dir`

Indicates that the comment style syntax for a compiler directive (pragma) was used.

`pp_parse`

Indicates that a `#if` directive has an invalid integer expression.

`pp_undef`

Indicates that the argument of the `#undef` directive is not permitted.

`pp_undef_cmdl`

Indicates that you attempted to undefine a macro that cannot be undefined on the command line. This message is available only in lint.

`pp_unrecognized_directive`

Detects a preprocessor directive that the preprocessor does not recognize.

`pp_unrecognized_pragma`

Detects a pragma that the compiler does not recognize.

`set_but_not_used`

Detects variables that are assigned a value but are not used. This message is available only in lint.

`shift_too_large`

Checks for right operands of shift operands that are too large. This option only examines operands that are constants. This message is available only in lint.

`short_cvt_truncates`

Indicates that an integral type larger than a `short` integer is being assigned to a `short` integer. This message is available only in lint.

`skip_to_char`

This diagnostic indicates that the compiler is skipping to a character to recover from an error.

`skip_to_eof`

This diagnostic indicates that the compiler must skip to the end of a file to recover from an error.

`strict_syntax`

Detects the lack of a semicolon after the last member in a `struct` or `union` declaration or the presence of a comma after the last enumeration constant in an `enum` declaration list.

`uns_compare_neg`

Indicates a comparison between a `short unsigned` integer and a negative constant. This message is available only in lint.

`uns_compare_zero`

Indicates a comparison between an unsigned integer and zero. This message is available only in lint.

`unsigned_suffix`

Detects use of the integer literal suffixes `U` or `u`. This option is not effective with the `=e` setting.

`varargs_on_proto`

Checks for use of lint directive `VARARGS` on a function that has a function prototype. Use the ellipsis operator instead. This message is available only in lint.

`varargs_too_large`

Reports an error condition when the lint `VARARGS` argument is greater than the number of formal arguments in the function it precedes. This message is available only in lint.

`void_pointer_cast`

Indicates that a pointer to `void` is involved in a cast operation. This message is available only in lint.

Error-message catalog

The error messages are listed in the following pages in alphabetical order.

Each message description includes:

- the message generated by the compiler,
- the message name, if it can be used with the `-d` compiler option, and
- a short explanation of the error.

cc: actual and formal point to different types (*arg #*)

Actual function parameters are parameters used in a function call, while formal function parameters are parameters declared in a function prototype or function definition. This error message occurs when the pointer declared in a function prototype is not compatible with the pointer passed to the function.

```
extern int func1(int *c);

main()
{
   int a;
   float *b;

   a = func1( b );
}
```

In this example, the actual parameter is a pointer to `float` and the formal parameter is a pointer to `int`.

If you must pass different pointer types to a function, declare the formal parameter type as a `void` pointer type as in the following example.

```
/* compile with cc -d arg_ptr_ref=e file.c */
extern int func1(void *c);

main()
{
   int a;
   int *b;

   a = func1( b );
}
```

A `void` pointer is a generic pointer compatible with any pointer type.

cc: ambiguous old form assignment operator: *operator* interpreted as *operator*

Some non-ANSI C compilers permit assignment operators of the form =*op*, where *op* could be one of `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`. Use of these operators can lead to ambiguous situations. For example,

```
x =- 7;
```

cc: could be either "assign -7 to x," or "decrement x by 7."

These operators are obsolete in ANSI C. The message indicates what assumption the compiler made when it encountered such an operator.

lint: arg #*arg* matching '*' must be 'int' not *type*

The `printf` function allows you to specify with an argument how many characters of a string to print. In the code below, the "`*`" character in the format specifies that the corresponding argument, `n`, supplies the field width:

```
printf("%*s", n, s);
```

The error occurs when the argument corresponding to `*` is a string rather than an integer. This error often occurs when programmers forget to insert the field width argument, as below:

```
printf("%.*s", s);
```

lint: arg #*arg* must be a pointer type

The `scanf` function returns pointers to objects, rather than the objects themselves. For instance, when scanning a character from stdin, `scanf` returns a pointer to type `char`. If the argument corresponding to the scanned character is not a pointer, as below, this error occurs.

```
scanf("%c",c);
```

cc: argument to sizeof() is an incomplete type

An *incomplete type* is a structure or union whose list of members has not been specified. Trying to evaluate the size of an incomplete type results in this error.

```
struct incomplete_type struct1;

main()
{
   int i;
   i=sizeof(struct1);
}
```

cc: argument *name* unused in function *func*

This message indicates that function *func* declares *name* as a function parameter, but does not use it in its function definition, as in the code below.

```
int func( int arg1, int arg2 )
{
   return (arg1);
}
```

Correct this error by removing the unused function parameter from the function definition. Be sure that all function prototypes for this function remain consistent with the function definition.

cc: argument '*arg*' is hidden by declaration in outermost block at line *line* of *file*

Message Name: `hidden_arg`

This message is only generated with source files compiled in the backward-compatible mode. It indicates that the parameter *arg* is hidden by an identifier with the same name declared in the outermost block of that function, as in the code below.

```
/* compile with
   cc -pcc -d hidden_arg=w file.c */

f( a )
int a;
{
   float a; /* Hides parameter a */
}
```

Correct this error by renaming one of the identifiers and all references to it.

---

cc: array declared with zero or negative number of elements.

When the code declares an array, the expression delimited by [and] must evaluate to a positive integral constant. If not, the compiler cannot allocate any storage for the array.

---

cc: array element type has unknown size

This message occurs when the code declares an array with an incomplete type. Incomplete types do not provide enough information to enable a compiler to determine their size:

```
struct some_s array[10];
```

This code generates this message if it has not declared a `struct` with the tag `some_s`. The compiler does not report this condition in backward-compatible mode.

---

cc: array has too many dimensions; limit is *limit*.

The code declared an array with too many dimensions. The maximum number of dimensions is *limit*.

---

cc: array is too big

The number of elements in the array exceeds the maximum size of an array. Replace the array with smaller arrays to correct this error.

cc: array of functions illegal.

An array of functions is illegal, but an array of pointers to functions is not. The syntax for an array of pointers to functions is

type (**array-name*[*num*])();

where

type	is the data type returned by the functions.
array-name	is the array name.
num	is the number of elements in the array.

The code below generates this message.

```
int *(*funcs[5])();
```

This example declares an array of five pointers to functions that each return a pointer to an integer.

cc: assignment operator occurs in conditional expression

Message Name: `assign_in_condition`

This message indicates that an assignment operation occurs in a context in which a conditional operation is expected. The following source code contains two examples that demonstrate this condition:

```
/* check with cc -d assign_in_condition=w
file.c */
main()
{
   int a,i;
   for(i=1; i=0; i++)    /* example 1 */
   /* null */;
      if(a = 3)          /* example 2 */
         return (a);
   return (1);
}
```

The first example detects a conditional operation in a `for` statement, and the second detects a conditional operation in an `if` statement. This may be intentional. If it is not, convert the assignment operator to the correct conditional operator.

cc: attempt to modify an object with const-qualified type

If the code qualifies an identifier with the `const` qualifier, it cannot assign that identifier a value unless it first initializes that identifier. The code below generates this message.

```
const int x;

int y;

x = y;
```

One way to initialize a variable with `const` qualified type is:

```
const int z = 5;
```

cc: automatic aggregates may not be initialized.

This message occurs only in the backward-compatible mode. An aggregate is a `struct` or `union` data type, and is automatic if its storage class is automatic. CONVEX C compilers prior to CONVEX C V4.0 did not permit automatic aggregates in block scope to be initialized. Consequently, they cannot be initialized in the backward-compatible mode.

cc: bad argument '*argument*' for *directive* directive

argument is illegal when used with the pragma (directive) indicated in the error message. Refer to Appendix B, "Pragmas and directives," for assistance with the pragma (directive).

cc: begin_tasks directive with no end_tasks

The optimization pragma `begin_tasks` tells the compiler to generate parallel code for the series of tasks that immediately follow. The `next_task` pragma marks the end of the preceding task and the start of another task. You must terminate all such sequences of tasks with the `end_tasks` pragma. These pragmas tell the compiler that certain nonloop sections of code can execute safely in parallel.

The following code fragment causes the compiler to generate this error message at optimization level `-O3`:

```
#pragma _CNX begin_tasks

task1();

#pragma _CNX next_task
task2();
```

Correct this error by appending the `end_tasks` pragma:

```
#pragma _CNX begin_tasks

task1();

#pragma _CNX next_task

task2();

#pragma _CNX end_tasks;
```

cc: a bit field must have type int or unsigned int

Message Name: `non_int_bit_field`

The compiler has detected a bit field that does not have type `int` or type `unsigned int`. The ANSI C standard permits only these two types in addition to `signed int`. The following code generates this error message:

```
/* Compile with
   cc -d non_int_bit_field=e file.c */

struct s {
   signed char d = 5;
};
```

cc: break statement not in do, for, switch, or while statement.

`break` statements may occur only within a `do`, `for`, `switch`, or `while` statement. The following code illustrates such a situation.

```
main()
{
   break;
   return(0);
}
```

This error is often caused by misplaced { and } braces.

---

cc: call error: formal and actual have different struct types.

This message occurs when a `struct` declared in a function prototype is not the same as the `struct` passed in a function call, as in the example below.

```
struct tag1 { int z; };

struct tag2 { char x; };

extern void f( struct tag1 formal );

int g() {
   struct tag2 actual;
   f( actual );
}
```

In this example, the record structure that declares `actual` is not the same as the record structure that declares `formal`. The tags must be the same; if the contents of `tag2` were an `int` instead of a `char`, the message would still be generated. One way to correct this error is to declare the actual parameter with a `typedef` name instead of a `struct` declaration:

```
struct tag1 { int z; };

typedef struct tag1 tag1_t;

extern void f( struct tag1 formal );

int g() {
   tag1_t actual;
   f( actual );
}
```

`typedef` names are synonyms for the types that define them.

cc: call error: incompatible formal and actual types.

This message occurs when the data types of the actual parameters of a function do not agree with the data types of the formal definition or declaration of the same function, as in the example below.

```
void func(int a);
main() {
   int *p;
   func(p);
}
```

In this example, the actual parameter is a pointer, while the formal parameter is an `int`. Similarly, if the actual parameter is a nonzero value and the formal parameter is a pointer, the message occurs. Correct the error by modifying the data type of the formal or actual function parameter.

cc: '*name*' can not be redefined with -D, ignored

This message indicates that you attempted to define a preexisting macro with the `-D` compiler directive. Preexisting macros are:

- `__LINE__`
- `__FILE__`
- `__DATE__`
- `__TIME__`
- `__STDC__`
- `defined`

cc: *macro* can not be redefined, ignored

This message indicates that a `#define` preprocessor directive has `__LINE__` or `__FILE__` as an argument. You cannot redefine these macro constants in any compatibility mode. Macro constants that cannot be redefined in the ANSI C compatibilities modes are `__DATE__`, `__TIME__`, and `__STDC__` (in the standard and strict modes).

cc: can not initialize array '*array*' with elements of unknown size.

This message occurs in the backward-compatible mode when an array, declared with an undefined `struct` or `union`, is initialized, as in the example below.

```
struct unknown_s a[10] = { 3 };
```

Because the code has not defined `unknown_s`, the compiler cannot initialize the array. Correct this error by defining the `struct` or removing the initialization.

---

cc: can not open file *name*

This message is only generated when you run the C preprocessor by itself with the `cpp` command. It indicates that the preprocessor was not able to open the output file included on the command line. This message can occur if you invoke the preprocessor in a directory for which you do not have write permission or you do not have write permission on the indicated file.

---

cc: can not use -> or . on type whose members are not defined.

This message occurs when the code declares the left operand of the `->` or `.` operator with an incomplete `struct` or `union`. A structure is incomplete if its members are not declared; the compiler does not know the memory requirements of the structure. The following code generates this message:

```
void func()
{
   struct incomplete_s *p;
   p->unknown = 5;
}
```

Correct this error by defining the members of incomplete structures.

---

cc: Can't close file "*source_file*" - can't continue.

lint creates an intermediate file when it processes your C source files. When lint is unable to close the intermediate file that it

creates, it generates this message. The intermediate file may have been corrupted by another system process. Contact your system manager when this error occurs.

---

cc: Can't find include file 'file_name'

The user-specified include file could not be found. Either it does not exist, the directory it is in was not specified with the `-I` option, or the permissions on the file or its containing directory are incorrect.

---

cc: Can't open file "*file_name*" - can't continue.

The compiler is unable to open file *file_name*. One cause of this error is that the compiler is unable to find a source file.

---

cc: Can't recover from previous errors

The compiler encountered too many errors to continue processing the source code. When the compiler reports an error, it assumes what the programmer intended before it continues compiling. If it assumes incorrectly, the compiler can report nonexistent errors. This *cascade effect* can prevent the compiler from checking the entire source file. Removing the initial error can eliminate subsequent errors.

---

lint: >>>>> C O M P I L E R E R R O R <<<<<
>>>>> See your system manager for help <<<<<
cc: Can't sort error file for listing (system command=*system-command*).

The compiler tried to call the system sort routine (usually bin/sort) and an error occurred. If this error message appears frequently, contact your system administrator.

---

cc: can't take the address of register variable '*var_name*'.

This message occurs when the code applies the address-of operator (`&`) to a variable that it has declared with the `register` storage class:

```
main()
```

```
{
   register int a;
   int *b;

   b = &a;
}
```

Correct this error by removing the `register` storage-class specifier from the variable declaration.

`cc: Can't write to file "`*file_name*`" - can't continue.`

The compiler is unable to open file *file_name* for writing. Check to make sure that you can write to

- the directory in which *file_name* resides and
- the *file_name* file.

`lint: >>>>> COMPILER ERROR <<<<<`
`>>>>> See your system manager for help <<<<<`
`cc: cannot compile; machine serial number mismatch`

The serial number of the CONVEX computer does not match the serial number embedded in the C compiler. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

`cc: Cannot open CXdb data file '`*name*`'.`

The compiler stores data used by the CONVEX visual debugger in the .CXdb directory, a subdirectory of the current directory. When this message occurs, check to make sure that the compiler has write permission in the .CXdb directory.

`cc: Cannot open CXdb data file directory.`

This message indicates that the compiler cannot open the .CXdb directory in the current directory. Check to make sure that you have write permission in your current directory and that your umask value is appropriate.

cc: case label not constant or constant expression.

In the construct `case expression:`, `expression` must be a constant expression.

cc: case statement not in switch statement.

The `case` statement can only occur within the compound statement of C `switch` statements. This message often indicates misplaced braces.

cc: character constant *name* is too long. Max length is *length*.

The maximum length of a character constant is *length* characters. For example, if the maximum length of a character constant is 8 characters,

```
char ch = 'abcdefgh';
```

is a legal declaration of a character constant, but

```
char ch2 = 'abcdefghi';
```

has too many characters in its initialization. In the strict and conforming modes, the maximum length of a character constant is 4 characters; in the other compatibility modes, the maximum length is 8 characters.

cc: Compile time division by zero detected

Message Name: `division_by_zero`

The compiler detects division by zero only when a constant with the value of zero appears in the denominator of a fraction. The following code produces a division-by-zero message.

```
/* Compile with
   cc -d division_by_zero=e file.c */

int x = 3/0;
```

cc: Compile time integer overflow detected

Message Name: `integer_overflow`

The compiler detects an integer overflow when an integral constant requires more than 64 bits of storage. The following code produces an integer overflow message.

```
/* compile with
   cc -d integer_overflow=e file.c */

char x = 0xfffffffffffffff1;
```

---

cc: >>>>> COMPILER ERROR <<<<< >>>>> See your system manager for help <<<<< cc: Compiler error on line *linenum* of *filename*.

An internal compiler error occurred. Contact your system manager or the CONVEX Technical Assistance Center (TAC). Refer to the contact(1) man page to learn how to contact the TAC.

filename is the name of a file used to create the C compiler; *linenum* is the line number on which the error occurred. If a line number and file name are generated before the colon in the last line, they are associated with your source code.

---

cc: const object *identifier* is not initialized

Message Name: `const_not_init`

When the code qualifies an identifier with `const`, it must initialize that identifier at the point that it declares the identifier; the code cannot assign a value after it has declared the identifier with `const`. The following code generates this message.

```
/* compile with
   cc -d const_not_init=w file.c */

main()
{
   const int x;
}
```

---

cc: constant in conditional context

Message Name: `const_condition`

This message indicates that an `if` expression or a `switch` expression results in a constant value. The code below generates this message:

```
/* check with lint -d const_condition=w
file.c */

main()
{
   int x;
   if( 4 ) /*null*/;
   switch( 4 ) /*null*/;
}
```

Conditional statements that depend on constant expressions can be simplified. The presence of these conditionals may indicate a logic error; alternatively, you may want to code these conditionals using preprocessor directives.

---

cc: continue statement not in do, for, or while statement.

`continue` statements can occur only within a `do`, `for`, or `while` statement. The code below generates this message.

```
main()
{
   continue;
   return(0);
}
```

Correct this error by deleting the `continue` statement.

---

cc: conversion from long long to int truncates

Message Name: `int_cvt_truncates`

This message occurs when the code assigns a value of type `long long` to an integer, as in the code below.

```
main()
{
  long long ll = 12345678901234567890;
  int i;

  i=ll;
  printf("i == %d\n",i);
}
```

cc: conversion from unsigned value to signed value of the same size may cause change of sign

Message Name: cvt_changes_sign

This message occurs when the code assigns an unsigned integer to a signed integer of the same size. The code below generates this message.

```
/* check with lint -d cvt_changes_sign=w
file.c */

main()
{
   unsigned h;
   signed j;
   j = h;
}
```

This assignment may result in a large positive number being converted into a negative number. Correct this error by changing the target integral type to an unsigned type.

This message may be useful in tracking down bugs that occur because of unintended sign changes. However, due to type conversion rules, the output of this diagnostic message may be rather large.

cc: conversion to character type truncates

Message Name: char_cvt_truncates

This message indicates that a char data type is the target of an integer that requires more storage than a char data type, possibly resulting in a truncation of the original value. The code below generates this message.

```
/* check with lint -d char_cvt_truncates=w
file.c */

char x;
unsigned int y=500;

x = y;
```

Correct this error by changing the target data type to a larger integral type.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

cc: conversion to integer type truncates

Message Name: `int_cvt_truncates`

This message indicates that your program has a `long long` variable that is being converted to a variable of type `int`. The following code produces this message.

```
/* check with lint -d int_cvt_truncates=w
file.c */

main()
{
   long long x = 1;
   int y;

   y = x;
}
```

This type of conversion has the potential for converting a large positive number into a negative number. Correct this error by increasing the size of the target integer.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

cc: conversion to short integer type truncates

Message Name: `short_cvt_truncates`

This message indicates that an integer that requires more bytes for its representation than a `short` integer is being assigned to a `short` integer. The following code produces this message.

```
/* check with lint -d short_cvt_truncates=w
file.c */

main()
{
   long h = 1;
```

```
    int i = 1;
    short j;

    j = i;
    j = h;
}
```

This type of conversion has the potential for converting a large positive number into a negative number. Correct this error by increasing the size of the target integer.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

cc: conversion to unsigned is undefined for negative values

Message Name: `cvt_to_unsigned`

This message occurs when the code assigns a signed integral type to an unsigned integral type. The unsigned integral type can be the same size or larger than the signed integral type. The code below generates this message.

```
/* check with lint -d cvt_to_unsigned=w
file.c */

main()
{
    signed src;
    unsigned targ1;
    unsigned long long targ2;
    targ1 = src;
    targ2 = src;
}
```

An assignment of this type converts a negative number to a positive number. Correct this error by changing the unsigned target integral type to a signed integral type.

This message may be useful in tracking down bugs that occur because of unintended sign changes. However, due to type conversion rules, the output of this diagnostic message may be rather large.

cc: Could not close CXdb data file '*name*'.

This message indicates that the compiler could not close the *name* CXdb data file. Check your file permissions and disk space.

cc: could not create metrics file *filename*

When you compile code with the `-metrics` option, the compiler creates a metrics file in the directory that holds the source file. If this error occurs, check your file permissions and disk space.

cc: Could not perform '*file-op*' operation on CXdb data file '*name*'.

This messages indicates that the compiler could not perform a `write` or `fseek` operation on the *name* CXdb data file. Check to make sure that you have sufficient disk space and that your data files are not corrupted.

C Series only

cc: CPU time limit exceeded

This message occurs when the compiler exceeds the maximum amount of CPU time permitted on your CONVEX system. To increase the amount of time permitted, use the `-tl` *time* option, where *time* is the new time limit in minutes. Determine the current CPU time limit for your system by entering the `limit` command. For further information, consult the setrlimit(2) man page.

cc: CXdb data file directory '*directory*' is a plain file.

This message occurs when the compiler is unable to open the *directory* subdirectory as a directory. One cause of this is that *directory* exists as a file instead of a directory. Correct this by renaming the *directory* file, and then recompile your program.

cc: decimal integer must range between 1 and 32767

A #line preprocessing directive has an argument greater than 32767:

```
#line 32768
```

cc: declaration contains two storage class keywords

This message indicates that a declaration specified more than one storage class:

```
static extern f( void );
main()
{
   register static int x = 5;
}
```

Five storage classes exist: `extern`, `static`, `auto`, `register`, and `typedef`. Using two or more of these in one declaration is contradictory.

cc: declaration of *identifier* conflicts with compiler support routine

This message indicates that your program contains an identifier that the compiler uses internally to implement features such as structure assignment. The following code produces this message.

```
int gen$bcopy;
main()
{
   struct {
      int x,y,z;
   } a,b;
   f( &a, &b );
   a = b;
}
```

Correct this error by renaming the identifier to an identifier not used internally. The compiler generates internal names containing the `$` character, so avoid using this character.

cc: declared formal parameter '*param*' is missing.

This message occurs in the following situation:

```
f()
int x;
{}
```

This definition is a function definition; it declares the parameter in the old C style. In this case, x does not exist in the function parameter list. The correct usage is:

```
f(x)
int x;
{}
```

---

cc: '*variable*' declared 'extern' within function may not be initialized

If the code declares a variable with the `extern` storage class, it cannot initialize that variable when the declaration appears in a function definition. The following code generates this error message.

```
int main(){

   extern int var = 4;
}
```

The code can initialize the `extern` variables (at file scope) only in the ANSI C modes.

---

cc: destination type is too small to represent ptr

This error occurs when you use a typecast operator to assign a pointer to an object whose size is smaller than that of a pointer:

```
char c, *s = "string";

c = (char) s;
```

---

cc: *directive name* directive ignored - loop or inner loop has exit

This message indicates that a loop to which you applied the *directive name* directive or pragma contains a non-sequential exit. Statements such as `return`, `break`, `goto` or a function call may cause a non-sequential exit from a loop. The following code produces this message.

```
#include <stdio.h>

main()
```

```
{
   int i,sum=0;
   int a[100];

   #pragma _CNX force_vector
   for(i=0; i<100; i++) {
      if( i == 20)
         break;
      sum += a[i];
   }
   printf("sum = %d\n", sum );
}
```

Correct this error by recoding the loop so that it does not depend on `return`, `break`, `goto`, or a function call to exit.

`cc:` *directive name* `directive ignored - switch statement`

This message indicates that a loop to which you applied the *directive name* directive or pragma contains a `switch` statement, as in the example below.

```
#include <stdio.h>

int main()
{
   int i,sum=0;
   int a[100];

   #pragma _CNX force_vector
   for(i=0; i<100; i++)
      switch( i ) {
         case 5: break;
         default: sum += a[i];
      }
   printf("sum = %d\n", sum );
}
```

The compiler cannot vectorize loops that contain `switch` statements. To correct this situation, consider replacing the `switch` statement with an `if` statement. You can convert the above example to:

```
#include <stdio.h>

int main()
```

```
{
   int i,sum=0;
   int a[100];

   #pragma _CNX force_vector
   for(i=0; i<100; i++)
      if( i != 5 )
         sum += a[i];
   printf("sum = %d\n", sum );
}
```

However, you cannot always convert `switch` statements to `if` statements.

cc: directives specified for loop are inconsistent - some ignored

This message occurs when you use pragmas or directives that are incompatible with each other. Refer to Appendix B, "Pragmas and directives," for a list of incompatible pragmas and directives.

cc: division by zero is possible at runtime.

This message indicates that the compiler has encountered a possible divide by zero operation when folding constants during optimization. The code below generates this message.

```
/* compile with cc -O1 file.c */
#include <stdio.h>
main()
{
   int i = -1;
   int j = 0;

   if( i )
      i = i/j;
   printf("i = %d\n", i );
}
```

This message only occurs at optimization level `-O1` or higher. To correct this error, you may have to trace back from the statement to determine which variable is causing the problem. In this example, initializing `j` to zero causes the divide by zero error message when `j` is an operand of the division operator.

cc: dollar sign character occurs in identifier *identifier*.

Message Name: `dollar_names`

Dollar-sign characters (`$`) are not permitted in identifiers in ANSI C source code, but they are allowed as a CONVEX extension. This message indicates a possible nonportable use of the dollar sign character. The code below generates this message:

```
/* compile with cc -d dollar_names=e file.c */
int heavy$;
```

cc: duplicate case label.

The compiler does not permit duplicate `case` labels in a `switch` statement. The code below generates this error message:

```
#define A 2
#define B 0x2

switch( variable ){
   case A:
   case B: ;
}
```

The preprocessor converts the two macros `A` and `B` to the same `case` label, causing the error message to be displayed.

cc: duplicate definition: label already defined.

The scope of a label is an entire function. This message indicates that the code has defined a label twice in one function.

cc: embedded begin_tasks directives disallowed

The following code fragment is illegal:

```
#pragma _CNX begin_tasks

task1();

#pragma _CNX next_task, begin_tasks
   taskA();
```

```
    #pragma _CNX next_task
    taskB();
#pragma _CNX end_tasks, next_task
task2();
#pragma _CNX end_tasks;
```

Embedded `begin_tasks` pragmas are not necessary because the embedded tasks are inherently parallel with the outer level of tasks. For example, `taskA` in the previous code fragment is independent of `task1`, so you do not need to embed it.

---

cc: end_tasks directive with no begin_tasks ignored

The code must precede an `end_tasks` pragma with a `begin_tasks` pragma. Otherwise, the preprocessor ignores the `end_tasks` pragma.

---

cc: enum constant must be in range of type 'int'

The data representation for the `enum` data type is the same as that for the `int` data type. Consequently, an `enum` constant must not exceed the range of an `int` data type.

---

cc: enum tag *tag_name* may not be used before declaring its members

This message results from using an incomplete `enum` type (for example, `enum x;`) before defining its members.

The following code produces this message if the members of `some_e` have not yet been declared:

```
enum some_e x;
```

---

cc: error '*diagnostic name*' can not be changed to warning message or suppressed.

The indicated message cannot be converted to a warning message or suppressed. For an example on the use of the `-d` compiler option, refer to the "Compiler diagnostic options" section on page 209.

lint: >>>>> COMPILER ERROR <<<<<
>>>>> See your system manager for help <<<<<
lint: error reading lint input file.

lint creates an intermediate file when it processes your C source files. When lint is unable to read the intermediate file that it creates, it generates this message. The intermediate file could have been corrupted by another system process. Contact your system manager when this error occurs.

---

lint: >>>>> COMPILER ERROR <<<<<
>>>>> See your system manager for help <<<<<
lint: error writing lint intermediate file.

This message indicates that lint was able to open the intermediate file, but encountered an error when writing to it. The intermediate file could have been corrupted by another system process. Contact your system manager when this error occurs.

---

cc: Escape sequence is out of range for character

Message Name: `escape_range_sequence`

This message occurs when an integer constant value exceeds `0xff`. The code below generates this error message

```
/* compile with
   cc -d escape_range_sequence=e file.c */
int c = '\xff1';
```

---

cc: expression has no side effect and value is not used

Message Name: `null_effect_expression`

This message occurs when math functions declared in the `math.h` include file return a value that is not used.

```
/* compile with
   cc -d null_effect_expression=e file.c */
#include <math.h>
main()
```

```
{
   sin(0.0);
}
```

cc: expression statement has no effect

The code contains a statement that does nothing. For instance, the statement

```
x;
```

has no effect on the variable x, other variables, input, or output.

cc: extern '*identifier*' is hidden when declared by line *line-num* of *file*

Message Name: `hidden_extern`

The compiler only generates this message in backward-compatible mode. It indicates that a declaration using `extern` inside a function declares an identifier that is not visible where it is declared.

```
/* compile with
cc -pcc -d hidden_extern=w \ file.c other.c */

extern int print_extern();
main()
{
   int f=10;
   {
      extern int f;
      printf("f = %d\n", f );
      f = 3;
      print_extern();
      printf("f = %d\n", f );
   }
   printf("f = %d\n", f );
}

/* other.c */
int f = 6;
int print_extern()
{
  printf("f in external function = %d\n", f );
}
```

The two source files in the example produce the following output when compiled in the backward-compatible mode:

```
f = 10
f in external function = 6
f = 3
f = 3
```

In backward-compatible mode, all `extern` declarations have file scope. Initializing variable `f` variable to 10 obscures all references to the external variable `f`. To correct this error, change the name of the local identifier.

`cc: extra tokens after` *name* `directive, ignored`

Message Name: `pp_extra`

This message indicates that the preprocessor found extra tokens in a line that contains a preprocessor directive. Possible causes include:

- `#undef` has more than one identifier when compiling in an ANSI C mode:

  ```
  # undef x extra
  ```

- The second argument of `#line` is not text enclosed in double quotes:

  ```
  # line 2 extra
  ```

- The `#line` preprocessor directive contains extra arguments following a legally formed second argument:

  ```
  # line 3 "text" extra
  ```

- `#if` or `#elif` has more than one argument expression:

  ```
  # if x==y a == b
  ```

- `#ifdef` or `#ifndef` has more than one argument expression when compiling in an ANSI C mode:

  ```
  # ifndef d == g h
  ```

- The `#include` preprocessor directive has additional arguments following `#file#` or `<file>`:

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

- The `#endif` preprocessing directive has any arguments when compiling in an ANSI C mode:

```
# endif comment
```

Correct these errors by correctly using the preprocessor directives. The corrections for each of the examples above are:

- `# undef x`
- `# line 2`
- `# line 3 "text"`
- `# if x==y`
- `# ifndef d`
- `# include <stdio.h>`
- `# endif /* comment */`

`cc: field size too large for field '`*member name*`'.`

The number of bits in a bit field cannot exceed the number of bits available in the data type used to declare the bit field, as in the code below:

```
unsigned field_1 : 41;
```

The maximum number of bits that can be allocated to `field_1` is 32 because the `unsigned` data type has a maximum of 32 bits.

cc: File size limit exceeded

This message occurs when a file that the compiler creates exceeds the maximum file size permitted on your system. For further information consult the man page for setrlimit(2). You can determine the current file size limit for your system by entering the `limit` command.

lint: flag '*flag*' appears twice in format

This message appears when a `printf` statement includes a repeated flag. For instance, the "-" flag specifies that the argument which follows it should be left-justified. However, in the code below, this flag appears twice:

```
printf("%--c\n",c);
```

cc: a floating point exception may occur here at runtime.

This message indicates that your code may cause a floating-point exception when it is executed. The following code is one example of this:

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

main()
{
   float y;
   int i = -1;

   if( abs(i) )
      y = sqrt(i);
   else
      y = 0.0;
   printf("y = %f\n", y );
}
```

If you compile this code at optimization level `-no` or `-O0`, the possible floating-point exception is not found: `sqrt(-1)`. When you run the executable generated by this code, it prints out:

```
 = 1.000000
```

It does not indicate errors. However, if you compile this code at optimization level `-O1` or higher, the compiler generates the error message because the compiler has folded `-1` into all occurrences of `i`, and then computed `sqrt(-1)` in an attempt to reduce the number of computations the compiler must make at runtime.

To correct the error, you may have to trace your code to see which values the compiler has folded; this trace enables you to determine what parts of your code may generate floating-point exceptions. Refer to Chapter 5, "CONVEX C intrinsics," for information on the relationship between intrinsic functions and floating-point exceptions.

---

cc: floating point literal *value* contains suffix

Message Name: `float_suffix`

This diagnostic option reports use of floating-point suffixes: `f`, `F`, `l`, `L`, `d`, and `D`.

```
/* check with lint -d float_suffix=w file.c */
float real = 4.5d;
```

The preceding example produces the message, but

```
/* compile with
   cc -d float_suffix=w -str file.c */
float real = 4.5d;
```

results in a syntax error.

---

cc: formal macro argument *arg1* and *arg2* are not distinct

A macro takes two arguments, both declared with the same label. This declaration results in the preprocessor regarding both arguments as the same variable:

```
#define test_macro (arg1, arg1)\
   #arg1 = 10
```

Since there is no reason for the user to pass the same argument to a macro twice, and since the presence of two arguments implies that they should be different, the preprocessor issues a warning.

cc: formal macro argument list is incomplete

This message indicates that the formal argument list of a macro definition does not terminate with a right parenthesis:

```
# define w(r,t,s
```

Correct this error by inserting a right parenthesis at the end of the formal argument list.

cc: formal '*parameter*' may not be initialized

Formal function parameters cannot be initialized.

```
int f(x)

int x = 3;
{
   return (1);
}
```

Correct this error by removing the initializer—parameters are always given an initial value in the function call.

cc: formal '*parameter*' may not have function type

Message Name: `function_parameter`

Functions cannot be parameters of a function. The following is illegal:

```
/* compile with
   cc -d function_parameter=e file.c */

int func1(void);
int func2( int func( void ) );
```

When this diagnostic is turned off or converted to a warning, the compiler interprets the declaration as being a pointer to a function instead of passing the function itself.

To legally pass a function, declare it as a pointer. The correct code is:

```
int func1( void );
int func2( int (*func1)(void) );
```

cc: formal parameter *parameter number* of function *function name* has no name

All parameters in a function definition must have a name.

```
int func(int param1, int )
{

}
```

In this example, the second parameter does not have a name. Correct this error by inserting an identifier for the missing function parameter. Function prototypes do not require parameter names.

```
int func( int, int );
```

cc: formal '*parameter*' was not explicitly declared

Message Name: no_arg_type

This message indicates that the code declares a formal argument without a data type.

```
/* compile with cc -d no_arg_type=e file.c */

void f(x)
{
   x = 4;
}
```

In such cases, the parameter implicitly has type `int`.

lint: format '*format*' can not convert type '*type*' (arg #*arg*)

The printf and scanf functions have a format string, containing conversion characters, and arguments to insert into that string. This error occurs when a conversion character does not match the corresponding argument, as in the statement below:

```
printf("%c", "string");
```

cc: function has incomplete return type

You cannot call a function that has an incomplete return type. Incomplete return types do not provide enough information to enable a compiler to determine the size of the data returned.

```
int main(){
   struct unknown_s func(void);
   (void) func();
}
```

In this example, `unknown_s` is a `struct` of unknown size. While calling a function with an incomplete return type is illegal, declaring one is not illegal.

cc: function *name* has return(e); and return;

This message indicates that a function has a `return` statement that contains an expression and a `return` statement that does not return an expression.

```
int far()
{
   return (1);
}
int main()
{
   if( far() )
      return (1);
   return;
}
```

The compiler detects the two different syntaxes of the `return` statement in the `main` function. Correct this error by using the same `return` syntax consistently.

cc: function *function name* is declared static but never defined

The scope of a `static` function is limited to the functions in the source file which defines it. This message indicates that a source file contains a `static` function declaration and a call to that function, but does not contain the definition of the `static` function. Correct the error by defining the function in the file or converting the function storage class to `extern` and defining the function in another file.

cc: function *identifier* may not be declared *storage_class* in block scope

ANSI C does not permit functions declared in block scope with the `auto`, `static`, or `register` storage classes.

```
void func()
{
   auto some_f();
}
```

cc: function *function name* may not be initialized

You cannot initialize functions. However, you can initialize a pointer to a function.

cc: function prototypes are not supported in this environment

The compiler does not support function prototypes in the backward-compatible mode (`-pcc`).

cc: function returning array illegal.

A function cannot return an array. For example,

```
int func( void )[];
```

is an illegal declaration of a function that returns an array of integers. However, functions can return the address of an array:

```
int ( *func( void ) )[];
```

returns a pointer to an array of integers.

cc: function returning function illegal.

An example that generates this message is:

```
int (func(void))(void);
```

This prototype declares a function that returns a function that returns an `int`, which is illegal.

The correct method is to return a pointer to a function:

```
int (*func(void))(void);
```

In these examples, function `func` and the function it returns have no parameters.

---

cc: function '*f*' declared with prototype may not have old-style formal declarations

This message indicates that the parameters of function *f* are declared with the prototype form and with the old style parameter declaration, as in:

```
int f(int x)

int x;
{
   return (1);
}
```

These two styles of parameter declarations cannot be mixed. To correct the error, delete the old-style parameter declarations.

---

cc: global declaration has no type or storage class specifier

This message indicates that a declaration in file scope did not specify a data type or a storage class. This type of declaration is permitted in the backward-compatible mode of the compiler, but ANSI C prohibits it.

```
a, b(), *c;

main()
{
   c = &a;
   a = b( c );
}
```

In the backward-compatible mode, `a` is interpreted as an `int`, `b` as a function returning an `int`, and `c` as a pointer to an `int`.

To bring this code into compliance with the ANSI C standard, a data type or storage class must be specified on all declarations excluding functions.

cc: greater than 255 tasking directives

The code can use a maximum of 255 `next_task` pragmas (directives) with each `begin_tasks` pragma (directive). Correct the error by dividing the pragmas (directives) into blocks of 255.

cc: '*identifier*' hides declaration at line *line-num* of *file*

Message Name: `hides_outer`

This message indicates that the declaration of an identifier hides a declaration of an identifier with the same name in an enclosing block or at file scope.

```
/* compile with cc -d hides_outer=w */

int a;

void f(void)
{
   int b;
   {
      int a; /* Hides decl. at file scope */
      int b; /* Hides decl. in enclosing
                block */
   }
}
```

This may indicate a flaw in the logic of your program. To correct this error, rename the identifier that obscures the identifier in the enclosing block or the identifier at file scope.

cc: identifier expected

This message occurs when a formal argument of a function-like macro is not a legal identifier, or an `#ifdef` or `#ifndef` does not have a legal identifier as an argument.

```
#define x(b,4

#ifdef 4
#endif
```

cc: ignoring dependence from '*identifier*' to '*identifier*' can cause generation of incorrect object code !!

This message occurs when the compiler detects a possible loop-carried dependency that the user has instructed the compiler to ignore with a pragma. The loop-carried dependency may be between the two variables, or it could be caused by a function call.

For more information on loop-carried dependencies and how they affect optimization, refer to the *CONVEX C Optimization Guide*. For Convex SPP Series systems, refer to the *Exemplar Programming Guide*.

cc: ignoring initializer on 'extern' declaration of '*identifier*'.

This message occurs only in the backward-compatible mode of the compiler. For example:

```
extern int i = 5;
```

In the backward-compatible mode, the compiler ignores the initialization of the variables declared with `extern`. Several solutions to this problem exist:

- Remove the external storage class specifier (convert to: `int i = 5;`).
- Don't compile the program in the backward-compatible mode.
- Use the external storage class specifier, but initialize the identifier in the body of a function.

cc: illegal -D macro definition

This message is only generated when the C preprocessor is run by itself with the `cpp` command. It indicates that the name used with the `-D` option is an identifier that C does not permit. The following command line generates this error because the name following the `-D` option contains a symbol, `#`, that cannot appear in a C identifier:

```
cpp -Dwer#re file.c
```

Correct this error by removing the illegal symbol.

cc: illegal -U undefinition, ignored

Message Name: pp_badflag

This message indicates that you attempted to undefine a macro that cannot be undefined. The macros that you cannot undefine are

- __LINE__
- __FILE__
- __DATE__
- __TIME__
- __STDC__ (in the standard and strict modes)
- defined

cc: illegal character in source (ignored)

This message occurs when a character not in the C character set is found in the source code. Two such characters are

- backquote "`"
- at sign "@"

cc: illegal indirection.

This message occurs when the code incorrectly applies the indirection operator, *. The operand of this operator can be:

- Pointer to a function
- Pointer to an object
- Pointer to a type

All other applications of this operator are illegal.

This operator can also evaluate subscript expressions. For example, the subscript expression

expression-1 [*expression-2*]

is equivalent to

*((*expression-1*) + (*expression-2*))*

Consequently, when the contents of the outermost parentheses do not evaluate to one of the three pointer types mentioned

previously, the compiler generates the `illegal indirection` message.

```
int i = 5;
i[3] = 4;
```

In this case, the operand of the * operator, which is used implicitly in the subscript expression, is `i + 3`. The expression `i + 3` is not a pointer, so an error occurs.

cc: illegal initialization.

Permitted initializations are as follows:

- Static variable with a constant expression
- Automatic variables with an expression
- Pointers with constant expressions that evaluate to a pointer of the correct type

All other initializations are illegal.

cc: illegal integer constant suffix

The following integer constant suffixes are available in all modes of the compiler:

- `L` or `l`: suffixes for `long` data types
- `U` or `u`: suffixes for `unsigned` data types

Combine `L` and `U` for unsigned long data types. The `LL` or `ll` integer suffixes are available only in the extended mode; these suffixes are used for the `long long` integral data types.

cc: illegal macro undefinition

This message occurs when the `#undef` preprocessor directive:

- Does not have an argument
- Has an argument that is not a legal identifier
- Has an argument that is either `__FILE__`, `__LINE__`, `__DATE__`, `__STDC__` (in standard and strict modes), or `__TIME__`

The following examples produce this error:

```
#undef
```

```
#undef 234

#undef __DATE__
```

Correct this error by using a legal argument of `#undef`.

---

cc: illegal pointer subtraction.

This message occurs when pointers of different types are subtracted from each other, as in the code below:

```
char *p;
int *q;
int i;

i = p - q;
```

You can eliminate this message by declaring both identifiers as pointers to the same type.

---

cc: illegal pointer/integer combination.

This message has several causes:

- Initializing pointer data types with expressions that are not pointer types
- Initializing nonpointer data types with expressions that are pointer types
- Combining integers and pointers in illegal operations

---

cc: illegal preprocessing directive syntax, skip to end of line

This message indicates that:

- You need to add parts of a preprocessor directive.
- A `#define` command is followed by white space.
- The text following `#define` does not specify an identifier.
- A `#include` preprocessor directive is not terminated by a `>`.
- The argument of the `#include` preprocessor directive could not be expanded into a legal form. Below are examples of non-expandable `#include` lines.

```
# line
```

```
# define

# define 345

# define !asdf

# include < stdio.h

# include something
```

Correct errors of this sort by using legally formatted preprocessor directives. You can correct the above examples with:

```
# line 24

# define blank

# define x345

# define asdf

# include <stdio.h>

# define something <string.h>

# include something
```

`cc: illegal storage class for function.`

The only storage classes that are available for function declarations are `static` and `extern`. A function that has a `static` storage class in its definition cannot be used by functions that are defined in other source files. The `extern` storage class is the default storage class for functions.

`cc: illegal storage class for parameter declaration.`

You can explicitly use the `register` storage class keyword only in a parameter declaration.

cc: illegal struct or union combination.

If the code assigns structures to each other, it must declare them with the same structure definition. The code below generates this message.

```
struct {int *r;} *p;
struct {int *r;} *q;

p = q;
```

The same definition declares identifiers p and q:

```
struct {int *r;} *p, *q;

p = q;
```

No message occurs. Similarly, if the code uses a structure tag, as below:

```
struct tag {int *r;};
struct tag *p;
struct tag *q;

p = q;
```

no message occurs because the same structure definition declares p and q.

cc: illegal type combination.

This message occurs when the code uses two types to declare an identifier, as in:

```
int float x;
```

The code can use only one data type to declare an identifier.

cc: illegal type for bit field.

If a bit field is not an `int` or `unsigned int`, a warning message occurs. For example, `float x:3;` can cause this message. However, integral types, such as `char` and `long` (and `long long` in the extended mode) are permitted.

cc: illegal type: function type can not be 'const/volatile' qualified

This message indicates that the code defines a function type with `typedef` and qualifies it with a `const` or `volatile` qualifier:

```
typedef int f();

volatile f x;
```

cc: illegal type: 'const' appears twice in declarator

This message occurs when the `const` qualifier appears twice in the declarator of a declaration.

```
int * const const x;
```

Correct this error by deleting the duplicate qualifier.

A qualifier can appear twice in a declaration:

```
const int * const x = 5;

/* const ptr to const int */
```

In this case, one qualifier occurs in the declarator, and one in the type specification of the declaration. Similarly, two different qualifiers can appear in the declarator:

```
int * volatile const x = 5;
```

cc: illegal type: 'volatile' appears twice in declarator

This message occurs when the `volatile` qualifier appears twice in the declarator of a declaration:

```
int * volatile volatile x;
```

Correct this error by deleting the duplicate qualifier.

A qualifier can appear twice in a declaration:

```
/* volatile ptr to volatile int */

volatile int * volatile x = 5;
```

In this case, one qualifier occurs in the declarator, and one in the type specification of the declaration. Similarly, two different qualifiers can appear in the declarator:

```
int * volatile const x = 5;
```

cc: illegal type: 'const' appears twice in type specifier

This message occurs when the `const` qualifier appears twice in a variable declaration:

```
const const int x;
```

Correct this error by deleting one of the `const` qualifiers. *See also* `cc: illegal type: 'const' appears twice in declarator`.

cc: illegal type: 'volatile' appears twice in type specifier

This message occurs when the `volatile` qualifier appears twice in a variable declaration:

```
volatile volatile int x;
```

Correct this error by deleting one of the `volatile` qualifiers. *See also* `cc: illegal type: 'volatile' appears twice in declarator`

cc: illegal type: 'const' on type which is already 'const'

This message occurs when the code uses `const` to qualify a type defined with `typedef` that it has already qualified with `const`:

```
typedef const int cint_t;

const cint_t x;
```

Correct this error by deleting one of the `const` qualifiers.

cc: illegal type: 'volatile' on type which is already 'volatile'

This message occurs when the code uses `volatile` to qualify a type defined with `typedef` that it has already qualified with `volatile`:

```
typedef volatile int vint_t;

volatile vint_t x;
```

Correct this error by deleting one of the `volatile` qualifiers.

---

cc: illegal {.

This message indicates redundant braces around the initializers for a bit field:

```
struct {
   int bits : 4;
} x = {{{3}}};
```

Correct this error by removing the extra braces.

---

cc: implicit declaration "extern int *function_name*()" supplied

Message Name: `implicit_decl`

This message indicates that the compiler detects an implicit function declaration, as in the code below.

```
/*compile with
   cc -d implicit_decl=e file.c*/
main()
{
   float rc = some_func();
}
```

---

cc: include file nesting is limited to *n* files; can't include '*filename*'

The compiler can only handle concurrently nested include files to the specified depth. One possible cause is a "recursive" `#include`, where an include file includes itself, either directly or indirectly.

cc: include file *file* not found

This message indicates that the preprocessor was unable to locate *file*. When *file* is delimited by double quotes, the preprocessor looks in the following locations (in order of first place searched to last place searched):

1. Directories specified by the `-I` preprocessor option
2. Current file directory
3. System directory, /usr/include

When *file* is delimited by angle brackets, <*file*>, the preprocessor looks in the following locations:

1. Directories specified by the `-I` preprocessor option
2. System directory, /usr/include

cc: incorrect use of the stringizing operator

Message Name: `pp_badstr`

This message indicates that the operand of the stringizing operator (#) is not one of the formal arguments in the formal argument list of the function-like macro definition:

```
/* compile with cc -d pp_badstr=w file.c */

# define x(f) #g
```

Correct this error by ensuring that the operand of the stringizing operator is an argument in the formal argument list.

cc: incorrect use of the tokenpasting operator

Message Name: `pp_badtp`

This message indicates that the result of combining the left and right operands of the ## macro definition operator is not

- a legal operator,
- an identifier that has been defined, or
- a legal number.

The following example show these errors:

```
# define op(b,c) b##c
x op(+,-) 5; /* +- is not a valid token */

# define id(num) "id" ## num
int id = id(3);
            /* "id"3 is not a valid token */

# define frac(dot,num) dot ## num
float x = frac(.,z);
            /* .z is not a valid token */
```

To correct errors like these, you may examine the output of the preprocessor using the `-E` preprocessor command line option.

`cc: integer constant expected.`

Some C contexts require integer constants. One such context is the length of a bit field. The code below is illegal:

```
int y = 3;
struct { int x : y; } z;
```

This code is correct:

```
struct { int x : 3; } z;
```

`cc: integer literal` *value* `contains long long suffix`

Message Name: `long_long_suffix`

The `long long` data type is a CONVEX extension. This diagnostic option detects a `long long` integer suffix (`ll` or `LL`).

The following code generates this message:

```
/* check with
   lint -std -d long_long_suffix=e file.c */

long long int x = 4LL;
```

`cc: integer literal` *value* `contains unsigned suffix`

Message Name: `unsigned_suffix`

This message occurs when the compiler detects an integral suffix `U` or `u`.

`cc: invalid integer expression found while parsing #if:` *message*

This message indicates that a parsing error occurred in a `#if`, or `#elif` preprocessor directive. *message* provides additional information on the cause of the error.

This code generates the messages shown here:

```
#if 4.
#endif
illegal token
syntax error
```

`cc: invalid wide character constant` *name*. **`Length must be 1.`**

A character constant that has an `L` prefix is a wide character constant. The wide character constant has a maximum length of one character:

```
char wch = L'ss';
```

This code is illegal. The extended and backward-compatible modes permit a character constant to have between one and eight characters, while the strict and conforming modes permit between one and four characters.

However, if the character constant must be a wide character constant, delete the extra characters.

`cc: '`*identifier*`' is not a DIRECTIVE.`**

This message is displayed when the compiler does not recognize a pragma (directive). Appendix B, "Pragmas and directives," describes the pragmas (directives).

cc: label '*name*' defined but not referenced.

This message indicates that label *name* is not used by a `goto` statement in your program:

```
main()
{
   label:
   ;
}
```

Correct this error by removing the unused label.

cc: label '*label name*' referenced but not defined.

The code uses the indicated *label name* in a `goto` statement, but does not define it anywhere in the function as a label.

cc: left operand of -> or . does not have a member '*member name*'.

This message occurs when *member name* is not a member of the `struct` that declared the left operand of the `->` or . operator.

lint: >>>>> C O M P I L E R E R R O R <<<<<
>>>>> See your system manager for help <<<<<
lint: Lint file does not match this version of lint

The lint program is dependent on lint versions of all standard libraries that accompany the C compiler. lint libraries contain information that lint uses to ensure that functions in the library are called correctly by the functions in your program. The lint libraries have names of the form `llib-l`*name*`.ln`. This error message indicates that the specified lint library was created using a previous version of the lint program. If you created the lint library using a previous version of lint, you must create a new library using the current lint program. Otherwise, inform your system manager when this error occurs.

cc: long long is not available in ANSI compatible mode

The `long long` data type is a CONVEX extension to the ANSI C standard. Therefore, it is not available in the conforming (`-std`) and strict (`-str`) modes.

cc: loop directive not textually immediately before loop.

Pragmas (directives) that affect loops must immediately precede that loop. The assignment to n between the pragma and the loop causes this message:

```
typedef int Mat[100][28];

int main()
{
   int i,j,n,m=100;
   Mat a,b;

   #pragma _CNX scalar
   n = 28;
   for(j=0; j<n; j++)
      for(i=0; i<m; i++)
        a[i][j] = b[i][j] + 1;
}
```

cc: lvalue required.

An lvalue (locator value) is an expression that locates a data object in memory. Possible lvalues are:

- Arithmetic variables
- Pointer variables
- Enumerated variables
- Structure variables
- Union variables

Also:

- A subscript expression *array-name*[*integral-expression*] is an lvalue.

- The value resulting from a `struct/union` `.` or `->` operator is an lvalue if the left operand is an lvalue. (`func().x` is not an lvalue.)
- If the operand of the indirection operator, `*`, points to a data object, not a function, the result of the operator is an lvalue.
- A parenthesized expression is an lvalue if its unparenthesized expression is an lvalue.
- Some operators that require an lvalue are: `++`, `--`, `=`, and unary `&`.
- If the code does not meet one of these requirements, the compiler generates an appropriate message:

```
int b[5];
b = 5;
```

The assignment operator generates a message. Even though b is the left operand of an assignment operator, it is also the base address of an array, which can never be modified:

```
int func(void), r(void);

main()
{
   func = r;
   return(0);
}
```

The left operand of the assignment operator cannot be modified because the addresses of functions cannot be modified.

`cc: macro actuals ended unexpectedly, inserting )`

This message indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis:

```
int j;
# define a(b,c,d) b##c##d
a(i,n,t
   j;
```

In this case, the preprocessor inserts the right parenthesis, permitting the code to declare the `j` identifier as an `int`. Correct this error by inserting the right parenthesis.

cc: macro *name* can not be modified or removed

The code attempted to redefine or undefine a predefined macro. For instance, since __FILE__ is a predefined macro, you cannot undefine it as in the code below.

```
#include <stdlib.h>

#undef __FILE__
```

Predefined macros include the following:

- __LINE__
- __FILE__
- __DATE__
- __TIME__
- __STDC__
- defined

cc: macro formal argument *identifier1* and *identifier2* are not distinct

Message Name: pp_macro_arg

This message indicates that the formal arguments of a function-like macro defined with #define have the same name:

```
/* compile with cc -d pp_macro_arg=w file.c */

# define x(y,y) y##y
```

Correct this error by making each of the formal parameters distinct and converting each reference to the formal parameters in the macro definition to the appropriate name.

cc: macro *name* redefined with a new replacement list

Message Name: pp_macro_redefinition

This message indicates that the replacement list for the *name* macro definition is not the same as in its original definition:

```
/* compile with
   cc -d pp_macro_redefinition=w file.c */
```

```
# define foo(x,y) x##y
# define foo(a,b) a##b
```

Replacement lists in a macro redefinition *must* be exactly the same. However, when the preprocessor encounters this error, it replaces the previous definition with the current definition.

---

cc: malformed number: *number*

This message indicates that the preprocessor has encountered an illegally formed number; these numbers have a letter embedded in them, or an illegal suffix, as in the code below.

```
int x = 0x34g;
int y = 23145;
```

---

cc: maximum number of scalars exceeded. Restructure your computations.

This message occurs when the number of scalar variables referenced in a function exceeds the size of a compiler table. Multiple references to a variable identifier require one table entry; multiple references to the same variable using different identifiers require one entry per identifier.

---

cc: Maximum optimization level available is *level*

This message occurs if the compiler does not support the specified optimization option. The compiler has two versions: one that performs scalar optimizations and one that performs vector and parallel optimizations in addition to scalar optimizations. The example below generates this message.

```
cc -O2 file.c
```

This command line generates the message if the scalar version of the compiler is being used because `-O1` (or `-O`) is the maximum level of optimization on the scalar compiler.

---

cc: misplaced #elif

This message indicates that a `#elif` preprocessor directive follows a `#else` preprocessor directive without an intervening `#if`, `#ifdef`, or `#ifndef` preprocessor directive, as in the example below.

```
#ifndef _<|>_FILE_<|>_
#else
#elif 1
#endif
```

This message may indicate a logic error or an extra `#elif` preprocessor directive.

---

cc: misplaced #else

This message indicates that an `#else` preprocessor directive follows another `#else` preprocessor directive without an intervening `#if`, `#ifdef`, or `#ifndef` preprocessor directive, as in the example below.

```
#ifndef _<|>_FILE_<|>_
#else
#else
#endif
```

This message may indicate a logic error or an extra `#else` preprocessor directive.

---

cc: misplaced comma in formal macro argument list

This message indicates that the preprocessor found a comma or a right parenthesis when an identifier was expected in the formal argument list of a function-like macro definition:

```
# define a(,b)
# define c(d,)
```

Correct this error by removing the unnecessary comma or by inserting another identifier.

---

lint: lint: misplaced lint directive.

Message Name: `misplaced_lint_directive`

This message indicates that a lint directive occurs in the wrong context. Possible causes of this message are:

- ARGSUSED directive does not occur in file scope.
- LINTLIBRARY directive does not occur in file scope.
- NOTREACHED directive occurs in file scope.

- VARARGS directive does not occur in file scope.

Turn off this message by checking your source code with `lint -d misplaced_lint_directive`.

---

cc: missing #endif

This message indicates that an `#if`, `#elif`, or `else` does not have a terminating `#endif`. The following example demonstrates this problem:

```
main()
{
   int x;
#if y }
```

Correct this error by including the `#endif` directive in the correct location.

---

cc: missing component for -k in preprocessing input file name

Message Name: `pp_badkfile`

This message is only generated when you run the C preprocessor by itself with the `cpp` command. It indicates that the `-k` file is specified on the command line without an input file. In this case, the preprocessor is expecting input from stdin. However, the `-k` option does not process stdin input.

---

cc: missing newline at end of file (supplied)

A source file should end with a newline character. If it does not, the compiler issues this warning and supplies the newline character.

---

lint: lint: negative constant converted to unsigned type

Message Name: `negative_to_uns`

This message indicates that the code assigns a negative constant to an unsigned type:

```
/* check with lint -d negative_to_uns=w
file.c */
```

```
unsigned x;
x = -1;
```

Converting a negative constant to an unsigned type results in a very large positive number. If this is not the intention of the assignment, change the type of the identifier.

---

cc: negative field size for field '*member name*'.

The integer used to specify the number of bits in a bit field must be a positive integer that is less than the number of bits required to represent the type of the field.

---

cc: nested macro invocation is missing complete argument list

This message indicates that you have some nested macros which do not have a complete argument list:

```
#define a b(3

#define b(i) (i+1)

#define c(x) x

c(a)
```

Correct this error by ensuring that your nested macros have complete argument lists.

---

cc: newline in string or character constant (inserting close quote)

You cannot embed newline characters in string constants or character constants, as in the code below:

```
char c[] = "asdf

;lkj";
```

The compiler tries to recover from the error by inserting a double quote before the newline character (after "f"). If you must embed a newline character in a string constant or character constant, use the newline character constant, "\n". For example, one correct version of the previous example is:

```
char c[] = "asdf\n;lkj";
```

cc: next_task directive with no begin_tasks ignored

You must precede the `next_task` pragma with a `begin_tasks` pragma. The compiler ignores the `next_task` pragma if this is not the case.

lint: >>>>> C O M P I L E R E R R O R <<<<< >>>>> See your system manager for help <<<<< NO AVAILABLE MEMORY

The compiler has insufficient memory to compile the source file. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

cc: no comma separating formal arguments to macro

This message indicates that commas do not separate the formal arguments of a function-like macro definition:

```
# define f(x y) x##y
```

Correct the error by inserting commas in appropriate locations.

lint: no conversion character in '*statement*'

The `printf` function consists of a format string and a list of arguments; the format string contains conversion characters which the routine uses to format and insert the arguments into the output string. The "`%`" character and an optional set of flags precede each conversion character. If the `%` character and one or more flags appear but the conversion character does not, as in the line below, this error occurs.

```
printf("%-10",c);
```

cc: no digits in exponent of floating point constant

This message occurs when the exponent part of a floating-point constant does not contain any digits:

```
float value = 75.E;
```

Correct this error by including digits in the exponent part of a floating-point constant.

cc: no digits in hexadecimal constant

A number beginning with `0x` or `0X` is a hexadecimal constant. If no hexadecimal digits follow one of these prefixes, as in the following example, this message occurs:

```
int value = 0x;
```

Correct this error by following the hexadecimal prefix with hexadecimal digits.

cc: no matching #if

This message indicates that a `#endif` preprocessor directive occurred without a preceding `#if`, `#ifdef`, or `#ifndef`.

cc: no members of this type declared

This message indicates that the code declared a `struct` or `union` member, but the declaration did not include an identifier:

```
struct s { int; };
```

Correct this error by including an identifier in the declaration.

cc: non-standard syntax at or before *'character'*

Message Name: `strict_syntax`

This message occurs when the code does not include a semicolon after the last member in a `struct` or `union` declaration or by placing a comma after the last enumeration constant in an `enum` declaration list. These syntaxes are supported for backward-compatibility and are not portable.

cc: Nothing was declared by this declaration

Message Name: nothing_declared

This message occurs when the code includes empty declarations such as:

```
int ;

struct { int x; };
```

cc: NULL at end of string initializer 'characters' truncated

This message occurs when the number of characters in the string initializer of an array is equal to the number of elements in the array:

```
char arrx[5] = "12345";
```

The message indicates that the array does not have enough room for a NULL character to terminate the string. You should increase the dimension of the array to allow the NULL character because most string handling functions expect it. *See also* `cc: too many initializers.`

cc: null dimension.

This message occurs when an array has no specified size for one of its dimensions, as in the code below.

```
int a[5][];
```

A declaration must include all indexes so that the compiler can allocate sufficient space for the array.

lint: lint: Octal constant contains digit 8 or 9

Old versions of C permitted the digits 8 and 9 to be used as octal digits. ANSI C does not permit this:

```
#include <stdio.h>
main()
{
   int x = 09;
```

```
    printf("x = %o\n", x);
}
```

The output of this program is `x = 11` because the digit `9` is an octal `11` and the digit `8` is an octal `10`.

cc: old form assignment operator used.

Some non-ANSI C compilers permitted assignment operators of the form *=op* to be used, where *op* could be one of `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`. The following example demonstrates an ambiguous situation:

```
#include <stdio.h>
main()
{
    int x = 2;

    x =- 3;
    (void)printf("x = %d\n", x );
}
```

When you execute this program, it displays `x = -3`. If you compile with `cc -d old_form_assign`, it displays `x = -1`.

cc: old style directive, use #pragma

Message Name: `pp_old_dir`

This message indicates that a directive of the form `/*$dir` *directive* `*/` is used. Replace directives that use this form with the pragma syntax. Refer to Appendix B, "Pragmas and directives," for information on the new format of compiler directives:

```
/* check with cc -d pp_old_dir=w file.c */

main()
{
    int x;
    /* $dir no_side_effects (func) */
    x = func();
}
```

You can remove this message by using the pragma syntax for a directive, as in:

```
main()
{
   int x;
   #pragma _CNX no_side_effects( func )
   x = func();
}
```

---

cc: operand of ! must have scalar type

The operand of the logical operator ! must be an expression that evaluates to an integral value, a floating-point number, or a pointer.

---

cc: operand '*identifier*' of no_side_effects is not a function identifier.

This message occurs when the *identifier* is not a function name, as in the example below.

```
int func(void);

void main()
{
   int bad;
   #pragma _CNX no_side_effects (bad)
   z = func();
}
```

Correct this error by using the correct function name as an argument of the `no_side_effects` pragma.

---

cc: The operand of unary minus must have arithmetic type

The minus operator that takes one operand is called a unary minus operator; it reverses the sign of its operand. Its operand must be one of the following types:

- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`, `signed short`
- `int`, `unsigned int`, `signed int`
- `long`, `unsigned long`, `signed long`
- `long long`, `unsigned long long`, `signed long long` (This is a CONVEX extension.)

- Enumerated types
- `float`, `double`, `long double`
- `long float` (a CONVEX extension available only in the backward-compatible mode)
- A qualified form of one of these types. The operand of unary plus must have arithmetic type

The plus operator that takes one operand is called a unary plus operator; it does not change the sign of its operand. Its operand must be one of the following types:

- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`, `signed short`
- `int`, `unsigned int`, `signed int`
- `long`, `unsigned long`, `signed long`
- `long long`, `unsigned long long`, `signed long long` (This is a CONVEX extension.)
- Enumerated types
- `float`, `double`, `long double`
- `long float` (a CONVEX extension available only in the backward-compatible mode)
- A qualified form of one of these types

`cc: operand to function call operator () does not identify a function.`

The function-call operator `()` has two operands: an expression that precedes the left parenthesis and an argument list delimited by left and right parentheses. The expression must result in a pointer to a function, a pointer to a pointer to a function, and so on. Any other expression results in this message.

`cc: operands of` *operand* `have incompatible types.`

Each operator in C has requirements on what can be used as its operands. For example, operands of the `-` operator can both be arithmetic, can both be of the same pointer type, or the left operand can be a pointer only if the right pointer is an integer. The combination of any other data types results in the above diagnostic message. Similar situations occur for the remaining operators.

cc: operands of *relational operator* must be pointers to either compatible object types or compatible incomplete types.

This message occurs when code compares two incompatible data type pointers:

```
void func()
{
    int *p;
    char *q;

    if(p < q);
}
```

Compatible pointers include:

- Pointers to compatible data types
- Pointers to arrays of unknown size
- Pointers to structures of unknown size
- Pointers to `void`

All other pointer comparisons, including pointers to function types, are not allowed.

Compatible data types include:

- Types that have the same data type
- Structure types that have the same number of members, the same member names, and compatible member types
- Union types that have the same number of members, the same member names, and compatible member types
- Enumerated types that have the same number of members, the same member names, and compatible member types
- Declarations that refer to the same object or function
- Enumerated types are compatible with integral types
- Qualified types that have the identically qualified version of a compatible type

cc: operands of *operand* point to incompatible types.

This message indicates that the two operands of *operand* are pointers to incompatible types:

```
main()
{
    int *x;
    float *y;

    x -= y; /* or */
    x = y;
}
```

Correct this situation by making the operands of *operator* compatible.

lint: Optimization level lowered to -no in function calling setjmp

Optimization level lowered to -no in function calling _setjmp

Optimization level lowered to -no in function calling sigsetjmp

When the code calls the `setjmp` family of functions and ignores the return value, the compiler lowers the optimization level to `-no`:

```
#include <setjmp.h>

main()
{
    volatile int flag;
    jmp_buf jb;

    flag = 0;
    (void)setjmp(jb);
    flag++;
    if ( flag == 1 ) {
        printf("I didn't get here from
                longjmp\n");
        longjmp(jb,1);
    } else {
```

```
        printf("I did do a longjmp!\n");
    }
    exit(0);
}
```

At higher optimization levels it appears that the `if` clause is executed unconditionally. If this message occurs, rewrite the code using the more traditional style, as in the code below.

```
#include <setjmp.h>

main()

{
    jmp_buf jb;

    if (setjmp(jb) == 0){
      printf("I didn't get here by longjmp\n");
      longjmp(jb,1);
    } else {
      printf("I did do a longjmp!\n");
    }
    exit(0);
}
```

---

cc: '*-string*' option not recognized

This message occurs when the compiler does not recognize a command line option passed to it. This could be an option such as:

```
cc -unknown
```

The `-unknown` option does not exist. This message also occurs when the compiler does not recognize an option argument.

```
cc -or function
```

The `-or` command line option does not have a `function` argument.

cc: order of evaluation undefined for '*expression*'

Message Name: eval_order

This message indicates that *expression* has an undefined order of evaluation. The following source code has two examples that generate this message:

```
/* check with lint -d eval_order=w file.c */

extern int f(int a, int b);

main()
{
   int x=5, y=6;

   y=7, y=1;
   x = x++ + y; /* example 1 */
   return( f( x, x=4 ) ); /* example 2 */
}
```

Order of evaluation is undefined whenever a variable's value changes more than once in a statement (example 1). Two exceptions exist:

- arguments of functions and
- operands of the comma operator.

A function argument assigned a value and passed in two or more argument positions, as in example 2, generate this message. However, the comma operator permits you to assign multiple values to one variable in one statement, as shown in the code above.

Correct this error by splitting your statement into several statements. For example, you could rewrite the above code as:

```
extern int f(int a, int b);
main()
{
   int x=5, y=6, tmp;

   y=7, y=1;
   x = x + y; /* example 1 */
   x++;
   tmp = 4;
   return( f( x, tmp ) ); /* example 2 */
}
```

cc: output file for -k could not be opened

This option is only available when you run the c preprocessor (cpp) on a source file or set of source files. `-k` specifies the list of source files; if you specify more than one file, the last file listed is the output file. Otherwise, cpp sends the results to stdout.

This error occurs when cpp cannot create the output file. For instance, suppose that the user does not have write permission for the local directory:

```
%/usr/convex/bin/cpp -k file1.c file2.c output
cpp: output file for -k could not be opened
```

cc: pointer cast may introduce inefficient alignment

Message Name: `pointer_alignment_efficiency`

This message indicates that the evaluation of the expression on the indicated line may result in decreased memory efficiency. The compiler achieves maximum efficiency when data are given an alignment equal to the size of the item in bytes. For example, `short` integers are aligned on 2-byte boundaries, and `long` integers are aligned on 4-byte boundaries. However, `long long` integers need only be aligned on 4-byte boundaries.

In the following example, a pointer to a `short` integer is cast into a pointer to a `long` integer:

```
/* check with lint -d /
  pointer_alignment_efficiency=e file.c */
int f()
{
   short *a;
   void *x;

   x = (long *)a;
}
```

The compiler detects that a long integer might be aligned on a 2-byte boundary. One way to correct this error is to change a to a pointer to `long`.

cc: Pointer operands of *operator* must reference object types

This message occurs when the code performs pointer arithmetic on pointers to `void`, pointers to incomplete types, or pointers to functions. The code below generates this message.

```
void func()
{
   void *p;

   p += 2;
}
```

This code is illegal because the objects pointed to have an unknown size.

cc: pointer to function type involved in cast

Message Name: `function_pointer_cast`

This message indicates that a pointer to an object is being cast to a function pointer:

```
/* compile with
   cc -d function_pointer_cast=e file.c */
int (*g)();
char *h;

main()
{
   g = (int (*)()) h;
}
```

Correct this error by either changing the type of `h` to a pointer to a function or removing the incorrect assignment.

cc: pointer to function used in arithmetic

Performing arithmetic operations on function pointers is illegal. See the code below.

```
int noid(void){
   return(3);
}
```

```
int (*func) (void);
main()
{
    func = noid;
    func += 2;
    (*func) ();
}
```

This code produces an executable program that may or may not run. Redesign this code so that it performs no pointer arithmetic operations.

---

cc: pointer to void type involved in cast

Message Name: void_pointer_cast

This message indicates that an object of type pointer to void is involved in a cast operation. The following sample code shows three examples that cause this error message:

```
/* check with lint -d void_pointer_cast=e
file.c */
main()
{
    int *i, *j;
    void *v;

    i = (int *) v;
    v = (void *) j;
    i = (void *) j;
}
```

Any cast operation that assigns its results to a `void *` object or vast a `void *` object causes this message. Also, if the type name of the cast is void *, the message is generated. Correct this error by using a different pointer type.

---

cc: positive decimal integer not found after #line

This message indicates that the first argument of the #line preprocessor directive is not a positive decimal integer. Below are some examples which trigger this message.

```
# line @
# line "asdf"
# line 0xff
```

```
# line -1
```

The correct syntax of the `#line` preprocessor directive is:

```
# line decimal_integer ["filename"]
```

where *decimal_integer* is the new line number and the optional *filename* is the new file name.

---

cc: Possibly nested comments - /* seen in comment

This message occurs when the characters /* are found in a comment. This indicates a possible error, because comments cannot be embedded in other comments. Conditional preprocessor directives are recommended for use when blocks of code containing comments must be temporarily disabled.

---

cc: pragma not recognized by CONVEX

Message Name: `pp_unrecognized_pragma`

This message indicates that the word that follows the pragma directive on the indicated line is not _CNX. The following code generates this message:

```
main()
{
    #pragma no_side_effects(func)
}
```

All CONVEX pragmas must be preceded by _CNX or _cnx.

---

lint: precision in 'identifier' is not used

The `printf` and `scanf` functions allow you to scan and print variables with a certain precision. For integers, *precision* determines the minimum number of digits to print. This error occurs when a precision is specified for a conversion character that does not use one:

```
printf("%.10c", 1);
```

cc: preprocessing #error directive: *text*

This message does not indicate an error. The preprocessor displays it whenever it encounters the `#error` preprocessor directive in a file. For example, if the code includes the lines below:

```
#if OTHER_COMPUTER != 0 && CONVEX != 0
#    error "illegal environment"
#endif
```

and if the if conditions are satisfied, the preprocessor generates the message below:

```
cc: Error on line 25 of ctest.c:
preprocessing #error directive:  "illegal
environment"
```

cc: program contains loop to self.

This message indicates that your program contains code that may result in an infinite loop. The following program contains some code that prevents the program from halting:

```
main()
{
   label:
   ;
   if(1 == 0)
      return;
   goto label;
}
```

Correct this error by re-coding your program so that it does not contain an infinite loop. The compiler detects this error only at optimization level `-O1` and above.

lint: >>>>> C O M P I L E R E R R O R <<<<<
>>>>> See your system manager for help <<<<<
source file: **This program is too complex - the parser's stack overflowed.**

The complexity of the source file caused the storage in the compiler's parser stack to be exceeded. Modularization of source files and functions reduces the chance of this error

occurring. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

---

lint: >>>>> C O M P I L E R E R R O R <<<<< >>>>> See your system manager for help <<<<< This program is too complex - too many characters for symbol table.

The storage for the compiler symbol table was exceeded. This error can occur if too many characters are in the source file. One solution is to split the source file into several smaller files. Contact your system manager or the CONVEX Technical Assistance Center (TAC). Refer to the contact(1) man page to learn how to contact the TAC.

---

lint: >>>>> C O M P I L E R E R R O R <<<<< >>>>> See your system manager for help <<<<< This program is too complex - too many names for symbol table.

The storage for the compiler symbol table was exceeded. This error can occur if too many identifiers are in the source file. One solution is to split the source file into several smaller files. Contact your system manager or the CONVEX Technical Assistance Center (TAC). Refer to the contact(1) man page to learn how to contact the TAC.

---

cc: real constant either too large or too small.

The constant indicated by the message cannot be represented as a real constant. The minimum and maximum real constants of the `float`, `double`, `long float`, and `long double` are contained in the `float.h` include file.

---

cc: *'variable'* redeclared: appears twice in formal parameter list

The compiler displays this message for each duplicate of an identifier in a function formal parameter list or prototype.

```
int func(int a, int a, int a);
```

This code causes this message to be displayed twice. Correct this error by renaming the duplicate identifiers.

---

cc: *'function name'* redeclared: body of this function already seen

Each source file may contain only one function definition for each function. The code below generates this error message:

```
int func()
{
   return(2);
}
int func()
{
   return(2);
}
```

This code generates the message because the function func is defined more than once. The definition need not be the same for this error to occur. Correct this error by deleting one of the function definitions.

---

cc: *'identifier'* redeclared: can not change storage class from extern to static after referencing it.

This message occurs when an identifier is redeclared with the `static` storage class after being declared with the `extern` storage class:

```
extern int i;
void f()
{
   i = 1;
}
static int i = 5;
```

When a reference is made to the identifier between the two storage class declarations, the error message occurs. But if no reference to the identifier is made between the declarations, the identifier is declared with the static storage class.

cc: *'variable'* redeclared: declaration as typedef can't override previous declaration.

An identifier that has already been defined cannot be redeclared as a `typedef` *name*. See the code below.

```
int ident;
typedef char ident;
```

Correct this error by renaming either the typedef type or the first identifier.

Identifiers that are labels, tags, or members of structures and unions can have the same name as a typedef name because the compiler is able to distinguish between the two names based on the syntactic context.

cc: *'function'* redeclared: incompatible types.

This message occurs when the return type or argument of a function is not consistent between the function's declaration and definition. For example, after the following declaration:

```
int f(int ch);
```

either of these declarations is illegal:

```
int f(char ch);
int f(const int ch);
```

cc: *'variable'* redeclared: recursive struct or union declaration

Recursive definitions of structures, such as in the code below, are not permitted.

```
struct some_s {
   struct some_s {
      int x;
   } y;
   int z;
} w;
```

Such a declaration is meaningless because the interior `some_s` is not the same as the exterior `some_s`.

If you need a self-referencing structure, Convex suggests that you use pointers:

```
struct some_s {
   struct some_s *sptr;
   int z;
} w;
```

This structure declares a pointer that can point to a copy of itself.

---

cc: *'variable'* redeclared: saw 'static' definition and then external definition.

This message occurs when a static function or variable is redeclared without a storage class keyword, as in the code below.

```
static int i;
int i;
```

Such functions or variables may be redeclared with either `static` or `extern`. Correct this error by redeclaring the variable with a storage class keyword, or delete the redeclaration.

---

cc: *'identifier'* redeclared: saw extern declaration and then static declaration; using static.

Changing *identifier* from the `extern` storage class to the `static` storage class is not defined in the ANSI C standard.

For example, the semantics of the following are not defined:

```
extern int i;
static int i = 5;
```

If the code does not refer to *identifier* between the declarations, the identifier is declared with the `static` storage class. This message may also occur when a function is first declared with the `no_side_effects` pragma and then declared with the `static` storage class:

```
#pragma _CNX no_side_effects( func )
static int func();
```

When the no_side_effects pragma has as an argument a function that has not been defined or declared, that function is implicitly declared as:

```
extern int func();
```

When this happens, precede the no_side_effects pragma with a function prototype of your affected function.

---

cc: *'variable'* redeclared: saw external definition and then 'static' definition.

This message occurs when a function or variable is declared with the extern storage class and then redeclared with the static storage class:

```
extern int func();
static int func();
main()
{}

or:

extern int x = 3;
   /* error requires initialization */
static int x;
```

Correct this error by deleting the incorrect declaration. This message may occur when other statements separate the extern and static declarations as long as the function or variable is not referenced.

---

cc: *'tag'* redeclared: saw struct or union tag, then enum tag

The tags for struct, union, and enum type specifiers occupy the same name space; you cannot duplicate tag names for these type specifiers the same scope as in the code below.

```
enum bool {false, true};

main()
{
   struct bool { int value;};
   enum bool;
}
```

This code generates this message. The declaration of `bool` as a struct is legal because it is in a new block; this declaration hides the declaration of `bool` as an `enum` type in file scope. The second `enum` declaration of `bool` is illegal because the code has already declared it to be a `struct` tag name in the block.

`cc: '`*variable*`' redeclared: saw two initializers`

This message occurs when the code declares and initializes an identifier more than once, as in the code below.

```
int i = 5;
int i = 4;
```

Correct this by deleting the incorrect initialization or declaration. Function definitions or other declarations may occur between the two initializations.

`cc: '`*identifier*`' redeclared: was previously declared as constant of enumerated type.`

Code cannot declare an identifier as a constant of an enumerated type, then redeclare it in the same scope as another type. The code below generates this message.

```
enum tf { true, false };
int true = 10;
main()
{}
```

Correct this error by renaming the `enum` constant or the variable.

Identifiers that are not label names, tags, or members of structures or unions occupy the same *name space*. You cannot have duplicate identifiers in the same name space.

`cc: '`*identifier*`' redeclared: was previously declared as typedef name.`

An identifier declared as a typedef name cannot be redeclared in the same scope as another type. The following code generates this error message.

```
typedef int ident;
int ident = 5;
```

Correct this error by renaming either the typedef identifier or the variable.

Identifiers that are not label names, tags, or members of structures or unions occupy the same *name space*. Duplicate identifiers in the same name space are not allowed.

---

cc: repeated value '*value*' in select directive

None of the parameters of the `select` pragma (directive) can have the same numerical value, as in the example below.

```
#pragma _CNX select(5,7,5)
```

In this example, both the vector and parallel-vector versions of a loop should be run when the trip count of a loop is five or six, which is impossible. The parameters of the `select` pragma (directive) can never have the same value.

---

cc: [] requires a pointer or array type.

This message occurs when none of the operands of the array subscript operator, `[]`, is a pointer or an array name. The following example generates this error message:

```
main()
{
   int i,j,a[20];
   i[j] = 5;
}
```

When you use the array subscript operator, `[]`, one of the operands must be a pointer or array name, and the other must be an integer. Correct this error by ensuring that one operand is a pointer or array name.

---

cc: saw asm statement in '*file*', optimization level reduced to -O0.

This message indicates that the compiler encountered an `asm` statement in file *file*:

```
main()
{
   asm("dsi");
   asm("eni");
```

```
}
```

The compiler cannot optimize code that contains `asm` statements beyond optimization level `-O0`. Isolate functions that contain `asm` statements in separate source files, if possible, so that they do not prevent other functions from optimizing at a higher level.

`lint: scan into const object (arg #`*arg*`)`

This error occurs when the `scanf` function attempts to scan a value into an object declared as `const`:

```
const char c = 'a';

scanf("%c", &c);
```

`lint: scan into volatile object (arg #`*arg*`)`

This error occurs when the `scanf` function attempts to scan a value into an object declared as `volatile`:

```
volatile char c = 'a';

scanf("%c", &c);
```

`cc:` ***variable*** **`set but not used in function`** ***function_name*****`.`**

Message Name: `set_but_not_used`

This message indicates that a function declared an auto variable and assigned it a value, but never used it. The code below generates this message.

```
main()
{
   int i;
   i = 5;
}
```

This situation may indicate a logic error in the program.

`cc: shift by a negative value is undefined`

Message Name: `neg_shift`

This message indicates that the right operand of a shift operator is a negative constant, as in the code below.

```
/* compile with
   lint -d neg_shift=e file.c */
int x = 5;
x = x >> -1;
```

If the right operand of a left shift () is negative, a left shift is performed. In such cases, the right shift or the left shift, uses the absolute value of the right operand.

---

cc: shift by a value greater than object size

Message Name: `shift_too_large`

This message indicates that the right operand of a shift operation exceeds the number of bit positions that the left operand can be displaced. The integral data types `char` and `short int` have the same bit count as the data types `int` and `long int` in this operation. See the code below.

```
/* check with lint -d shift_too_large=e
file.c */
main()
{
   char x = -5;
   x = x >> 32;
}
```

Correct this error by changing the data type to a larger size or reducing the right operand of the shift operator.

---

cc: signed long long bitfields are not supported

This message indicates that a `signed long long int` bit field has been defined, as in the code below.

```
/* compile with
   cc -d non_int_bit_field file.c */
struct {
   long long i:20; /* ok - unsigned */
   signed long long j: 40;/* not supported */
} some_s;
```

The `signed long long` type of bit field is not supported.

`cc: sizeof(bitfield) disallowed`

This message indicates that the code uses a bit field as the operand of the `sizeof` operator. ANSI C does not allow code that uses a bit field as an argument of the `sizeof` operator.

`cc: sizeof(function) disallowed.`

The operand of the `sizeof` operator cannot be a function, but it may be a pointer to a function, as in the code below.

```
int func(int a);
int x = sizeof(func);
```

is illegal, while

```
int (*func)(int a);
int x = sizeof(func);
```

is not illegal.

`cc: sizeof(void) disallowed.`

The `sizeof` operator does not permit `void` as an operand. The following code generates this message.

```
int k = sizeof(void);
```

Correct this error by replacing `void` with the correct data type.

`cc: static variable` *name* `unused`

This message indicates that the code declares the variable *name* in file scope with the `static` storage class, but does not use it. The code below causes this message to appear.

```
static int j;
main()
{
static int no_err;
}
```

Correct this error by deleting the unused `static` variable.

cc: storage class ignored - no declarators were declared

Message Name: class_ignored

Although you may use an explicit storage class when declaring a structure, it is not useful: the scope of the declaration does not extend to any variables declared with that struct type. The code below generates this message.

```
/* compile with
   cc -d class_ignored=e file.c */
static struct some_s {
   int x;
   int y;
};
struct some_s bad;
```

In this example, bad does not have static storage; the static storage class has no impact on struct declarations defined with some_s.

cc: storage class specifier not allowed in this context

This message indicates that a storage-class keyword appears in a context in which it is not allowed. The five storage-class keywords are auto, extern, register, static, and typedef. For example, you cannot use storage-class specifiers when declaring struct members:

```
struct x {
   typedef int number;
   register float x;
};
```

Correct this error by removing the typedef declarations, if any, or the storage-class specifiers.

cc: string or character constant not terminated

This message indicates that a string or character constant was not terminated by a closing double quote (") or a closing single quote ('), respectively. The example below generates this message.

```
#line 4 "s
#define x '23
```

Correct this error by including the closing double or single quote.

---

cc: struct *name* has no member information

Message Name: `incomplete_record`

This message indicates that the code declares the *name* structure as a `struct`, but does not declare its members. The code below generates this message.

```
/* check with lint -d incomplete_record=w
file.c */
struct empty_s;
```

Correct this situation by declaring the members of the `struct` or by deleting the declaration. This diagnostic option also detects empty `union` declarations.

---

cc: struct/union or struct/union pointer required.

This message occurs when the . operator is used instead of the -> operator, as in the code below.

```
struct { int x; } *p;
int i;
i = p.x;
```

Because `p` is a pointer to a `struct`, use the `->` operator instead of the . operator in this example.

```
i = p -> x;
```

---

cc: A structure or union member may not have a function type.

The member of a structure or union can not be a function, but it can be a pointer to a function.

```
struct sample_s {
   int func( void );
};
```

generates the message, but

```
struct sample_s {
   int (*func)( void );
};
```

does not because it uses a pointer to a function instead of a function type.

---

cc: A structure or union member may not have type void.

A structure or union member cannot have type `void` because the compiler cannot determine how much memory to allocate for the member. The following example generates this message:

```
struct some_s {
   void x;
};
```

Correct this error by changing the `void` type to some other type, possibly `void *`.

---

cc: subscript not constant or constant expression.

This message occurs when a nonconstant expression specifies the size of an array. The constant expression must evaluate to an integer at compile time. Integral constant expressions include:

Integer constants: decimal, octal, and hexadecimal

- Enumeration constants
- Character constants
- `sizeof` expressions
- Floating-point constants that are immediate operands of integral casts except as part of an operand to the `sizeof` operator

---

cc: subscript of array *array name* is not of an integer type

The expression contained within the `[ ]` operator must evaluate to an integer type. Thus, while

```
int b[10];
float c = 1.5;

b[ c ] = 5;
```

generates this message,

```
int b[10];
float c = 1.5;

b[ (int)c ] = 5;
```

does not generate any message.

---

cc: a switch case label must be integer.

A case label of a switch statement must be an integral type. The code below generates this message.

```
int expn = 3;

switch( expn ){
   case 3.0: ;
}
```

---

cc: a switch control expression must be integer.

The control expression of a switch statement must have an integral type, as in the code below.

```
main()
{
   float expn = 3.0;

   switch( expn )
   ;
}
```

---

cc: a switch control expression must be an integral type

The switch control expression must evaluate to an integral value. Therefore, the following code will generate this error:

```
main() {
```

```
    float f = 4.5;

    switch(f) {
    .
    .
    .
    }
}
```

cc: synch_parallel directive ignored - directive not allowed on outer loop

This message occurs when you did not use the `synch_parallel` optimization pragma on the innermost loop of code that has nested loops, as in the code below.

```
main()
{
    int h,i,j;
    float a[20][300][10],b[20][300][10];

    for( h=0; h<20; h++ )
          #pragma _CNX synch_parallel
          for(j=0; j<10; j++)
    for(i=0; i<100; i++ )
          a[h][i+1][j] = a[h][i][j] +
          b[h][i][j];
}
```

Correct this error by applying the `synch_parallel` pragma to innermost loops only.

cc: Syntax Error at "*string*"

The code has a syntax error in the indicated file at *string*. If other errors precede this error, the error may go away when you fix the preceding errors.

cc: Syntax Error at typedef name *identifier*

This error is similar to an unqualified syntax error, but indicates the compiler has interpreted the *identifier* as a `typedef` name.

cc: syntax error in *directive* directive

The indicated pragma (directive) contains a syntax error. Refer to Appendix B, "Pragmas and directives," for assistance with *directive*.

cc: Syntax error recovery skipped to '*char*' *file*

Message Name: `skip_to_char`

This message indicates that the compiler is skipping to *char* in an attempt at error recovery. It will follow another syntax error message and does not require additional action to fix.

cc: Syntax error recovery skipped to end of file

Message Name: `skip_to_eof`

This message indicates that the compiler is skipping to the next input file because it cannot recover from compilation errors. It will follow another syntax error message and does not require additional action to fix.

cc: tag redeclared: "union *tag*;" can't match any existing tag.

This message occurs when *tag* is identical to another tag (`enum`, `struct`, or `union`) declared previously in the same block scope, as in the code below.

```
int f(){
   union some_tag { true, false };
   union some_tag;
}
```

To avoid this error, rename *tag* with a tag name not used elsewhere in the block.

cc: tag redeclared: "struct *tag*;" can't match any existing tag.

This message occurs when *tag* is identical to another tag (`enum`, `struct`, or `union`) declared previously in the same block scope, as in the code below.

```
int f(){
   struct some_tag { true, false };
   struct some_tag;
}
```

To avoid this error, rename the *tag* with a tag name not used elsewhere in the block.

cc: tag '*tag_name*' redeclared.

C provides three types of tags: `struct`, `union`, and `enum`. This message occurs when more than one declaration of a single tag is visible at one time in a source file, as in the code below.

```
enum color { red, white };
struct color {
   int hue;
   int intensity;
};
```

This code fragment generates this message. Correct the error by renaming one of the tags.

cc: '*option*' target machine not supported

This message occurs when the compiler does not recognize an argument of the `-tm` command line option, as in the example below.

```
cc -tm e1 file.c
```

This code generates this message because `e1` is not a valid argument of the `-tm` command line option.

cc: There is no control path leading to a return from this function.

This message occurs when the compiler is unable to determine whether the indicated function has an exit, as in the code below.

```
int main()
{
start:
   exit(0);
   goto start;
}
```

cc: this machine does not have hardware support for IEEE.

This message occurs when the compiler is invoked with the `-fi` command line option on a computer that is not equipped with the IEEE support hardware.

If the machine on which the program runs is not the same as the machine used to compile the program, both machines must have hardware support for IEEE if such hardware is required by the program.

cc: too few arguments in call to function prototype

This message indicates that the number of arguments in a function call does not agree with the number of arguments declared in its function prototype. The code below generates this message.

```
void func(int a, int b);

main()
{
   func(3);
}
```

lint: too few values for format

In the formatted output and input functions `printf` and `scanf`, the number of arguments should equal the number of conversion characters in the format string. If the number of values is smaller than the number of conversion characters, this

error appears. For instance, in the statement below, `scanf` has one conversion character in its format, but no values to insert into it.

```
scanf("%c");
```

cc: too many arguments in call to function prototype

This message indicates that the number of arguments in a function call does not agree with the number of arguments declared in its function prototype, as in the code below.

```
void func(int a, int b);
main()
{
   func(3, 4, 5);
}
```

cc: Too many errors to continue

The code has too many errors to permit the compiler to complete its examination of the source code. When the compiler reports an error, it makes an assumption about what the programmer intended before it continues compiling. If that assumption is incorrect, the compiler can report errors that don't exist.

This *cascade effect* can prevent the compiler from checking the entire source file. Removing the initial error can resolve subsequent errors.

cc: too many initializers.

This message occurs when an array, `struct`, or `union` is initialized with too many values, as in the code below.

```
char arry[5] = "123456";
```

In this declaration the array `arry` has five elements, but the string has 6 characters in it. *See related error* `NULL at end of string initializer 'characters' truncated.`

lint: too many values for format

In the formatted output and input functions `printf` and `scanf`, the number of arguments should equal the number of

conversion characters in the format string. If the number of values is greater than the number of conversion characters, this error appears. For instance, in the statement below, `printf` has one conversion character in its format, but two values to insert into it.

```
printf("%c\n",c,c);
```

cc: Translation unit contains no external declarations

Message Name: `no_external_declaration`

Every translation unit (source file, compilation unit) must have at least one external declaration. This is an ANSI C requirement, but it is not a requirement of CONVEX C.

cc: Trigraph processing is not available in pcc mode

The code is compiled in backward-compatible (`-pcc`) mode, but includes one or more trigraphs. Trigraphs are valid in ANSI C but not in the original Kernighan and Ritchie definition. For more information on trigraphs, see Table 9 on page 61.

cc: the type in a function definition can not be provided by a typedef

It is legal to declare a function type using `typedef`:

```
typedef int func_t(int param);
```

However, such a function type cannot be used to *define* a function:

```
typedef int func_t(int param);

func_t func
{
}
```

The above function definition is illegal. Resolve this message by defining the function with a type synonymous with the `typedef` name. Use the following example to resolve the problem with your code.

```
int func(int param)
{
}
```

cc: type of function *function name* is not a function type

This message occurs when the body of a function is improperly defined, as in the code below.

```
int func
{
}
```

In this example, the parameter list, including the parentheses, is missing. A set of parentheses must always precede the braces.

cc: type pointed to by formal lacks some qualifiers of actual. (arg #)

Message Name: `arg_ptr_qual`

Qualifiers of a formal function parameter that is a pointer must be the same as those for an actual function parameter that is a pointer. The code below generates this message.

```
/* compile with cc -d arg_ptr_qual=e file.c */

extern int func1( int *x );

main()
{
   int a;
   const int  *b;

   a = func1( b );
}
```

The actual parameter has a `const` qualifier, while the formal parameter does not. You can correct this situation by either declaring a `const` formal parameter or removing the `const` declaration on the actual parameter.

cc: Type pointed to by left operand of assignment must contain all qualifiers of type pointed to by right operand

You may assign pointers to `const`- or `volatile`- qualified data types to pointers of the same qualified types. The following code generates this message because it is not a `const`-qualified type.

```
int const * x;
int *y;

y = x;
```

The following code, however, is legal:

```
int * x;
int const * y;

y = x;
```

Note that this requirement does not exist for `const`- or `volatile`- qualified pointers. The following code is also legal:

```
int * const x;
int * y;

y = x;
```

cc: a typedef name may not have an initial value.

Because a `typedef` name identifies a type, not a variable, it cannot be initialized. The code below can generate this message:

```
typedef int var = 1;
```

To resolve this error, you must initialize the variable when it is declared or in the code itself, as in the code below.

```
typedef int var;

var x = 1;
```

cc: unacceptable operand of &.

Illegal operands of the & operator include:

- Bit fields in a structure
- Variables that have the `register` storage class
- Anything that is not an lvalue

These are objects that have no address, which is the function of the & operator.

cc: unary & for array or function ignored.

This message occurs only in the backward-compatible mode of the compiler. It results when the code uses the unary & operator needlessly. You can specify the base address of an array in several ways. For example, given the declaration

```
int arrx[10];
```

you can specify the base address of the array as one of the following in ANSI C:

- `arrx    /* type is int *  */`
- `&arrx   /* type is ptr to array of 10 ints */`
- `&arrx[0]     /* type is ptr to int */`

This message occurs when you use the second form, `&arrx`. Similarly, you can specify the address of a function in several ways. For example, given the function prototype

```
int func( void );
```

the address of the function can be specified as one of the following:

- `func`
- `&func`

This message occurs when the code uses the second form, `&func`; the unary & operator is redundant in these examples.

cc: unexpected end of file

This message indicates that the compiler encountered the end of a file while it was processing a comment, as in the code below.

```
int j;

/* this comment doesn't end ...
```

Correct this error by terminating the comment causing the error.

cc: Unexpected end of source file.

The compiler encountered the end of the source file unexpectedly. Several causes of this error are listed below.

- End of source file in a comment
- Mismatched pairs of { and }
- Mismatched pairs of (and)

Possible causes for the last two problems include misplaced comments or conditionally compiled code.

cc: union *name* has no member information

Message Name: `incomplete_record`

This message indicates that the code declared *name* `struct` as a `struct`, but did not declare its members. The code below generates this message.

```
/* check with lint -d incomplete_record=w
file.c */

union empty_u;
```

Correct this situation by declaring the members of the `union` or by deleting the declaration. This diagnostic option also detects empty `struct` declarations.

cc: unknown size for variable '*variable name*'.

This message results when a declaration for a variable does not specify a storage size. For example,

```
int arrx[];
```

does not indicate how many elements are in array `arrx`. This is sometimes referred to as an incomplete type.

This message also occurs when a member of a `struct` definition refers to the tag of the `struct` in which the member is declared.

```
struct some_s {
   struct some_s bad;
};
```

This code generates the message because the compiler does not know how much memory to allocate for `some_s`; its definition is not complete. Remedy this situation by referring to the structure with a pointer:

```
struct some_s {
   struct some_s *correct;
};
```

While the structure `some_s` may be undefined at the time the code declares `correct`, pointers have a known size.

`lint: unmatched '[' in format`

A beginning bracket (`[`) appears without a matching end (`]`) bracket.

`lint: unrecognized conversion character in format '`*format*`'`

This error occurs when the format string in a `printf` or `scanf` function uses an unknown conversion character. Conversion characters follow the "`%`" character. In the example below, `z` is an unrecognized conversion character.

```
printf("%z",c);
```

`cc: Unrecognized diagnostic name '`*identifier*`' --ignored`

The compiler did not recognize the *identifier* specified with the `-d` command line option. The beginning of this appendix lists the recognized diagnostic names.

cc: Unrecognized Escape Sequence '*sequence*'

The compiler recognizes three types of escape sequences:

- Character escape sequences including: \', \", \?, \e, \a, \b, \f, \n, \r, \t, \v
- Octal escape sequences: \ followed by one, two, or three octal digits
- Hexadecimal escape sequences: \x followed by hexadecimal digits

The compiler interprets \c as c when c is not one of the recognized characters in a character escape sequence. This feature is implementation-defined and may be different on another compiler. Refer to Chapter 3, "Compatibility modes," for more information on escape sequences.

cc: unrecognized preprocessing directive *identifier*, skip to end of line

This message indicates that the compiler encountered an unknown preprocessor directive, as in the code below.

```
# ifdefined x
```

cc: unrecognized preprocessing flag '*flag*', ignored

Message Name: pp_badflag

This message is only generated when the C preprocessor is run by itself with the cpp command. It indicates that you used an option that the preprocessor does not recognize. You may have used a command line option that the C compiler recognizes but the C preprocessor does not.

cc: 'unsigned *comparison-operator* 0' can be simplified

Message Name: uns_compare_zero

This message indicates that a comparison operation between an unsigned number and zero is not necessary. The comparison operator cannot be == or !=. The code below generates this message.

```
/* check with lint -d uns_compare_zero=w
file.c */

unsigned x;
if( x >= 0 ) x++;
```

In this case, the expression x can replace the expression `x >= 0`. Similarly, you can replace the <, >, and <= operators.

---

cc: 'unsigned *comparison-operator* negative constant' is constant

Message Name: `uns_compare_neg`

This message indicates that the code compares a `short unsigned` integer and a negative constant, as in the code below.

```
/* check with lint -d /
  uns_compare_neg=w file.c */

c()
{
   short unsigned x;
   if( x <= -1 );
}
```

Because x is unsigned, it is always greater than any negative constants. Consequently, you can replace the expression with a constant Boolean value, such as `1`.

---

cc: *name* unused in function *func*

This message indicates that function *func* declares *name* as a local variable, but does not use it in its function definition, as in the code below.

```
int func( int arg1 )
{
   int y;
   return (arg1);
}
```

Correct this error by removing the unused local variable from the function definition.

cc: '*identifier*' used before being assigned.

This message indicates that the code uses *identifier* in an expression before it assigns *identifier* a value, as in the code below.

```
#include <stdio.h>

main()
{
   int j;
   int a[10];

   a[j] = 5;
   printf("a[%d] = %d\n", j, a[j] );
}
```

This code assigns a default value of 0 to only `static` variables. Correct this error by initializing *identifier* before you use it.

cc: Using -lc in -pcc mode causes incorrect library usage

The `-lc` switch causes the linker to link files with a library supplied for use with the extended mode of the compiler. Using it in the backward-compatible mode can cause linkage errors. In most cases, removing `-lc` from the command line allows linking to proceed normally and produce the correct result. cc automatically searches the correct library sets, so using `-lc` is not necessary.

cc: VARARGS applied to prototype, use '...'.

Message Name: `varargs_on_proto`

This message occurs when the `VARARGS` directive precedes a function definition that has a function prototype declared for it, as in the code below.

```
/* check with lint -d varargs_on_proto=e
file.c */
/*VARARGS*/
int f( int a, int b);
int f( int a, int b )
{
   return (1);
}
```

You can remove this error message in one of two ways. You can declare the function prototype using the '...' syntax.

```
int f( int a, ... );
int f( int a, int b )
{
   return (1);
}
```

Or you can compile with the `-d varargs_on_proto` command line option.

---

cc: VARARGS greater than number of formals.

Message Name: `varargs_too_large`

This message occurs when the argument of the `VARARGS` lint directive is greater than the number of formal arguments in the function that follows it, as in the code below.

```
/* check with lint -d /
varargs_too_large=e file.c */
/*VARARGS3*/
int f( a, b )
int a,b;
{
   return (1);
}
```

Correct this error by changing the reducing the value of `VARARGS`'s argument to a number smaller than the number of formal arguments.

---

cc: variable '*variable name*' can not be declared 'auto' in file scope.

The `auto` storage class only declares local variables. The following program generates this message:

```
auto int z;
main()
{
}
```

Correct this error by removing the `auto` storage class from the declaration of the file scope variable.

cc: variable '*variable name*' can not be declared '*register*' in file scope.

The `register` storage class can only declare local variables and formal parameters. The code below generates this message.

```
register int z;
main()
{
}
```

Correct this code by removing the `register` keyword because the compiler automatically optimizes register use.

cc: variable '*var_name*' redeclared.

This message occurs when a block of code declares an identifier more than once. The brackets { and } delimit a block of code, including the definition of a `struct` or a `union`. The code below generates this message.

```
void func()
{
   int x;
   float x;
}
```

You can correct this condition by deleting the inappropriate declaration or renaming one of the variables.

cc: variable '*var_name*' undefined.

This message occurs when the code uses an identifier in a context where its declaration is not visible. Either the identifier is not defined in the block which uses it or it is not defined with file scope prior to the block which uses it. The code below generates this error.

```
void func()
{
   x = 5;
}
```

To resolve this error, define the variable in a location that is visible to the block which uses it.

cc: vector directive inside vector directive ignored

This message indicates that a vector optimization directive or pragma that inside of a loop that you have directed the compiler to vectorize, as in the code below.

```
#include <stdio.h>
main()
{
   int i,j,sum=0;
   int a[100][100];
   #pragma _CNX force_vector
   for(i=0; i<<100; i++)
      #pragma _CNX force_vector
      for(j=0; j<<100; j++)
         sum += a[i][j];
   printf("sum = %d", sum );
}
```

The `force_vector` pragma that precedes the second loop causes this error message to occur. To correct this error, remove any vectorization pragmas or directives that occur in loops preceded by another vectorization pragma or directive.

cc: void type not allowed in this declaration.

Code cannot use the `void` type to declare variables, as in the code below.

```
void var;
void a[5];
int func(void param);
```

Typically, `void` is used as the return type of a function, the base type of a pointer, or an indiction that a function has no parameters.

cc: void type used in expression.

Declaring the elements of an array as type `void` or defining function parameters as type `void` is illegal, as in the code below.

```
void a[5];
int func(void param);
```

A declaration of type `void` is illegal in these cases because the compiler cannot determine the amount of memory to allocate for the identifiers.

---

cc: warning: case value out of range

A case statement alternative is out of range of type `int`, as in the example below:

```
switch(i)
{
   case 1: printf("Choice 1"); break;
   case 2: printf("Choice 2"); break;
.
.
.
   case 9999999999999999999:
       printf("Choice 9999999999999999999");
      break;
}
```

Since 9999999999999999999 is out of range of type `int`, this statement produces the error message.

---

cc: warning: integer argument #*arg* lacks cast to pointer

This error indicates that the called function is expecting a pointer as an argument but receives a non-pointer instead. The code below results in this message.

```
void func( int *i )
{
   .
   .
   .
}
main()
{
   int n=0;

   func( n );
}
```

To correct this error, you could either pass the address of n:

```
func( &n );
```

or, if you do not plan to modify the variable, change the definition of the function argument:

```
void func( int i)
```

---

cc: warning: semantics of command line "-I<file> -I-" differs from gcc

The CONVEX C compiler (cc) differs from the Gnu C compiler (gcc) in the way they handle the `-I`*file* `-I-` construct. In gcc, any directories specified with `-I` options before the `-I-` option are searched only for the case of `#include "`*file*`"`; they are not searched for the case of `#include <`*file*`>`. cc, on the other hand, searches for any included files in the specified directories.

This difference could be very important, depending on how you have stored files. For instance, suppose that the file `test.c` includes the following `#include` directives:

```
#include "somefile.h"

#include <stdlib.h>
```

You compile it with the following command line:

```
cc -I/work/headerfiles -I- test.c
```

The directory that you have specified, /work/headerfiles, contains the files somefile.h and stdlib.h—and stdlib.h is different from the standard library file.

If you are used to compiling with gcc, you may assume that your code will include the standard stdlib.h and your personal somefile.h; however, cc will include personal versions of both files.

---

cc: wrong number of actual arguments to macro '*name***'**

Message Name: `pp_argcount`

This message indicates that the number of arguments in the actual argument list does not match the number of arguments in the formal argument list of a function-like macro, except in the case where there is only one formal argument and no actual argument. The code below generates this message.

```
/* compile with cc -d pp_argcount=w file.c */
#define a(b,c,d) b+c+d
int x = a(1,2);
int y = a(1,2,3,4);
```

Correct this error by specifying the correct number of actual arguments.

---

cc: zero field size for field '*member name*'.

Code can only use bit fields with a width of zero with unnamed structure members. Such members can pad the structure to the next word alignment. In the following code, the declaration of member x is illegal:

```
struct {
   int p:3;
   int x:0;
   int y:5;
} bad;
```

Correct this error by placing zero bit fields at the end of a structure definition.

---

cc: zero length character constant.

Character constants must contain at least one character in a character sequence. The following code is illegal:

```
int ch = '';
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

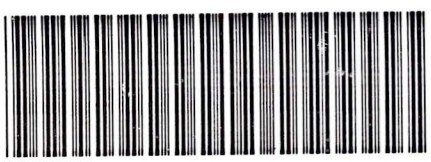
X

Y

Z

ORDER NUMBER
DSW-086

DOCUMENT NUMBER
720-000630-210



CONVEX PRESS