Cray C and C++ Reference Manual

S–2179–50

New Features

Cray C and C++ Reference Manual S–2179–50

Changes were made to this manual to support these features of the Cray C++ 5.0 and Cray C 8.0 releases:

Additional memory pages available for application use	The new `X1_PRIVATE_STACK_GAP` environment variable consolidates, when used with the `X1_PRIVATE_STACK_SIZE` environment variable, the four private stacks within a multistreaming processor (MSP) into one segment, which frees up nontext pages for application use. See Section 2.24, page 49 for more information.
Options to auto `aprun`	You can now use the new `CRAY_AUTO_APRUN_OPTIONS` environment variable to specify options for the `aprun` command when the command is called automatically. See Section 2.24, page 49.
Loopmark for C/C++	You can now create a source listing with loopmark information. See Section 2.10.6, page 20.
OpenMP	(Deferred implementation) You can use OpenMP directives, environment variables, and runtime library routines. See Chapter 4, page 97.

Record of Revision

Version	*Description*
2.0	January 1996 Original Printing. This manual supports the C and C++ compilers contained in the Cray C++ Programming Environment release 2.0. On all Cray systems, the C++ compiler is Cray C++ 2.0. On Cray systems with IEEE floating-point hardware, the C compiler is Cray Standard C 5.0. On Cray systems without IEEE floating-point hardware, the C compiler is Cray Standard C 4.0.
3.0	May 1997 This rewrite supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.0 and the C compiler is Cray C 6.0.
3.0.2	March 1998 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0.2, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.0.2 and the C compiler is Cray C 6.0.2.
3.1	August 1998 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.1, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.1 and the C compiler is Cray C 6.1.
3.2	January 1999 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.2, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.2 and the C compiler is Cray C 6.2.
3.3	July 1999 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.3, which is supported on the Cray SV1, Cray C90, Cray J90, and Cray T90 systems running UNICOS 10.0.0.5 and later, and Cray T3E systems running UNICOS/mk 2.0.4 and later. On all supported Cray systems, the C++ compiler is Cray C++ 3.3 and the C compiler is Cray C 6.3.

3.4	August 2000 This revision supports the Cray C 6.4 and Cray C++ 3.4 releases running on UNICOS and UNICOS/mk operating systems. It includes updates to revision 3.3.
3.4	October 2000 This revision supports the Cray C 6.4 and Cray C++ 3.4 releases running on UNICOS and UNICOS/mk operating systems. This revision supports a new inlining level, inline4.
3.6	June 2002 This revision supports the Cray Standard C 6.6 and Cray Standard C++ 3.6 releases running on UNICOS and UNICOS/mk operating systems.
4.1	August 20, 2002 Draft version to support Cray C 7.1 and Cray C++ 4.1 releases running on UNICOS/mp operating systems.
4.2	December 20, 2002 Draft version to support Cray C 7.2 and Cray C++ 4.2 releases running on UNICOS/mp operating systems.
4.3	March 31, 2003 Draft version to support Cray C 7.3 and Cray C++ 4.3 releases running on UNICOS/mp operating systems.
5.0	June 2003 Supports Cray C++ 5.0 and Cray C 8.0 releases running on UNICOS/mp 2.1 or later operating systems.

Contents

Page

Page

Page

Page

Page

Page

Page

Page

Page

Preface

This publication describes the C and C++ languages implemented by the Cray C++ compiler version 5.0 and the Cray C compiler version 8.0. These compilers are supported on Cray X1 systems running on UNICOS/mp 2.1 or later operating systems.

It is assumed that readers of this manual have a working knowledge of the C and C++ programming languages.

This preface describes how to access Cray documentation and error message explanations, interpret our typographical conventions, order Cray documentation, and contact us about this document.

Accessing Cray Documentation

Each software release package includes the CrayDoc documentation system, a collection of open-source software components that gives you fast, easy access to and the ability to search all Cray manuals, man pages, and glossary in HTML and/or PDF format from a web browser at the following locations:

- Locally, using the network path defined by your system administrator
- On the Cray public web site at:

  `http://www.cray.com/craydoc/`

All software release packages include a software release overview that provides information for users, user services, and system administrators about that release. An installation guide is also provided with each software release package. Release overviews and installation guides are supplied in HTML and PDF formats as well as in printed form. Most software release packages contain additional reference and task-oriented documentation, like this document, in HTML and/or PDF formats.

Man pages provide system and programming reference information. Each man page is referred to by its name followed by a number in parentheses:

manpagename(*n*)

where *n* is the man page section identifier:

1	User commands
2	System calls

3	Library routines
4	Devices (special files) and Protocols
5	File formats
7	Miscellaneous information
8	Administrator commands

Access man pages in any of these ways:

- Enter the `man` command to view individual man pages in ASCII format; for example:

  ```
  man ftn
  ```

  To print individual man pages in ASCII format, enter, for example:

  ```
  man ftn | col -b | lpr
  ```

- Use a web browser with the CrayDoc system to view, search, and print individual man pages in HTML format.
- Use Adobe Acrobat Reader with the CrayDoc system to view, search, and print from *collections* of formatted man pages provided in PDF format.

If more than one topic appears on a page, the man page has one primary name (`grep`, for example) and one or more secondary names (`egrep`, for example). Access the ASCII or HTML man page using either name; for example:

- Enter the command `man grep` or `man egrep`
- Search in the CrayDoc system for `grep` or `egrep`

Error Message Explanations

Access explanations of error messages by entering the `explain` *msgid* command, where *msgid* is the message ID string in the error message. For more information, see the `explain`(1) man page.

Typographical Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
`...`	Ellipses indicate that a preceding element can be repeated.

Ordering Documentation

To order software documentation, contact the Cray Software Distribution Center in any of the following ways:

E-mail:
`orderdsk@cray.com`

Web:
`http://www.cray.com/craydoc/`

Click on the `Cray Publications Order Form` link.

Telephone (inside U.S., Canada):
1–800–284–2729 (BUG CRAY), then 605–9100

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–651–605–9100

Fax:
+1–651–605–9001

Mail:
Software Distribution Center
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`swpubs@cray.com`

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
Contact your Cray representative, or call +1–715–726–4993 (Cray Customer Support Center)

Mail:
Software Publications
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Introduction [1]

The Cray C++ Programming Environment contains both the Cray C and C++ compilers. The Cray C compiler conforms to the International Organization of Standards (ISO) standard ISO/IEC 9899:1999 (C99). The Cray C++ compiler conforms to the ISO/IEC 14882:1998 standard, with some exceptions. The exceptions are noted in Appendix D, page 195.

Throughout this manual, the differences between the Cray C and C++ compilers are noted when appropriate. When there is no difference, the phrase *the compiler* refers to both compilers.

The information is presented as follows:

- Chapter 1, page 1 contains introductory information.
- Chapter 2, page 7 contains information on the `CC`, `cc`, `c89`, `c99`, and `cpp` commands.
- Chapter 3, page 55 contains information on the `#pragma` directives supported by the Cray C and C++ compilers.
- Chapter 4, page 97 contains information about the (Deferred implementation) C and C++ OpenMP API
- Chapter 5, page 129 contains information about Cray Unified Parallel C (UPC).
- Chapter 6, page 137 contains information about supported and unsupported standard C++ features and about the Dinkum C++ library.
- Chapter 7, page 139 contains information on Cray C++ template instantiation.
- Chapter 8, page 147 contains information on the extensions to the C and C++ languages.
- Chapter 9, page 151 contains information on predefined macros.
- Chapter 10, page 155 contains information on debugging Cray C and C++ code.
- Chapter 11, page 157 contains information on interlanguage communication.
- Chapter 12, page 175 contains information on implementation-defined behavior.

- Appendix A, page 183 contains information on requirements for non-C99 code.
- Appendix B, page 185 contains information on the libraries and the loader.
- Appendix C, page 187 contains information on using C++ code developed under Cray C++ Programming Environment 3.5 release or earlier.
- Appendix D, page 195 contains information on the Cray C and C++ dialects.
- Appendix E, page 211 contains information on how to extract information on compiler messages and how to manipulate the message system.
- Appendix F, page 217 contains information on intrinsic functions.

1.1 The Trigger Environment

The user on the Cray X1 system interacts with the system as if all elements of the Programming Environment are hosted on the Cray X1 mainframe, including Programming Environment commands hosted on the Cray Programming Environment Server (CPES). CPES-hosted commands have corresponding commands on the Cray X1 mainframe that have the same names. These are called triggers. Triggers are required only for the Programming Environment.

Understanding the trigger environment will aid administrators and end users in identifying what part of the system a problem occurs when using the trigger environment.

When a user enters the name of a CPES-hosted command on the command line of the Cray X1 mainframe, the corresponding trigger executes, which sets up an environment for the CPES-hosted command. This environment duplicates the portion of the current working environment on the Cray X1 mainframe that relates to the Programming Environment. This allows the CPES-hosted commands to function properly.

To replicate the current working environment, the trigger captures the current working environment on the Cray X1 system and copies the standard I/O as follows:

- Copies the standard input of the current working environment to the standard input of the CPES-hosted command
- Copies the standard output of the CPES-hosted command to standard output of the current working environment

- Copies the standard error of the CPES-hosted command to the standard error of the current working environment

All catchable interrupts, quit signals, and terminate signals propagate through the trigger to reach the CPES-hosted command. Upon termination of the CPES-hosted command, the trigger terminates and returns with the CPES-hosted commands return code.

Uncatchable signals have a short processing delay before the signal is passed to the CPES-hosted command. If you execute its trigger again before the CPES-hosted command has time to process the signal, an indeterministic behavior may occur.

Because the trigger has the same name, inputs, and outputs as the CPES-hosted command, user scripts, makefiles, and batch files can function without modification. That is, running a command in the trigger environment is very similar to running the command hosted on the Cray X1 system.

The commands that have triggers include:

- `ar`
- `as`
- `c++filt`
- `c89`
- `c99`
- `cc`
- `ccp`
- `CC`
- `ftn`
- `ftnlx`
- `ftnsplit`
- `ld`
- `nm`
- `pat_build`
- `pat_help`

- pat_report
- pat_remps
- remps

1.1.1 Working in the Programming Environment

To use the Programming Environment, you must work on a file system that is cross-mounted to the CPES. If you attempt to use the Programming Environment from a directory that is not cross-mounted to the CPES, you will receive this message:

```
trigexecd: trigger command cannot access current directory.
[directory] is not properly cross-mounted on host [CPES]
```

The default files used by the Programming Environment are installed in the /opt/ctl file system. The default include file directory is /opt/ctl/include. All Programming Environment products are found in the /opt/ctl file system.

1.1.2 Preparing the Trigger Environment

To prepare the trigger environment for use, you must use the module command to load the PrgEnv module. This module loads all Programming Environment products and sets up the environment variables necessary to find the include files, libraries, and product paths on the CPES and the Cray X1 system.

Enter the following command on the command line to load the Programming Environment:

```
module load PrgEnv
```

Loading the PrgEnv module causes all Programming Environment products to be loaded and available to the user. A user may swap an individual product in the product set, but should not unload any one product.

To see the list of products loaded by the PrgEnv module, enter the following on the command line:

```
module list
```

If you have questions on setting up the programming environment, contact your system support staff.

1.2 General Compiler Description

Both the Cray C and C++ compilers are contained within the same Programming Environment. If you are compiling code written in C, use the `cc`(1), `c89`(1), or `c99` command to compile source files. If you are compiling code written in C++, use the `CC`(1) command.

1.2.1 Cray C++ Compiler

The Cray C++ compiler consists of a preprocessor, a language parser, a prelinker, an optimizer, and a code generator. The Cray C++ compiler is invoked by a command called `CC`(1) in this manual, but it may be renamed at individual sites. The `CC`(1) command is described in Section 2.1, page 8, and on the `CC`(1) man page. Command line examples are shown in Section 2.22, page 47.

1.2.2 Cray C Compiler

The Cray C compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. The Cray C compiler is invoked by a command called `cc`(1), `c89`(1), or `c99`(1) in this manual, but it may be renamed at individual sites. The `cc`(1) and `c99`(1) commands are discussed in Section 2.2, page 8, the `c89`(1) command is described in Section 2.3, page 9. All are also discussed in the `CC`(1) man page. Command line examples are shown in Section 2.22, page 47.

Note: C code developed under other C compilers of the Cray C++ Programming Environments that do not conform to the C99 standard may require modification to successfully compile with the `c99` command. Refer to Appendix A, page 183.

1.3 Related Publications

The following documents contain additional information that may be helpful:

- *Man Page Collection: Programmer's User Commands*
- *Man Page Collection: C/C++ Library Functions*
- *Optimizing Applications on the Cray X1 System*
- *Cray C++ Tools Library Reference Manual*, Rogue Wave document, *Tools.h++ Introduction and Reference Manual*, publication TPD-0005
- *Cray C++ Mathpack Class Library Reference Manual* by Thomas Keefer and Allan Vermeulen, publication TPD-0006

- *LAPACK.h++ Introduction and Reference Manual, Version 1*, by Allan Vermeulen, publication TPD-0010

Compiler Commands [2]

This chapter describes the compiler commands and the environment variables necessary to execute the Cray C and C++ compilers. These are the commands for the compilers:

- `CC`, which invokes the Cray C++ compiler.
- `cc` and `c99`(1), which invoke the Cray C compiler.
- `c89`, which invokes the Cray C compiler. This command is a subset of the `cc` command. It conforms with POSIX standard (P1003.2, Draft 12).
- `cpp`, which invokes the C language preprocessor. By default, the `CC`, `cc`, `c89`, and `c99`(1) commands invoke the preprocessor automatically. The `cpp` command provides a way for you to invoke only the preprocessor component of the Cray C compiler.

A successful compilation creates an absolute binary file, named `a.out` by default, that reflects the contents of the source code and any referenced library functions. This binary file, `a.out`, can then be executed on the target system. For example, the following sequence compiles file `mysource.c` and executes the resulting executable program:

```
cc mysource.c
a.out
```

With the use of appropriate options, compilation can be terminated to produce one of several intermediate translations, including relocatable object files (`-c` option), assembly source expansions (`-S` option), or the output of the preprocessor phase of the compiler (`-P` or `-E` option). In general, the intermediate files can be saved and later resubmitted to the `CC`, `cc`, `c89`, or `c99`(1) command, with other files or libraries included as necessary.

By default, the `CC`, `cc`, `c89`, and `c99`(1) commands automatically call the loader, which creates an executable file. If only one source file is specified, the object file is deleted. If more than one source file is specified, the object files are retained. The following example creates object files *file1*`.o`, *file2*`.o`, and *file3*`.o`, and the executable file `a.out`:

```
CC file1.c file2.c file3.c
```

The following command creates the executable file `a.out` only:

```
CC file.c
```

2.1 CC Command

The CC command invokes the Cray C++ compiler. The CC command accepts C++ source files that have the following suffixes:

.c

.C

.i

.c++

.C++

.cc

.cxx

.Cxx

.CXX

.CC

.cpp

The CC command also accepts object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The CC command format is as follows:

```
CC [-c] [-C] [-d string] [-D macro[=def]] [-E] [-g] [-G level]
   [-h arg] [-I incldir] [-l libfile] [-L libdir] [-M] [-nostdinc]
   [-o outfile] [-O level] [-P] [-s] [-S] [-U macro] [-V]
   [-Wphase,"opt..."] [-X npes] [-Yphase,dirname] [-#] [-##] [-###]
   file [file]...
```

See Section 2.5, page 10 for an explanation of the command line options.

2.2 cc and c99 Commands

The cc command invokes the Cray C compiler. The cc and c99 commands accept C source files that have the .c and .i suffixes; object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The cc and c99 commands format are as follows:

```
cc or c99 [-c] [-C] [-d string] [-D macro[=def]] [-E] [-g] [-G level]
   [-h arg] [-I incldir] [-l libfile] [-L libdir] [-M] [-nostdinc]
   [-o outfile] [-O level] [-P] [-s] [-S] [-U macro] [-V]
   [-Wphase,"opt...") [-X npes] [-Yphase,dirname] [-#] [-##] [-###]
   files [file] ...
```

See Section 2.5, page 10 for an explanation of the command line options.

2.3 c89 Command

The c89 command invokes the Cray C compiler. This command is a subset of the cc command and conforms with the POSIX standard (P1003.2, Draft 12). The c89 command accepts C source files that have a .c or .i suffix; object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The c89 command format is as follows:

```
c89 [-c] [-D macro[=def]] [-E] [-g] [-I incldir] [-l libfile] [-L libdir]
   [-o outfile] [-O level] [-s] [-U macro] [-Yphase,dirname] files [file] ...
```

See Section 2.5, page 10 for an explanation of the command line options.

2.4 cpp Command

The cpp command explicitly invokes the preprocessor component of the Cray C compiler. Most cpp options are also available from the CC, cc, c89, and c99 commands.

The cpp command format is as follows:

```
cpp [-C] [-D macro[=def]] [-E] [-I incldir] [-M] [-N] [-nostdinc] [-P]
   [-U macro] [-V] [-Yphase,dirname] [-#] [-##] [-###] [infile[outfile]]
```

The *infile* and *outfile* files are, respectively, the input and output for the preprocessor. If you do not specify these arguments, input is defaulted to

standard input (`stdin`) and output to standard output (`stdout`). Specifying a minus sign (`-`) for *infile* also indicates standard input.

See Section 2.5, page 10 for an explanation of the command line options.

2.5 Command Line Options

The following subsections describe options for the `CC`, `cc`, `c89`, `c99`, and `cpp` commands. These options are grouped according to function, as follows:

- Language options:
  - The standard conformance options (Section 2.6, page 12):

Section	Option
Section 2.6.1, page 12	`-h [no]c99`
Section 2.6.2, page 13	`-h [no]conform` and `-h [no]stdc`
Section 2.6.3, page 13	`-h cfront`
Section 2.6.4, page 13	`-h [no]parse_templates`
Section 2.6.5, page 13	`-h [no]dep_name`
Section 2.6.6, page 14	`-h [no]exceptions`
Section 2.6.7, page 14	`-h [no]anachronisms`
Section 2.6.8, page 14	`-h new_for_init`
Section 2.6.9, page 15	`-h [no]tolerant`

  - The template options (Section 2.7, page 15):

Section	Option
Section 2.7.1, page 15	`-h [no]autoinstantiate`
Section 2.7.2, page 15	`-h one_instantiation_per_object`
Section 2.7.3, page 15	`-h instantiation_dir =` *dirname*
Section 2.7.4, page 16	`-h instantiate=`*mode*
Section 2.7.5, page 16	`-h [no]implicitinclude`
Section 2.7.6, page 16	`-h remove_instantiation_flags`
Section 2.7.7, page 16	`-h prelink_local_copy`
Section 2.7.8, page 16	`-h prelink_copy_if_nonlocal`

  - The virtual function options (Section 2.8, page 16): `-h forcevtbl` and `-h suppressvtbl`.

  - General language options (Section 2.9, page 17):

Section	Options
Section 2.9.1, page 17	`-h keep=`*file*
Section 2.9.2, page 17	`-h restrict=`*args*
Section 2.9.3, page 18	`-h [no]calchars`
Section 2.9.4, page 18	`-h [no]signedshifts`

- Optimization options:
  - General optimization options (Section 2.10, page 19)
  - Multistreaming Processor (MSP) options (Section 2.11, page 23)
  - Vectorization options (Section 2.12, page 24)
  - Inlining options (Section 2.13, page 26).
  - Scalar optimization options (Section 2.14, page 27)
- Math options (Section 2.15, page 28)
- Debugging options (Section 2.16, page 31)
- Message control options (Section 2.17, page 33)
- Compilation phase control options (Section 2.18, page 34)
- Preprocessing options (Section 2.19, page 37)
- Loader options (Section 2.20, page 40)
- Miscellaneous options (Section 2.21, page 42)
- Command line examples (Section 2.22, page 47)
- Compile-time environment variables (Section 2.23, page 48)
- Run time environment variables (Section 2.24, page 49)

Options other than those described in this manual are passed to the loader. For more information on the loader, see the `ld`(1) man page.

There are many options that start with `-h`. Multiple `-h` options can be specified using commas to separate the arguments. For example, the `-h parse_templates` and `-h fp0` command line options can be specified as `-h parse_templates,fp0`.

If conflicting options are specified, the option specified last on the command line overrides the previously specified option. Exceptions to this rule are noted in the individual descriptions of the options.

The following examples illustrate the use of conflicting options:

- In this example, `-h fp0` overrides `-h fp1`:

  ```
  CC -h fp1,fp0 myfile.c
  ```

- In this example, `-h vector2` overrides the earlier vector optimization level 3 implied by the `-O3` option:

  ```
  CC -O3 -h vector2 myfile.c
  ```

Most `#pragma` directives override corresponding command line options. For example, `#pragma _CRI novsearch` overrides the `-h vsearch` option. `#pragma _CRI novsearch` also overrides the `-h vsearch` option implied by the `-h vector2` or `-O2` option. Exceptions to this rule are noted in descriptions of options or `#pragma` directives.

2.6 Standard Language Conformance Options

This section describes standard conformance language options. Each subsection heading shows in parentheses the compiler with which the option can be used.

2.6.1 `-h [no]c99` (cc, c99)

Default option: `-h noc99` (cc)

`-h c99` (c99)

The `-h c99` option enables language features new to the C99 standard and Cray C compiler, while providing support for features that were previously defined as Cray extensions. If the previous implementation of the Cray extension differed from the C99 standard, both implementations will be available when the is option enabled. The `-h c99` option is also required for C99 features not previously supported as extensions.

When `-hnoc99` is used, c99 language features such as VLAs and restricted pointers that were available as extensions previously to adoption of the c99 standard remain available to the user.

2.6.2 `-h [no]conform` (`CC`, `cc`, `c99`), `-h [no]stdc` (`cc`, `c99`)

Default option: `-h [no]conform`, `-h nostdc`

The `-h conform` and `-h stdc` options specify strict conformance to the ISO C standard or the ISO C++ standard. The `-h noconform` and `-h [no]stdc` options specify partial conformance to the standard. The `-h exceptions`, `-h dep_name`, and `-h parse_templates` options are enabled by the `-h conform` option in Cray C++.

Note: The `c89` command does not accept the `-h conform` or `-h stdc` option. It is enabled by default when the command is issued.

2.6.3 `-h cfront` (`CC`)

The `-h cfront` option causes the Cray C++ compiler to accept or reject constructs that were accepted by previous `cfront`-based compilers (such as Cray C++ 1.0), but which are not accepted in the C++ standard. The `-h anachronisms` option is implied when `-h cfront` is specified.

2.6.4 `-h [no]parse_templates` (`CC`)

Default option: `-h noparse_templates`

This option allows existing code that define templates using previous versions of the Cray STL (before Programming Environment 3.6) to compile successfully with the `-h conform` option. Consequently, this allows you to compile existing code without having to use the Cray C++ STL. To do this, use the `noparse_templates` option. Also, the compiler defaults to this mode when the `-h dep_name` option is used. To have the compiler verify that your code uses the Cray C++ STL properly, use the `parse_templates` option.

2.6.5 `-h [no]dep_name` (`CC`)

Default option: `-h nodep_name`

This option enables or disables dependent name processing (that is, the separate lookup of names in templates when the template is parsed and when it is instantiated). The `-h dep_name` option cannot be used with the `-h noparse_templates` option.

2.6.6 `-h [no]exceptions` (CC)

Default option: The default is `-h exceptions`, however if the `CRAYOLDCPPLIB` environment variable is set to a nonzero value, the default is `-h noexceptions`.

The `-h exceptions` option enables support for exception handling. The `-h noexceptions` option issues an error whenever an exception construct, a `try` block, a `throw` expression, or a `throw` specification on a function declaration is encountered. `-h exceptions` is enabled by `-h conform`.

2.6.7 `-h [no]anachronisms` (CC)

Default option: `-h noanachronisms`

The `-h [no]anachronisms` option enables or disables anachronisms in Cray C++. This option is overridden by `-h conform`.

2.6.8 `-h new_for_init` (CC)

The `-h new_for_init` option enables the new scoping rules for a declaration in a `for-init-statement`. This means that the new (standard-conforming) rules are in effect, which means that the entire `for` statement is *wrapped* in its own implicitly generated scope. `-h new_for_init` is implied by the `-h conform` option.

This is the result of the scoping rule:

```
{
  .
  .
  .
  for (int i = 0; i < n; i++)  {
    .
    .
    .
  } // scope of i ends here for -h new_for_init
  .
  .
  .
}  // scope of i ends here by default
```

2.6.9 `-h` [`no`]`tolerant` (`cc`, `c99`)

Default option: `-h notolerant`

The `-h tolerant` option allows older, less standard C constructs to facilitate porting of code written for previous C compilers. Errors involving comparisons or assignments of pointers and integers become warnings. The compiler generates casts so that the types agree. With `-h notolerant`, the compiler is intolerant of the older constructs.

The use of the `-h tolerant` option causes the compiler to tolerate accessing an object with one type through a pointer to an entirely different type. For example, a pointer to `long` might be used to access an object declared with type `double`. Such references violate the C standard and should be eliminated if possible. They can reduce the effectiveness of alias analysis and inhibit optimization.

2.7 Template Language Options

This section describes template language options. See Chapter 7, page 139 for more information on template instantiation. Each subsection heading shows in parentheses the compiler with which the option can be used.

2.7.1 `-h` [`no`]`autoinstantiate` (`CC`)

Default option: `-h autoinstantiate`

The `-h` [`no`]`autoinstantiate` option enables or disables automatic instantiation of templates by the Cray C++ compiler.

2.7.2 `-h one_instantiation_per_object` (`CC`)

The `-h one_instantiation_per_object` option puts each template instantiation used in a compilation into a separate object file that has a `.int.o` extension. The primary object file will contain everything else that is not an instantiation. See the –`h instantiation_dir` option for the location of the object files.

2.7.3 `-h instantiation_dir =` *dirname* (`CC`)

Default option: `./Template.dir`

The `-h instantiation_dir =` *dirname* option, specifies the instantiation directory that the `-h one_instantiation_per_object` option should use.

If directory *dirname* does not exist, it will be created. The default directory is `./Template.dir`.

2.7.4 `-h instantiate=`*mode* (CC)

Default option: `-h instantiate=none`

Normally, during compilation of a source file, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by using the `-h instantiate=`*mode* option. *mode* is specified as `none` (the default), `used`, `all`, or `local`.

2.7.5 `-h` [no]`implicitinclude` (CC)

Default option: `-h implicitinclude`

The `-h` [no]`implicitinclude` option enables or disables implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

2.7.6 `-h remove_instantiation_flags` (CC)

The `-h remove_instantiation_flags` option causes the prelinker to recompile all the sources to remove all instantiation flags.

2.7.7 `-h prelink_local_copy` (CC)

The `-h prelink_local_copy` indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations.

2.7.8 `-h prelink_copy_if_nonlocal` (CC)

The `-h prelink_copy_if_nonlocal` option specifies that assignment of an instantiation to a nonlocal object file will result in the object file being recompiled in the current directory.

2.8 Virtual Function Options (`-h forcevtbl`, `-h suppressvtbl` (CC))

The `-h forcevtbl` option forces the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition

of virtual function tables provide no guidance. The `-h suppressvtbl` option suppresses the definition of virtual function tables in these cases.

The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first noninline, nonpure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity).

The `-h forcevtbl` option differs from the default behavior in that it does not force the definition to be local.

2.9 General Language Options

This section describes general language options. Each subsection heading shows in parentheses the compiler with which the option can be used.

2.9.1 `-h keep=`*file* (`CC`)

When the `-h keep=`*file* option is specified, the static constructor / destructor object (`.o`) file is retained as *file*. This option is useful when linking `.o` files on a system that does not have a C++ compiler. The use of this option requires that the `main` function must be compiled by C++ and the static constructor / destructor function must be included in the link. With these precautions, mixed object files (files with `.o` suffixes) from C and C++ compilations can be linked into executables by using the loader command instead of the `CC` command.

2.9.2 `-h restrict=`*args* (`CC`, `cc`, `c99`)

The `-h restrict=`*args* option globally instructs the compiler to treat certain classes of pointers as restricted pointers. You can use this option to enhance optimizations (this includes vectorization).

Classes of affected pointers are determined by the value contained in *args*, as follows:

args	Description
`a`	All pointers to object and incomplete types are to be considered restricted pointers, regardless of where they appear in the source code. This includes pointers in `class`, `struct`, and `union` declarations, type casts, function prototypes, and so on.
`f`	All function parameters that are pointers to objects or incomplete types can be treated as restricted pointers.

`t` All parameters that are `this` pointers can be treated as restricted pointers (Cray C++ only).

The *args* arguments instruct the compiler to assume that, in the current compilation unit, each pointer (`=a`), or each pointer that is a function parameter (`=f`), or each `this` pointer (`=t`) points to a unique object. This assumption eliminates those pointers as sources of potential aliasing, and may allow additional vectorization or other optimizations. These options cause only data dependencies from pointer aliasing to be ignored, rather than all data dependencies, so they can be used safely for more programs than the `-h ivdep` option.

Caution: Like `-h ivdep`, the arguments make assertions about your program that, if incorrect, can introduce undefined behavior. You should not use `-h restrict=a` if, during the execution of any function, an object is modified and that object is referenced through either of the following:

- Two different pointers
- The declared name of the object and a pointer

The `-h restrict=f` and `-h restrict=t` options are subject to the analogous restriction, with "function parameter pointer" replacing "pointer."

2.9.3 `-h [no]calchars` (`CC`, `cc`, `c99`)

Default option: `-h nocalchars`

The `-h calchars` option allows the use of the `@` and `$` characters in identifier names. This option is useful for porting codes in which identifiers include these characters. With `-h nocalchars`, these characters are not allowed in identifier names.

Caution: Use this option with extreme care, because identifiers with these characters are within UNICOS/mp name space and are included in many library identifiers, internal compiler labels, objects, and functions. You must prevent conflicts between any of these uses, current or future, and identifier declarations or references in your code; any such conflict is an error.

2.9.4 `-h [no]signedshifts` (`CC`, `cc`, `c99`)

Default option: `-h signedshifts`

The `-h [no]signedshifts` option affects the result of the right shift operator. For the expression `e1 >> e2` where `e1` has a signed type, when

`-h signedshifts` is in effect, the vacated bits are filled with the sign bit of `e1`. When `-h nosignedshifts` is in effect, the vacated bits are filled with zeros, identical to the behavior when `e1` has an unsigned type.

Also refer to Section 12.1.2.5, page 178 about the effects of this option when shifting integers.

2.10 General Optimization Options

This section describes general optimization options. Each subsection heading shows in parentheses the compiler with which the option can be used.

2.10.1 `-h gen_private_callee` (`CC`, `cc`, `c99`)

The `-h gen_private_callee` option is used when compiling source files containing routines that will be called from streamed regions, whether those streamed regions are created by CSD directives or by the use of the `ssp_private` or `concurrent` directives to cause autostreaming. Refer to Section 3.8.1, page 73 for more information about the `ssp_private` directive or to Section 3.9, page 76 about CSDs.

2.10.2 `-h` [`no`]`aggress` (`CC`, `cc`, `c99`)

Default option: `-h noaggress`

The `-h aggress` option provides greater opportunity to optimize loops that would otherwise by inhibited from optimization due to an internal compiler size limitation. `-h noaggress` leaves this size limitation in effect.

With `-h aggress`, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size.

2.10.3 `-h display_opt`

The `-h display_opt` option displays the current optimization settings for this compilation.

2.10.4 `-h` [`no`]`fusion` (`CC`, `cc`, `c99`)

Default option: `-h fusion`

The –h [no]fusion option globally allows or disallows loop fusion. By default, the compiler attempts to fuse all loops, unless the –h nofusion option is specified. Fusing loops generally increases single processor performance by reducing memory traffic and loop overhead. On rare occasions loop fusing may degrade performance.

Note: Loop fusion is disabled when the vectorization level is set to 0 or 1.

Refer to *Optimizing Applications on the Cray X1 System* for more information about loop fusion.

2.10.5 -h [no]intrinsics (CC, cc, c99)

Default option: -h intrinsics

The -h intrinsics option allows the use of intrinsic hardware functions, which allow direct access to some hardware instructions or generate inline code for some functions. This option has no effect on specially-handled library functions.

Intrinsic functions are described in Appendix F, page 217.

2.10.6 -h list=*opt* (CC, cc, c99)

The -h list=*opt* option allows the creation of loopmark listings. The listings are written to *source_file_name_without_suffix*.lst.

For additional information on loopmark listings, see *Optimizing Applications on the Cray X1 System*.

The values for *opt* are:

a	Use all list options
b	Add page breaks to listing
e	Expand include files
i	Intersperse optimization messages within the source listing rather than at the end
m	Create loopmark listing
s	Create a complete source listing (include files not expanded)
w	Create a wide listing rather than the default of 80 characters

Using `-h list=m` creates a loopmark listing. The `b`, `e`, `i`, `s`, and `w` options provide additional listing features. Using `-h list=a` combines all options.

2.10.7 `-h msp` (`CC`, `cc`, `c99`)

Default option: `-h msp`

The `-h msp` option causes the compiler to generate code and to select the appropriate libraries to create an executable that runs on one or more multistreaming processors (MSP mode). Any code, including code using Cray-supported distributed memory models, can use MSP mode.

Executables compiled for MSP mode can contain object files compiled with MSP or SSP mode. That is, MSP and SSP object files can be specified during the load step as follows:

```
cc -h msp -c ... /* Produce MSP object files */
cc -h ssp -c ... /* Produce SSP object files */
/* Link MSP and SSP object files */
/* to create an executable to run on MSPs */
cc sspA.o sspB.o msp.o ...
```

For more information about MSP mode, refer to *Optimizing Applications on the Cray X1 System*.

2.10.8 `-h` [no]`pattern` (`CC`, `cc`, `c99`)

Default option: `-h pattern`

The `-h` [no]`pattern` option globally enables or disables pattern matching. Pattern matching is on by default.

2.10.9 `-h` [no]`overindex` (`CC`, `cc`, `c99`)

Default option: `-h nooverindex`

The `-h overindex` option declares that there are array subscripts that index a dimension of an array that is outside the declared bounds of that array. The `-h nooverindex` option declares that there are no array subscripts that index a dimension of an array that is outside the declared bounds of that array.

2.10.10 (Deferred implementation) `-h ssp` (`CC`, `cc`, `c99`)

Default option: `-h msp`

The `-h ssp` option causes the compiler to compile the code and select the appropriate libraries to create an executable that runs on one single-streaming processor (SSP mode). Any code, including code using Cray-supported distributed memory models, can use SSP mode.

Executables compiled for SSP mode can contain only object files compiled in SSP mode. When loading object files separately from the compile step, the SSP mode must be specified during the load step as this example shows:

```
/* Produce SSP object files */
cc -h ssp -c ...

/* Link SSP object files */
/* to create an executable to run on a single SSP */
cc -h ssp sspA.o sspB.o ...
```

Since SSP mode does not use streaming, the compiler automatically specifies the `-h stream0` option. This option then causes the compiler to ignore CSDs.

Note: Code explicitly compiled with the `-h stream0` option can be linked with object files compiled with MSP or SSP mode. You can use this option to create a universal library that can be used in MSP or SSP mode.

For more information about SSP mode, refer to *Optimizing Applications on the Cray X1 System*.

2.10.11 `-h [no]unroll` (`CC`, `cc`, `c99`)

Default option: `-h unroll`

The `-h` *no*`unroll` option globally allows or disallows unrolling of loops. By default, the compiler attempts to unroll all loops, unless the `-h nounroll` option is specified, or the `unroll 0` or `unroll 1` pragma is specified for a loop. Loop unrolling generally increases single processor performance at the cost of increased compile time and code size.

Refer to *Optimizing Applications on the Cray X1 System* for more information about loop unrolling.

2.10.12 `-O` *level* (`CC`, `cc`, `c89`, `c99`)

Default option: Equivalent to the appropriate `-h` option

The `-O` *level* option specifies the optimization level for a group of compiler features. Specifying `-O` with no argument is the same as not specifying the `-O` option; this syntax is supported for compatibility with other vendors.

A value of `0`, `1`, `2`, or `3` sets that level of optimization for each of the `-h inline`*n*, `-h scalar`*n*, `-h stream`*n*, and `-h vector`*n* options.

For example, `-O2` is equivalent to the following:

```
-h inline2,scalar2,stream2,vector2
```

Optimization features specified by `-O` are equivalent to the `-h` options listed in Table 1.

Table 1. `-h` Option Descriptions

`-h` option	Description location
`-h stream`*n*	Section 2.11.1, page 23
`-h vector`*n*	Section 2.12.3, page 25
`-h inline`*n*	Section 2.13.1, page 26
`-h scalar`*n*	Section 2.14.2, page 27

2.11 Multistreaming Processor Optimization Options

This section describes the multistreaming processor (MSP) options. For information on MSP `#pragma` directives, see Section 3.8, page 72. For information about streaming intrinsics, see Appendix F, page 217. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.11.1 `-h streamn` (`CC`, `cc`, `c99`)

The `-h stream`*n* option specifies the level of automatic MSP optimizations to be performed. Generally, vectorized applications that execute on a one-processor system can expect to execute up to four times faster on a processor with multistreaming enabled.

These can be used for the *n* argument:

n	Description
0	No automatic multistreaming optimizations are performed.
1	Conservative automatic multistreaming optimizations. Automatic multistreaming optimization is limited to inner vectorized loops and some bit matrix multiplication (BMM) operations. MSP operations performed generate the same results that would be obtained from scalar optimizations; for example, no floating-point reductions are performed. This level is compatible with `-h vector1`, `2`, and `3`.
2	Moderate automatic multistreaming optimizations. Automatic multistreaming optimization is performed on loop nests and appropriate BMM operations. This level is compatible with `-h vector2` and `3`.
3	Aggressive automatic multistreaming optimizations. Automatic multistreaming optimization is performed as with `stream2`. This level is compatible with `-h vector2` and `3`.

2.12 Vector Optimization Options

This sections describes vector optimization options. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.12.1 `-h` [`no`]`infinitevl` (`CC`), `cc`, `c99`)

Default option: `-h infinitevl`

The `-h infinitevl` option instructs the compiler to assume an infinite safe vector length for all `#pragma ivdep` directives. The `-h noinfinitevl` option instructs the compiler to assume a safe vector length equal to the maximum supported vector length on the machine for all `#pragma ivdep` directives.

2.12.2 `-h` [`no`]`ivdep` (`CC`, `cc`, `c99`)

Default option: `-h noivdep`

The `-h ivdep` option instructs the compiler to ignore vector dependencies for all loops. This is useful for vectorizing loops that contain pointers. With `-h noivdep`, loop dependencies inhibit vectorization. To control loops individually, use the `#pragma ivdep` directive, as discussed in Section 3.7.1, page 67.

This option can also be used with "vectorization-like" optimizations found in Section 3.7, page 67.

Caution: This option should be used with extreme caution because incorrect results can occur if there is a vector dependency within a loop. Combining this option with inlining is dangerous because inlining can introduce vector dependencies.

Caution: This option severely constrains other loop optimizations and should be avoided if possible.

2.12.3 `-h vector`*n* (`CC`, `cc`, `c99`)

Default option: `-h vector2`

The `-h vector`*n* option specifies the level of automatic vectorizing to be performed. Vectorization results in dramatic performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option.

Argument *n* can be one of the following:

n	Description
`0`	No automatic vectorization. Characteristics include low compile time and small compile size. This option is compatible with all scalar optimization levels.
`1`	Specifies conservative vectorization. Characteristics include moderate compile time and size. No loop nests are restructured; only inner loops are vectorized. Not all vector reductions are performed, so results do not differ from results obtained when the `-h vector0` option is specified. No vectorizations that might create false exceptions are performed. The `-h vector1` option is compatible with `-h scalar1`, `-h scalar2`, `-h scalar3`, or `-h stream1`.
`2`	Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-h vector1` is specified because of vector reductions. The `-h vector2` option is compatible with `-h scalar2` or `-h scalar3` and with `-h stream0`, `-h stream1`, and `-h stream2`.

`3`	Specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-h vector1` is specified because of vector reductions. Vectorizations that might create false exceptions in rare cases may be performed.

Vectorization directives are described in Section 3.7, page 67.

2.12.4 `-h [no]vsearch` (`CC`, `cc`, `c99`)

Default option: `-h vsearch`

The `-h vsearch` option enables vectorization of all search loops. With `-h novsearch`, the default vectorization level applies. The `novsearch` directive is discussed in Section 3.7.4, page 69. This option is affected by the `-h vector`*n* option (see Section 2.12.3, page 25).

2.13 Inlining Optimization Options

This section describes inlining options. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.13.1 `-h inline`*n* (`CC`, `cc`, `c99`)

Default option: `-h inline2`

The `-h inline`*n* option specifies the level of inlining to be performed. Inlining eliminates the overhead of a function call and increases the opportunities for other optimizations. Inlining can also increase object code size. Inlining directives and the `inline` keyword are unaffected when *n* is not zero. They are ignored when *n* is zero.

Use one of these values for *n*:

n	Description
`0`	No inlining is performed.
`1`	Conservative inlining. Inlining is performed on functions explicitly marked by either: • The `inline` keyword • A `#pragma _CRI inline` directive

	• (C++) implicit inline applied to member functions
2	Same function as `inline1` except larger routines are loaded.
3	Aggressive automatic inlining. All functions are candidates for inlining except those specifically marked with a `#pragma noinline` directive.
4	More aggressive automatic inlining. The `inline4` optimization level is the same as `inline3` but may inline larger routines.

2.14 Scalar Optimization Options

This section describes scalar optimization options. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.14.1 `-h [no]interchange` (`CC`, `cc`, `c99`)

Default option: `-h interchange`

The `-h interchange` option allows the compiler to attempt to interchange all loops, a technique that is used to gain performance by having the compiler swap an inner loop with an outer loop. The compiler attempts the interchange only if the interchange will increase performance. Loop interchange is performed only at scalar optimization level 2 or higher.

The `-h nointerchange` option prevents the compiler from attempting to interchange any loops. To disable interchange of loops individually, use the `#pragma nointerchange` directive.

2.14.2 `-h scalar`*n* (`CC`, `cc`, `c99`)

Default option: `-h scalar1`

The `-h scalar`*n* option specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option (see Section 3.10, page 88).

Use one of these values for *n*:

n	Description
0	No automatic scalar optimization. The `-h matherror=errno` and `-h zeroinc` options are implied by `-h scalar0`.

`1`	Conservative automatic scalar optimization. This level implies `-h matherror=abort` and `-h nozeroinc`.
`2`	Moderate automatic scalar optimization. The scalar optimizations specified by `scalar1` are performed.
`3`	Aggressive automatic scalar optimization.

2.14.3 `-h [no]reduction` (`CC`, `cc`, `c99`)

Default option: `-h reduction`

The `-h reduction` option instructs the compiler to enable vectorization of all reduction loops. The `-h noreduction` option disables vectorization of all reduction loops. This option is affected by the `-h scalar`*n* option (see Section 2.14.2, page 27). Reduction loops and the `noreduction` directive are discussed in Section 3.10.3, page 89.

2.14.4 `-h [no]zeroinc` (`CC`, `cc`, `c99`)

Default option: `-h nozeroinc`

The `-h nozeroinc` option improves run time performance by causing the compiler to assume that constant increment variables (CIVs) in loops are not incremented by expressions with a value of 0.

The `-h zeroinc` option causes the compiler to assume that some CIVs in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code. For example, in a loop with index *i*, the expression *expr* in the statement *i* += *expr* can evaluate to 0. This rarely happens in actual code. `-h zeroinc` is the safer and slower option. This option is affected by the `-h scalar`*n* option (see Section 2.14.2, page 27).

2.15 Math Options

This section describes compiler options pertaining to math functions. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.15.1 `-h fp`*n* (`CC`, `cc`, `c99`)

The `–h fp` option offers finer control over floating-point optimizations than the `-h [no]ieeeconform` option. The *n* argument controls the level of optimization; 0 indicates minimum freedom to optimize floating-point operations, while

3 indicates maximum. The higher the optimization level, the lesser the conformance to the IEEE standard for floating point.

This option is useful for code that use unstable algorithms, but which are optimizable. It is also useful for applications that want aggressive floating-point optimizations that go beyond what the IEEE standard allows.

The `-h [no]ieeeconform` and `-h fp` options can be specified on the same compiler command line, but the compiler will use only the rightmost option. If this is the case or multiple `-h fp` are used, the compiler issues a message indicating such.

Table 2 compares the various optimization levels of the `-h fp` option (levels 2 and 3 are usually the same). The table lists some of the optimizations performed; the compiler may perform other optimizations not listed.

Table 2. Floating-point Optimization Levels

Optimization Type	`0`	`1`	`2`	`3`
Inline selected mathematical library functions	N/A	N/A	N/A	Accuracy is slightly reduced
Complex divisions accuracy and calculation speed	Accurate and slower	Accurate and slower	Less accurate (less precision) and faster	Less accurate (less precision) and faster
Exponentiation rewrite	None	Fast	Maximum performance	Maximum performance
Strength reduction	Fast	Fast	Aggressive	Aggressive
Rewrite division as reciprocal equivalent[1]	None	None	Yes	Yes
Safety	Maximum	Moderate	Moderate	Low

1 For example, `x/y` is transformed to `x * 1.0/y`.

Optimization Type	0	1	2	3
Optimizations	Same effect as `-h ieeeconform`. The `-h fp0` option causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode.[2]	Performs various, generally safe, non-conforming IEEE optimizations, such as *folding* `A == A to .TRUE.` where `A` is a floating-point object.	Includes optimizations of `-h fp1`.	Includes optimizations of `-h fp1`. Equivalent to the `-h noieeeconform` option.
When to use	The `-h fp0` and `-h fp1` options should never be used, except when your code pushes the limits of IEEE accuracy, or require strong IEEE standard conformance.	The `-h fp0` and `-h fp1` options should never be used, except when your code pushes the limits of IEEE accuracy, or require strong IEEE standard conformance.		The `-h fp3` option should be used when performance is more critical than the level of IEEE standard conformance provided by `-h fp2`.

The default is `-h fp2`.

2.15.2 `-h [no]ieeeconform` (`CC`, `cc`)

Default option: `-h noieeeconform` (equivalent to `-h fp0`)

The `-h ieeeconform` option causes the resulting executable code to conform more closely to the IEEE floating-point standard (ANSI/IEEE Std 754-1985). Use of this option disables many arithmetic identity optimizations and may result in significantly slower code.

2 When specified, many identity optimizations are disabled, executable code is slower than higher floating-point optimization levels, and a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow.

When -h noieeeconform is in effect, the compiler optimizes expressions such as x != x to 0 and x/x to 1 (where x has floating type). With the -h ieeeconform option in effect, these and other similar arithmetic identity optimizations are not performed. Optimizations on integral types are not affected by this option.

The -h ieeeconform option also turns on a scaled complex divide, which increases the range of complex values that can be handled without producing an underflow or an overflow.

2.15.3 -h matherror=*method* (CC, cc, c99)

Default option: -h matherror=abort

The -h matherror=*method* option specifies the method of error processing used if a standard math function encounters an error. The *method* argument can have one of the following values:

method	Description
abort	If an error is detected, errno is not set. Instead a message is issued and the program aborts. An exception may be raised.
errno	If an error is detected, errno is set and the math function returns to the caller. This method is implied by the -h conform, -h scalar0, -O0, -Gn, and -g options.

2.16 Debugging Options

This section describes compiler options used for debugging. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.16.1 -G *level* (CC, cc, c99) and -g (CC, cc, c89, c99)

The -g and -G *level* options enable the generation of debugging information that is used by symbolic debuggers such as TotalView. These options allow debugging with breakpoints. Table 3 describes the values for the -G option.

Table 3. `-G` *level* Definitions

level	Optimization	Breakpoints allowed on
`f`	Full	Function entry and exit
`p`	Partial	Block boundaries
`n`	None	Every executable statement

Less extensive debugging (such as full) permits greater optimization opportunities for the compiler. Debugging at any level may inhibit some optimization techniques, such as inlining.

The `-g` option is equivalent to `-Gn`. The `-g` option is included for compatibility with earlier versions of the compiler and many other UNIX systems; the `-G` option is the preferred specification. The `-Gn` and `-g` options disable all optimizations and imply `-O0`.

The debugging options take precedence over any conflicting options that appear on the command line. If more than one debugging option appears, the last one specified overrides the others.

Debugging is described in more detail in Chapter 10, page 155.

2.16.2 `-h` [`no`]`bounds` (`cc`, `c99`)

Default option: `-h nobounds`

The `-h bounds` option provides checking of pointer and array references to ensure that they are within acceptable boundaries. `-h nobounds` disables these checks.

The pointer check verifies that the pointer is greater than 0 and less than the machine memory limit. The array check verifies that the subscript is greater than or equal to 0 and is less than the array size, if declared.

2.16.3 `-h zero` (`CC`, `cc`, `c99`)

The `-h zero` option causes stack-allocated memory to be initialized to all zeros.

2.17 Compiler Message Options

This section describes compiler options that affect messages. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.17.1 `-h msglevel_`*n* (CC, cc, c99)

Default option: `-h msglevel_3`

The `-h msglevel_`*n* option specifies the lowest level of severity of messages to be issued. Messages at the specified level and above are issued. Argument *n* can be `0` (comment), `1` (note), `2` (caution), `3` (warning), or `4` (error).

2.17.2 `-h [no]message=`*n*`[:`*n*`...]` (CC, cc, c99)

Default option: Determined by `-h msglevel_`*n*

The `-h [no]message=`*n*`[:`*n*`...]` option enables or disables specified compiler messages. *n* is the number of a message to be enabled or disabled. You can specify more than one message number; multiple numbers must be separated by a colon with no intervening spaces. For example, to disable messages `CC-174` and `CC-9`, specify:

```
-h nomessage=174:9
```

The `-h [no]message=`*n* option overrides `-h msglevel_`*n* for the specified messages. If *n* is not a valid message number, it is ignored. Any compiler message except `ERROR`, `INTERNAL`, and `LIMIT` messages can be disabled; attempts to disable these messages by using the `-h nomessage=`*n* option are ignored.

2.17.3 `-h report=`*args* (CC, cc, c99)

The `-h report=`*args* option generates report messages specified in *args* and lets you direct the specified messages to a file. Use any combination of these for *args*:

args	Description
i	Generates inlining optimization messages
m	Generates multistream optimization messages
s	Generates scalar optimization messages
v	Generates vector optimization messages
f	Writes specified messages to file *file*.V where *file* is the source file specified on the command line. If the f option is not specified, messages are written to stderr.

No spaces are allowed around the equal sign (=) or any of the *args* codes. For example, the following example prints inlining and scalar optimization messages to file, myfile.c:

```
cc -h report=is myfile.c
```

2.17.4 -h [no]abort (CC, cc, c99)

Default option: -h noabort

The -h [no]abort option controls whether a compilation aborts if an error is detected.

2.17.5 -h errorlimit[=*n*] (CC, cc, c99)

Default option: -h errorlimit=100

The -h errorlimit[=*n*] option specifies the maximum number of error messages the compiler prints before it exits. *n* is a positive integer. Specifying -h errorlimit=0 disables exiting on the basis of the number of errors. Specifying -h errorlimit with no qualifier is the same as setting *n* to 1.

2.18 Compilation Phase Options

This section describes compiler options that affect compilation phases. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.18.1 -E (CC, cc, c89, c99, cpp)

If the -E option is specified on the command line (except for cpp), it executes only the preprocessor phase of the compiler. The -E and -P options are

equivalent, except that -E directs output to stdout and inserts appropriate #line preprocessing directives. The -E option takes precedence over the -h feonly, -S, and -c options.

If the -E option is specified on the cpp command line, it inserts the appropriate #line directives in the preprocessed output. When both the -P and -E options are specified, the last one specified takes precedence.

2.18.2 -P (CC, cc, c99, cpp)

When the -P option is specified on the command line (except for cpp), it executes only the preprocessor phase of the compiler for each source file specified. The preprocessed output for each source file is written to a file with a name that corresponds to the name of the source file and has .i suffix substituted for the suffix of the source file. The -P option is similar to the -E option, except that #line directives are suppressed, and the preprocessed source does not go to stdout. This option takes precedence over -h feonly, -S, and -c.

When the -P option is specified on the cpp command line, it is ignored. When both the -P and -E options are specified, the last one specified takes precedence.

2.18.3 -h feonly (CC, cc, c99)

The -h feonly option limits the Cray C and C++ compilers to syntax checking. The optimizer and code generator are not executed. This option takes precedence over -S and -c.

2.18.4 -S (CC, cc, c99)

The -S option compiles the named C or C++ source files and leaves their assembly language output in the corresponding files suffixed with a .s. If this option is used with -G or -g, debugging information is not generated. This option takes precedence over -c.

2.18.5 -c (CC, cc, c89, c99)

The -c option creates a relocatable object file for each named source file, but does not link the object files. The relocatable object file name corresponds to the name of the source file. The .o suffix is substituted for the suffix of the source file.

2.18.6 -#, -##, and -### (CC, cc, c99, cpp)

The -# option produces output indicating each phase of the compilation as it is executed. Each succeeding output line overwrites the previous line.

The -## option produces output indicating each phase of the compilation, as well as all options and arguments being passed to each phase, as they are executed.

The -### option is the same as -##, except the compilation phases are not executed.

2.18.7 -W*phase*,"*opt...*" (CC, cc, c99)

The -W*phase* option passes arguments directly to a phase of the compiling system. Table 4 shows the system phases *phase* can indicate.

Table 4. -W*phase* Definitions

phase	System phase	Command
p	Preprocessor	cpp
0	Compiler	CC,cc,c99
a	Assembler	as
l	Loader	ld

Arguments to be passed to system phases can be entered in either of two styles. If spaces appear within a string to be passed, the string is enclosed in double quotes. When double quotes are not used, spaces cannot appear in the string. Commas can appear wherever spaces normally appear; an option and its argument can be either separated by a comma or not separated. If a comma is part of an argument, it must be preceded by the \ character. For example, any of the following command lines would send -e name and -s to the loader:

```
cc -Wl,"-e name -s" file.c

cc -Wl,-e,name,-s file.c

cc -Wl,"-ename",-s file.c
```

Because the preprocessor is built into the compiler, -Wp and -W0 are equivalent.

2.18.8 -Y*phase*,*dirname* (CC, cc, c89, c99, cpp)

The -Y*phase*,*dirname* option specifies a new directory (*dirname*) from which the designated *phase* should be executed. *phase* can be one or more of the values shown in Table 5.

Table 5. -Y*phase* Definitions

phase	System phase	Command
p	Preprocessor	cpp
0	Compiler	CC,cc,c89,c89,cpp
a	Assembler	as
l	Loader	ld

Because there is no separate preprocessor, -Yp and -Y0 are equivalent. If you are using the -Y option on the cpp command line, p is the only argument for *phase* that is allowed.

2.19 Preprocessing Options

This section describes compiler options that affect preprocessing. Each subsection heading shows in parentheses the compiler command with which the option can be used in.

2.19.1 -C (CC, cc, c99, cpp)

The -C option retains all comments in the preprocessed source code, except those on preprocessor directive lines. By default, the preprocessor phase strips comments from the source code. This option is useful with cpp or in combination with the -P or -E option on the CC, cc, and c99 commands.

2.19.2 -D *macro*[=*def*] (CC, cc, c89, c99 cpp)

The -D *macro*[=*def*] option defines a macro named *macro* as if it were defined by a#define directive. If no =*def* argument is specified, *macro* is defined as 1.

Predefined macros also exist; these are described in Chapter 9, page 151. Any predefined macro except those required by the standard (see Section 9.1, page 151) can be redefined by the -D option. The -U option overrides the -D option

when the same macro name is specified regardless of the order of options on the command line.

2.19.3 `-h [no]pragma=`*name*`[: name...]` (CC, cc, c99)

Default option: `-h pragma`

The `[no]pragma=`*name*`[:`*name*`...]` option enables or disables the processing of specified directives in the source code. *name* can be the name of a directive or a word shown in Table 6 to specify a group of directives. More than one name can be specified. Multiple names must be separated by a colon and have no intervening spaces.

Table 6. `-h pragma` Directive Processing

name	Group	Directives affected
`all`	All	All directives
`allinline`	Inlining	`inline`, `noinline`
`allscalar`	Scalar optimization	`concurrent`, `nointerchange`, `noreduction`, `suppress`, `unroll`
`allvector`	Vectorization	`ivdep`, `novector`, `novsearch`, `prefervector`, `shortloop`

When using this option to enable or disable individual directives, note that some directives must occur in pairs. For these directives, you must disable both directives if you want to disable either; otherwise, the disabling of one of the directives may cause errors when the other directive is (or is not) present in the compilation unit.

2.19.4 `-I` *incldir* (CC, cc, c89, c99, cpp)

The `-I` *incldir* option specifies a directory for files named in `#include` directives when the `#include` file names do not have a specified path. Each directory specified must be specified by a separate `-I` option.

The order in which directories are searched for files named on `#include` directives is determined by enclosing the file name in either quotation marks (`""`) or angle brackets (`< and >`).

Directories for `#include "`*file*`"` are searched in the following order:

1. Directory of the input file.
2. Directories named in `-I` options, in command line order.
3. Site- and compiler release-specific include files directories.
4. Directory `/usr/include`.

Directories for `#include` *file* are searched in the following order:

1. Directories named in `-I` options, in command line order.
2. Site-specific and compiler release-specific include files directories.
3. Directory `/usr/include`.

If the `-I` option specifies a directory name that does not begin with a slash (/), the directory is interpreted as relative to the current working directory and not relative to the directory of the input file (if different from the current working directory). For example:

```
cc -I. -I yourdir mydir/b.c
```

The preceding command line produces the following search order:

1. `mydir` (`#include "`*file*`"` only).
2. Current working directory, specified by `-I`.
3. `yourdir` (relative to the current working directory), specified by `-I yourdir`.
4. Site-specific and compiler release-specific include files directories.
5. Directory `/usr/include`.

2.19.5 `-M` (`CC`, `cc`, `c99`, `cpp`)

The `-M` option provides information about recompilation dependencies that the source file invokes on `#include` files and other source files. This information is printed in the form expected by `make`. Such dependencies are introduced by the `#include` directive. The output is directed to `stdout`.

2.19.6 `-N` (`cpp`)

The `-N` option specified on the `cpp` command line enables the old style (referred to as K & R) preprocessing. If you have problems with preprocessing (especially non-C source code), use this option.

2.19.7 `-nostdinc` (`CC`, `cc`, `c89`, `c99`, `cpp`)

The `-nostdinc` option stops the preprocessor from searching for include files in the standard directories (`/usr/include/CC` and `/usr/include`).

2.19.8 `-U` *macro* (`CC`, `cc`, `c89`, `c99`, `cpp`)

The `-U` option removes any initial definition of *macro*. Any predefined macro except those required by the standard (see Section 9.1, page 151) can be undefined by the `-U` option. The `-U` option overrides the `-D` option when the same macro name is specified, regardless of the order of options on the command line.

Predefined macros are described in Chapter 9, page 151. Macros defined in the system headers are not predefined macros and are not affected by the `-U` option.

2.20 Loader Options

This section describes compiler options that affect loader tasks. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.20.1 `-l` *libfile* (`CC`, `cc`, `c89`, `c99`)

The `-l` *libfile* option identifies library files to be loaded. The given *libfile* is processed by searching for a file named `/lib`*libfile*`.a` for each different -L library *dir*. For example, if the command line includes `-L`*dir1* `-L`*dir2*`/`*subdir* `-lxyz`, then the loader will search for `libxyz.a`, first in *dir1*, then in *dir2*`/`*subdir*, and then in the remaining standard library directories.

There is no search order dependency for libraries. Default libraries are shown in the following list:

`libC.a` (Cray C++ only)

`libu.a`

`libm.a`

`libc.a`

`libsma.a`

`libf.a`

`libfi.a`

`libsci.a`

If you specify personal libraries by using the `-l` command line option, as in the following example, those libraries are added to the top of the preceding list. (The `-l` option is passed to the loader.)

```
cc -l mylib target.c
```

When the previous command line is issued, the loader looks for a library named `libmylib.a` (following the naming convention) and adds it to the top of the list of default libraries.

2.20.2 `-L` *libdir* (`CC`, `cc`, `c89`, `c99`)

The `-L` *libdir* option changes the `-l` option algorithm to search directory *libdir* before searching the default directories. If *libdir* does not begin with a slash (/), it is interpreted as relative to the current working directory.

The loader searches for library files in the compiler release-specific directories.

> **Note:** Multiple `-L` options are treated cumulatively as if all *libdir* arguments appeared on one `-L` option preceding all `-l` options. Therefore, do not attempt to load functions of the same name from different libraries through the use of alternating `-L` and `-l` options.

2.20.3 `-o` *outfile* (`CC`, `cc`, `c89`, `c99`)

The `-o` *outfile* option produces an absolute binary file named *outfile.* A file named `a.out` is produced by default. When this option is used in conjunction with the `-c` option and a single C or C++ source file, a relocatable object file named *outfile* is produced.

2.20.4 `-s` (`CC`, `cc`, `c89`, `c99`)

(Deferred implementation) The `-s` option produces executable files from which symbolic and other information not required for proper execution has been removed. If both the `-s` and `-g` (or `-G`) options are present, `-s` is ignored.

2.21 Miscellaneous Options

This section describes compiler options that affect general tasks. Each subsection heading shows in parentheses the compiler command with which the option can be used.

2.21.1 `-h command` (`cc`, `c99`)

The command mode option (`-h command`) allows you to create commands for Cray X1 systems to supplement commands developed by Cray. Your commands will execute on a single-streaming processor (SSP).

The commands created with the command mode option cannot multistream. If you want to disable vectorization, add the `-h vector0` option to the compiler command line. The compiled commands will have less debugging information, unless you specify a debugging option. The debugging information does not slow execution time, but it does result in a larger executable that may take longer to load.

For simplicity, you should use the C compiler to load your programs built with the command mode option, because the required options and libraries are automatically specified and loaded for you.

If you decide to load the libraries manually, you must use the loader command (`ld`) and specify on its command line the `-command` and `-ssp` options and the `-L` option with the path to the command mode libraries. The command mode libraries are found in the `cmdlibs` directory under the path defined by the `CRAYLIBS_SV2` environment variable. These must also be linked:

- `Start0.o`
- `libc` library
- `libm` library
- `libu` library

Programs linked with the -ssp option and -command must be compiled with the -h command option. That is, do not link object files built with the command mode option with object files that do not use the option.

The following sample command line illustrates compiling the code for a command named fierce:

```
% cc -h command -h vector0 -o fierce fierce.c
```

2.21.2 -h decomp (CC, cc, c99)

The -h decomp option decompiles (translates) the intermediate representation of the compiler into listings that resemble the format of the source code. This is performed twice, resulting in two output files, at different points during the optimization process. You can use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes you can make to your C or C++ source to improve its performance.

The compiler produces two decompilation listing files, with these extensions, per source file specified on the command line: .opt and .cg. The compiler generates the .opt file after applying most high level loop nest transformations to the code. The code structure of this listing most resembles your source code and is readable by most users. In some cases, because of optimizations, the structure of the loops and conditionals will be significantly different than the structure in your source file.

The .cg file contains a much lower level of decompilation. It is still displayed in a C or C++ like format, but is quite close to what will be produced as assembly output. This version displays the intermediate text after all multistreaming translation, vector translation, and other optimizations have been performed. An intimate knowledge of the hardware architecture of the system is helpful to understanding this listing.

The .opt and .cg files are intended as a tool for performance analysis, and are not valid C or C++ functions. The format and contents of the files can be expected to change from release to release.

The following examples show the listings generated when the -h decomp is applied to this example:

```
/* Source code, in file example.c */

double a[64], b[64], c[64];

void
```

```
example( void )
{
   long i;

   for ( i = 0; i < 64; i++ )
   {
      if ( a[i] > 0.0 )
      {
        b[i] = c[i];
      }
    }
    return;
}
```

This is the listing of the `example.opt` file after loop optimizations are performed:

```
4. void
4. example( void )
4. {
6.    @Induc01_N0 = 0;
6. #pragma ivdep
6.    do {
7.       if ( a[@Induc01_N0] > 0.0 ) {
8.          b[@Induc01_N0] = c[@Induc01_N0];
8.       }
6.       @Induc01_N0 = 1 + @Induc01_N0;
6.    } while ( @Induc01_N0 < 64 );
12.    return;
12. }
```

This is the listing of the `example.cg` file after other optimizations are performed:

```
4. void
4. example( void )
4. {
6.    vinfo( Begin_Short_Loop );
7.    $VMT_2 = _vm_gt( 0[&a:64:1].L, 0.0 );
8.    0[&b:64:1#$VMT_2].L = 0[&c:64:1#$VMT_2].L;
6.    vinfo( End_Short_Loop );
12.   return;
12.}
```

2.21.3 `-h ident=`*name* (`CC`, `cc`, `c99`)

Default option: File name specified on the command line

The `-h ident=`*name* option changes the `ident` name to *name*. This *name* is used as the module name in the object file (`.o` suffix) and assembler file (`.s` suffix). Regardless of whether the `ident` name is specified or the default name is used, the following transformations are performed on the `ident` name:

- All `.` characters in the `ident` name are changed to `$`.
- If the `ident` name starts with a number, a `$` is added to the beginning of the `ident` name.

2.21.4 `-h [no]omp` (`CC`, `cc`, `c99`, `cpp`)

(Deferred implementation) The `-h [no]omp` options enable or disable the compiler recognition of OpenMP directives. For details, see Chapter 4, page 97.

2.21.5 `-h upc`

The `-h upc` option enables compilation of Unified Parallel C (UPC) code. UPC is a C language extension for parallel program development that allows you to explicitly specify parallel programming through language syntax rather than through library functions such as are used in MPI or SHMEM.

The Cray X1 implementation of UPC is discussed in greater detail in Chapter 5, page 129.

2.21.6 `-V` (`CC`, `cc`, `c99`, `cpp`)

The `-V` option displays compiler version information. If the command line specifies no source file, no compilation occurs. Version information consists of the product name, the version number, and the current date and time, as shown in the following example:

```
% CC -V
Cray C++ Version 4.1.0.0 (u10c42004p44047s61a22e38)
08/15/02 08:53:51
```

2.21.7 `-X` *npes* (`CC`, `cc`, `c99`)

The `-X` *npes* option fixes the number of processing elements to use during execution. Specify an integer value for *npes* from 1 through 4096 inclusive.

Once set, the number of processing elements to use cannot be changed at load or run time. You must recompile the program with a different value for *npes* to change the number of processing elements.

> **Note:** Programs compiled with the `-X` option can be executed without using the `aprun` command. If the command is used, you must specify the same value specified at compile or load time.

If you use the loader on a program compiled with the `-X` option, you must specify the same value to the loader as was specified at compile time.

You can execute the compiled program without using the `aprun` command just by entering the name of the output file. If you use the command and specify the number of processing elements on the `aprun` command line, you must specify the same number to the command as was specified at compile time.

The `_num_pes` intrinsic can be used when programming UNICOS/mp systems. The value returned by `_num_pes` is equal to the number processing elements available to your program. The number of the first processing element is always 0, and the number of the last processing element is `_num_pes() - 1`. When the `-X` *npes* option is specified at compile time, the `_num_pes` intrinsic returns the value specified by the *npes* argument.

On the Cray X1 system, the `_num_pes` intrinsic can be used only in either of these situations:

- The `-X` *npes* option is specified on the command line, or
- The value of the expression containing the `_num_pes` intrinsic is not known until run time (that is, it can only be used in run time expressions)

One of the many uses for the `_num_pes` intrinsic is illustrated in the following example, which declares a variable length array of size equal to the number of processing elements:

```
int a[_num_pes()];
```

Using the `_num_pes` intrinsic in conjunction with the `-X` *npes* option allows the programmer to program the number of processing elements into a program in places that do not accept run time values. Specifying the number of processing elements at compile time can also enhance compiler optimization.

2.22 Command Line Examples

These examples illustrate a variety of command lines for the C and C++ compiler commands:

- This example compiles `myprog.C`, fixes the number of processing elements to 8, and instantiates all template entities declared or referenced in the compilation unit.

```
CC -X8 -h instantiate=all myprog.C
```

- This example compiles `myprog.C`. The `-h conform` option specifies strict conformance to the ISO C++ standard. No automatic instantiation of templates is performed.

```
CC -h conform -h noautoinstantiate myprog.C
```

- This example compiles input files `myprog.C` and `subprog.C`. Option `-c` specifies that object files `myprog.o` and `subprog.o` are produced and that the loader is not called. Option `-h inline1` instructs the compiler to inline function calls declared with the inline keyword or those declared within a class declaration.

```
CC -c -h inline1 myprog.C subprog.C
```

- This example specifies that the compiler search the current working directory (represented by a period (`.`)) for `#include` files before searching the default `#include` file locations.

```
CC -I. disc.C vend.C
```

- This example specifies that source file `newprog.c` be preprocessed only. Compilation and linking are suppressed. In addition, the macro `DEBUG` is defined.

```
cc -P -D DEBUG newprog.c
```

- This example compiles `mydata1.C`, writes object file `mydata1.o`, and produces a scalar optimization report to `stdout`.

```
CC -c -h report=s mydata1.C
```

- This example compiles `mydata3.c` and produces the executable file `a.out`. A 132-column pseudo assembly listing file is also produced in file `mydata3.L`.

```
cc -h listing mydata3.c
```

- This example compiles `myfile.C` and instructs the compiler to attempt to inline calls aggressively to functions defined within `myfile.C`. An inlining report is directed to `myfile.V`.

```
CC -h inline3,report=if myfile.C
```

2.23 Compile Time Environment Variables

These environment variables are used during compilation.

Variable	Description
`CRAYOLDCPPLIB`	Enables, when set to a nonzero value, C++ code to use these nonstandard Cray C++ headers files: • `common.h` • `complex.h` • `fstream.h` • `generic.h` • `iomanip.h` • `iostream.h` • `stdiostream.h` • `stream.h` • `strstream.h` • `vector.h` If you want to use the standard header files, your code may require modification to compile successfully. Refer to Appendix C, page 187. **Note:** Setting the `CRAYOLDCPPLIB` environment variable disables exception handling, unless you compile with the `-h exceptions` option.
`CRI_CC_OPTIONS`, `CRI_cc_OPTIONS`, `CRI_c89_OPTIONS`, `CRI_cpp_OPTIONS`	Specifies command line options that are applied to all compilations. Options specified by this environment variable are added following the options specified directly on the command line.

	This is especially useful for adding options to compilations done with build tools.
LANG	Identifies your requirements for native language, local customs, and coded character set with regard to compiler messages.
MSG_FORMAT	Controls the format in which you receive compiler messages.
NLSPATH	Specifies the message system catalogs that should be used.
NPROC	Specifies the number of processes used for simultaneous compilations. The default is 1. When more than one source file is specified on the command line, compilations may be multiprocessed by setting the environment variable NPROC to a value greater than 1. You can set NPROC to any value; however, large values can overload the system.
TARGET	(Deferred implementation) Specifies the type and characteristics of the hardware on which you are running. You can also set the TARGET environment variable to the characteristics of another system to cross-compile source code for that system.

2.24 Run Time Environment Variables

These environment variables are used during run time.

Variable	Description
CRAY_AUTO_APRUN_OPTIONS	The CRAY_AUTO_APRUN_OPTIONS environment variable specifies options for the aprun command when the command is called automatically (auto aprun). Calling the aprun command automatically occurs when only the name of the program and, where applicable, associated program options are entered on the command line; this will cause the system to automatically call aprun to run the program. The CRAY_AUTO_APRUN_OPTIONS environment variable does not specify options for the aprun command when you explicitly

specify the command on the command line, nor does it specify options for your program.

When setting options for the aprun command in the CRAY_AUTO_APRUN_OPTIONS environment variable, surround the options within double quotes and separate each option with a space. Do not use spaces between an option and its associated value. For example,

```
setenv CRAY_AUTO_APRUN_OPTIONS "-n10 -m16G"
```

If you execute a program compiled with a fixed number of processing elements (that is, the -X compiler option was specified at compile time) and the CRAY_AUTO_APRUN_OPTIONS also specifies the -n option, you must ensure that the values used for both options are the same. To do otherwise is an error.

X1_DYNAMIC_COMMON_SIZE

The X1_DYNAMIC_COMMON_SIZE sets the size of the dynamic COMMON block defined by the loader. Refer to the -LD_LAYOUT:dynamic= option in the ld(1) man page. Also refer to *Optimizing Applications on the Cray X1 System* for more information about dynamic COMMON blocks.

X1_COMMON_STACK_SIZE
X1_PRIVATE_STACK_SIZE
X1_STACK_SIZE
X1_LOCAL_HEAP_SIZE
X1_SYMMETRIC_HEAP_SIZE
X1_HEAP_SIZE
X1_PRIVATE_STACK_GAP

These environment variables allow you to change the default size of the application stacks or heaps , or consolidate the private stacks:

- X1_COMMON_STACK_SIZE, change the common stack size to the specified value.
- X1_PRIVATE_STACK_SIZE, change the private stack size to the specified value.
- X1_STACK_SIZE, set the size of the common and private stack to the specified value.

- X1_LOCAL_HEAP_SIZE, change the local heap size to the specified value.
- X1_SYMMETRIC_HEAP_SIZE, change the symmetric heap size to the specified value.
- X1_HEAP_SIZE, change the local and symmetric heap size to the specified value.
- X1_PRIVATE_STACK_GAP, consolidate, when used with X1_PRIVATE_STACK_SIZE, the four private stacks within an MSP into one segment, which frees up nontext pages for application use. The specified value, in bytes, indicates the gap to separate each stack. This gap serves as a guard region in case any of the stacks overflow.

The default size of each application stack or heap is 1 GB.

The X1_STACK_SIZE and X1_HEAP_SIZE are termed general environment variables in that they set the values for multiple stacks or heaps, respectively. The other variables in this section are termed specific because they set the value for a particular stack or heap. A specific variable overrides a general variable if both are specified as follows:

- The X1_COMMON_STACK_SIZE variable overrides the X1_STACK_SIZE variable if both are specified.
- The X1_PRIVATE_STACK_SIZE variable overrides the X1_STACK_SIZE if both are specified.
- The X1_LOCAL_HEAP_SIZE variable overrides the X1_HEAP_SIZE variable if both are specified.
- The X1_SYMMETRIC_HEAP_SIZE overrides the X1_HEAP_SIZE variable if both are specified.

The value you specify for a variable sets the size of a stack or heap in bytes. This number can be expressed as a decimal number, an octal number with a leading zero, or a hexadecimal number with a leading "0x".

If you specify a number smaller than the page size you gave to the aprun or mpirun command, the system will silently enforce a single-page minimum size. If you do not use the aprun command or do not specify a page size for aprun, the minimum

page size is set to 64 KB. Refer to the `-p text:other` option of the `aprun`(1) man page for more information about page sizes.

Using the `X1_PRIVATE_STACK_GAP` and `X1_PRIVATE_STACK_SIZE` environment variables together to consolidate the private stacks may help applications that have problems obtaining a sufficient number of large nontext pages via the `aprun` or `mpirun` commands. When the private stacks are consolidated, the pages that would have been used by the other private stacks are freed so they can be used by the application.

Each MSP used by an application uses four private stacks where each private stack occupies an integral number of pages, but if the application actually needs a private stack that is much smaller than the integral number of pages, space is wasted. In some of these cases, consolidating all four private stacks into one segment will free up the wasted space so it can be used by the application. For example, an application uses 256MB pages, which means the size of each private stack is a multiple of 256 MB. If the application only needs 60MB for each private stack, we can consolidate all four private stacks into a 256 MB page by setting `X1_PRIVATE_STACK_SIZE` to 0x3c00000 (60MB) and `X1_PRIVATE_STACK_GAP` to 0x400000 (4Mb). This packs the four private stacks into one 256MB page with a 4MB guard region between the stacks. This saves three 256MB physical pages on each MSP.

Warning: You should be aware that there is no protection against overflowing the private stacks; one private stack may corrupt another with unpredictable results if stack overflow occurs.

2.25 (Deferred implementation) OpenMP Environment Variables

This section describes the OpenMP C and C++ API environment variables that control the execution of parallel code. The names of environment variables must be uppercase. The values assigned to them are case insensitive and may have leading and trailing white space. Modifications to the values after the program has started are ignored.

The environment variables are as follows:

- `OMP_SCHEDULE` sets the run time schedule type and chunk size

- OMP_NUM_THREADS sets the number of threads to use during execution
- OMP_DYNAMIC enables or disables dynamic adjustment of the number of threads
- OMP_NESTED enables or disables nested parallelism

The examples in this section only demonstrate how these variables might be set in UNIX C shell (csh) environments:

```
setenv OMP_SCHEDULE "dynamic"
```

In Korn shell environments, the actions are similar, as follows:

```
export OMP_SCHEDULE="dynamic"
```

2.25.1 (Deferred implementation) OMP_SCHEDULE

OMP_SCHEDULE applies only to for and parallel for directives that have the schedule type runtime. The schedule type and chunk size for all such loops can be set at run time by setting this environment variable to any of the recognized schedule types and to an optional *chunk_size*.

For for and parallel for directives that have a schedule type other than runtime, OMP_SCHEDULE is ignored. The default value for this environment variable is implementation-defined. If the optional *chunk_size* is set, the value must be positive. If *chunk_size* is not set, a value of 1 is assumed, except in the case of a static schedule. For a static schedule, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.

Example:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
```

2.25.2 (Deferred implementation) OMP_NUM_THREADS

The OMP_NUM_THREADS environment variable sets the default number of threads to use during execution, unless that number is explicitly changed by calling the omp_set_num_threads library routine (see the omp_threads(3) man page) or by an explicit num_threads clause on a parallel directive.

The value of the OMP_NUM_THREADS environment variable must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction

between the OMP_NUM_THREADS environment variable and dynamic adjustment of threads, see Section 4.3, page 98.

If no value is specified for the OMP_NUM_THREADS environment variable, or if the value specified is not a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined.

Example:

```
setenv OMP_NUM_THREADS 16
```

2.25.3 (Deferred implementation) OMP_DYNAMIC

The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions unless dynamic adjustment is explicitly enabled or disabled by calling the omp_set_dynamic library routine (see the omp_threads(3) man page). Its value must be TRUE or FALSE. The default condition is FALSE.

If set to TRUE, the number of threads that are used for executing parallel regions may be adjusted by the run time environment to best utilize system resources.

If set to FALSE, dynamic adjustment is disabled.

Example:

```
setenv OMP_DYNAMIC TRUE
```

2.25.4 (Deferred implementation) OMP_NESTED

The OMP_NESTED environment variable enables or disables nested parallelism unless nested parallelism is enabled or disabled by calling the omp_set_nested library routine (see the omp_nested(3) man page). If set to TRUE, nested parallelism is enabled; if it is set to FALSE, nested parallelism is disabled. The default value is FALSE.

Example:

```
setenv OMP_NESTED TRUE
```

#pragma Directives [3]

#pragma directives are used within the source program to request certain kinds of special processing. #pragma directives are part of the C and C++ languages, but the meaning of any #pragma directive is defined by the implementation. #pragma directives are expressed in the following form:

```
#pragma [ _CRI] identifier [arguments]
```

The _CRI specification is optional and ensures that the compiler will issue a message concerning any directives that it does not recognize. Diagnostics are not generated for directives that do not contain the _CRI specification.

These directives are classified according to the following types:

- General
- Instantiation (Cray C++ only)
- Vectorization
- Scalar
- Inlining
- Multistreaming

Macro expansion occurs on the directive line after the directive name. That is, macro expansion is applied only to *arguments.*

At the beginning of each section that describes a directive, information is included about the compilers that allow the use of the directive, and the scope of the directive. Unless otherwise noted, the following default information applies to each directive:

Compiler:	Cray C and Cray C++
Scope:	Local and global

3.1 Protecting Directives

To ensure that your directives are interpreted only by the Cray C and C++ compilers, use the following coding technique in which *directive* represents the name of the directive:

```
#if _CRAYC
    #pragma _CRI directive
#endif
```

This ensures that other compilers used to compile this code will not interpret the directive. Some compilers diagnose any directives that they do not recognize. The Cray C and C++ compilers diagnose directives that are not recognized only if the `_CRI` specification is used.

3.2 Directives in Cray C++

C++ prohibits referencing undeclared objects or functions. Objects and functions must be declared prior to using them in a `#pragma` directive. This is not always the case with C.

Some `#pragma` directives take function names as arguments (for example: `#pragma weak`, `#pragma suppress`, `#pragma inline`, and `#pragma noinline`). No overloaded or member functions (no qualified names) are allowed for these directives. This limitation does not apply to the `#pragma` directives for template instantiation. This is described in Section 7.4, page 144.

3.3 Loop Directives

Many directives apply to groups. Unless otherwise noted, these directives must appear before a `for`, `while`, or `do...while` loop. These directives may also appear before a label for `if...goto` loops. If a loop directive appears before a label that is not the top of an `if...goto` loop, it is ignored.

3.4 Alternative Directive form: `_Pragma`

Compiler directives can also be specified in the following form, which has the advantage in that it can appear inside macro definitions:

```
_Pragma("_CRI identifier");
```

This form has the same effect as using the #pragma form, except that everything that appeared on the line following the #pragma must now appear inside the double quotation marks and parentheses. The expression inside the parentheses must be a single string literal, but it cannot be a macro that expands into a string literal. _Pragma is an extension to the C and C++ standards.

The following is an example using the #pragma form:

```
#pragma _CRI ivdep
```

The following is the same example using the alternative form:

```
_Pragma("_CRI ivdep");
```

In the following example, the loop automatically vectorizes wherever the macro is used:

```
#define SEARCH(A, B, KEY, SIZE, RES)         \
{                                            \
   int i;                                    \
   _Pragma("_CRI ivdep");                    \
   for (i = 0; i < (SIZE); i++)              \
      if ( (A)[ (B)[i] ] == (KEY)) break;    \
   (RES)=i;
}
```

Macros are expanded in the string literal argument for _Pragma in an identical fashion to the general specification of a #pragma directive.

3.5 General Directives

General directives specify compiler actions that are specific to the directive and have no similarities to the other types of directives. The following sections describe general directives.

3.5.1 [no]bounds Directive (Cray C Compiler)

The bounds directive specifies that pointer and array references are to be checked. The nobounds directive specifies that this checking is to be disabled.

When bounds checking is in effect, pointer references are checked to ensure that they are not 0 or are not greater than the machine memory limit. Array references are checked to ensure that the array subscript is not less than 0 or greater than or

equal to the declared size of the array. Both directives take effect starting with the next program statement in the compilation unit, and stay in effect until the next `bounds` or `nobounds` directive, or until the end of the compilation unit.

These directives have the following format:

```
#pragma _CRI bounds

#pragma _CRI nobounds
```

The following example illustrates the use of the `bounds` directive:

```
int a[30];
#pragma _CRI bounds
void f(void)
{
   int x;
   x = a[30];
   .
   .
   .
}
```

3.5.2 `duplicate` Directive (Cray C Compiler)

Scope: Global

The `duplicate` directive lets you provide additional, externally visible names for specified functions. You can specify duplicate names for functions by using a directive with one of the following forms:

```
#pragma _CRI duplicate actual as dupname...

#pragma _CRI duplicate actual as (dupname...)
```

The *actual* argument is the name of the actual function to which duplicate names will be assigned. The *dupname* list contains the duplicate names that will be assigned to the actual function. The *dupname* list may be optionally parenthesized. The word `as` must appear as shown between the *actual* argument and the comma-separated list of *dupname* arguments.

The duplicate directive can appear anywhere in the source file and it must appear in global scope. The actual name specified on the directive line must be defined somewhere in the source as an externally accessible function; the actual function cannot have a static storage class.

The following example illustrates the use of the duplicate directive:

```
#include <complex.h>

extern void maxhits(void);

#pragma _CRI duplicate maxhits as count, quantity     /* OK */

void maxhits(void)
{
   #pragma _CRI duplicate maxhits as tempcount
   /* Error: #pragma _CRI duplicate can't appear in local scope */
}

double _Complex minhits;

#pragma _CRI duplicate minhits as lower_limit
/* Error: minhits is not declared as a function */

extern void derivspeed(void);

#pragma _CRI duplicate derivspeed as accel
/* Error: derivspeed is not defined */

static void endtime(void)
{
}

#pragma _CRI duplicate endtime as limit
/* Error: endtime is defined as a static function */
```

Because duplicate names are simply additional names for functions and are not functions themselves, they cannot be declared or defined anywhere in the compilation unit. To avoid aliasing problems, duplicate names may not be referenced anywhere within the source file, including appearances on other directives. In other words, duplicate names may only be referenced from outside the compilation unit in which they are defined.

The following example references duplicate names:

```
void converter(void)
{
   structured(void);
}

#pragma _CRI duplicate converter as factor, multiplier /* OK */

void remainder(void)
{
}

#pragma _CRI duplicate remainder as factor, structured
/* Error: factor and structured are referenced in this file */
```

Duplicate names can be used to provide alternate external names for functions, as shown in the following examples.

main.c:

```
extern void fctn(void), FCTN(void);

main()
{
  fctn();
  FCTN();
}
```

fctn.c:

```
#include <stdio.h>

void fctn(void)
{
  printf("Hello world\n");
}

#pragma _CRI duplicate fctn as FCTN
```

Files `main.c` and `fctn.c` are compiled and linked using the following command line:

```
cc main.c fctn.c
```

When the executable file a.out is run, the program generates the following output:

```
Hello world
Hello world
```

3.5.3 message Directive

The message directive directs the compiler to write the message defined by *text* to stderr as a warning message. Unlike the error directive, the compiler continues after processing a message directive. The format of this directive is as follows:

```
#pragma _CRI message "text"
```

The following example illustrates the use of the message compiler directive:

```
#define FLAG 1

#ifdef FLAG
#pragma _CRI message "FLAG is Set"
#else
#pragma _CRI message "FLAG is NOT Set"
#endif
```

3.5.4 no_cache_alloc Directive

The no_cache_alloc directive is an advisory directive that specifies objects that should not be placed into the cache. Advisory directives are directives the compiler will honor if conditions permit it to. When this directive is honored, the performance of your code may be improved because the cache is not occupied by objects that have a lower cache hit rate. Theoretically, this makes room for objects that have a higher cache hit rate.

Here are some guidelines that will help you determine when to use this directive. This directive works only on objects that are vectorized. That is, other objects with low cache hit rates can still be placed into the cache. Also, you should use this directive for objects you feel should not be placed into the cache.

To use the directive, you must place it only in the specification part, before any executable statement.

This is the form of the directive:

```
#pragma no_cache_alloc base_name [, base_name] ...
```

base_name specifies the base name of the object that should not be placed into the cache. This can be the base name of any object such as an array, scalar structure, etc., without member references like C[10]. If you specify a pointer in the list, only the references, not the pointer itself, have the no cache allocate property.

3.5.5 [no]opt Directive

Scope: Global

The noopt directive disables all automatic optimizations and causes optimization directives to be ignored in the source code that follows the directive. Disabling optimization removes various sources of potential confusion in debugging. The opt directive restores the state specified on the command line for automatic optimization and directive recognition. These directives have global scope and override related command line options.

The format of these directives is as follows:

```
#pragma _CRI opt
```

```
#pragma _CRI noopt
```

The following example illustrates the use of the opt and noopt compiler directives:

```
#include <stdio.h>

void sub1(void)
{
        printf("In sub1, default optimization\n");
}

#pragma _CRI noopt
void sub2(void)
{
        printf("In sub2, optimization disabled\n");
}
#pragma _CRI opt

void sub3(void)
{
        printf("In sub3, optimization enabled\n");
}

main()
{
        printf("Start main\n");
        sub1();
        sub2();
        sub3();
}
```

3.5.6 `weak` Directive

Scope: Global

The `weak` directive specifies an external identifier that may remain unresolved throughout the compilation. A `weak` external reference can be to a function or to a data object. A `weak` external does not increase the total memory requirements of your program.

Declaring an object as a weak external directs the loader to do one of these tasks:

- Link the object only if it is already linked (that is, if a strong reference exists); otherwise, leave it is as an unsatisfied external. The loader does not display an unsatisfied external message if weak references are not resolved.
- If a strong reference is specified in the `weak` directive, resolve all weak references to it.

Note: The loader treats weak externals as unsatisfied externals, so they remain silently unresolved if no strong reference occurs during compilation. Thus, it is your responsibility to ensure that run time references to weak external names do not occur unless the loader (using some "strong" reference elsewhere) has actually loaded the entry point in question.

These are the forms of the `weak` directive:

```
#pragma _CRI weak var
```

```
#pragma
  _CRI weak sym1 = sym2
```

var	The name of an external
sym1	Defines an externally visible weak symbol
sym2	Defines an externally visible strong symbol defined in the current compilation.

The first form allows you to declare one or more weak references on one line. The second form allows you to assign a strong reference to a weak reference.

The `weak` directive must appear at global scope.

The attributes that weak externals must have depend on the form of the weak directive that you use:

- First form, weak externals must be declared, but not defined or initialized, in the source file.
- Second form, weak externals may be declared, but not defined or initialized, in the source file.
- Either form, weak externals cannot be declared with a `static` storage class.

The following example illustrates these restrictions:

```
extern long x;
#pragma _CRI weak x  /* x is a weak external data object */
extern void f(void);
#pragma _CRI weak f  /* f is a weak external function */

extern void g(void);
#pragma _CRI weak g=fun;  /* g is a weak external function
                             with a strong reference to fun */

long y = 4;
#pragma _CRI weak y  /* ERROR - y is actually defined */

static long z;
#pragma _CRI weak z  /* ERROR - z is declared static */

void fctn(void)
{
#pragma _CRI weak a  /* ERROR - directive must be at global scope */
}
```

3.5.7 `vfunction` Directive

Scope: Global

The `vfunction` directive lists external functions that use the call-by-register calling sequence. Such functions can be vectorized but must be written in Cray Assembly Language (CAL). The format of this directive is as follows:

```
#pragma _CRI vfunction func
```

The *func* variable specifies the name of the external function.

The following example illustrates the use of the `vfunction` compiler directive:

```
extern double vf(double);
#pragma _CRI vfunction vf


void f3(int n) {
    int i;
    for (i = 0; i < n; i++) {    /* Vectorized */
        b[i] = vf(c[i]);
    }
}
```

3.5.8 `ident` Directive

The `ident` directive directs the compiler to store the string indicated by *text* into the object (`.o`) file. This can be used to place a source identification string into an object file.

The format of this directive is as follows:

```
#pragma _CRI ident
   text
```

3.6 Instantiation Directives

The Cray C++ compiler recognizes three instantiation directives. Instantiation directives can be used to control the instantiation of specific template entities or sets of template entities. The following directives are described in detail in Section 7.4, page 144:

- `#pragma _CRI instantiate`
- `#pragma _CRI do_not_instantiate`
- `#pragma _CRI can_instantiate`
- The `#pragma _CRI instantiate` directive causes a specified entity to be instantiated.
- The `#pragma _CRI do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.

- The #pragma _CRI can_instantiate directive indicates that a specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

  See Chapter 7, page 139 for more information on template instantiation.

3.7 Vectorization Directives

Because vector operations cannot be expressed directly in Cray C and C++, the compilers must be capable of vectorization, which means transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures. For more information, see *Optimizing Applications on the Cray X1 System*.

The subsections that follow describe the compiler directives used to control vectorization.

3.7.1 ivdep Directive

Scope: Local

The ivdep directive tells the compiler to ignore vector dependencies for the loop immediately following the directive. Conditions other than vector dependencies can inhibit vectorization. If these conditions are satisfactory, the loop vectorizes. This directive is useful for some loops that contain pointers and indirect addressing. The format of this directive is as follows:

```
#pragma _CRI ivdep
```

The following example illustrates the use of the ivdep compiler directive:

```
p = a; q = b;
#pragma _CRI ivdep
for (i = 0; i < n; i++) {       /* Vectorized */
         *p++ = *q++;
}
```

On the Cray X1 system, the compiler assumes an infinite safe vector length; that is, any vector length can safely be used to vectorize the loop. You can use the -h [no]infinitevl compiler option to change this behavior.

3.7.2 `nopattern` Directive

Scope: Local

The `nopattern` directive disables pattern matching for the loop immediately following the directive.

The format of this directive is as follows:

```
#pragma _CRI nopattern
```

By default, the compiler detects coding patterns in source code sequences and replaces these sequences with calls to optimized library functions. In most cases, this replacement improves performance. There are cases, however, in which this substitution degrades performance. This can occur, for example, in loops with very low trip counts. In such a case, you can use the `nopattern` directive to disable pattern matching and cause the compiler to generate inline code.

In the following example, placing the `nopattern` directive in front of the outer loop of a nested loop turns off pattern matching for the matrix multiply that takes place inside the inner loop:

```
double a[100][100], b[100][100], c[100][100];

void
nopat(int n)
{
  int i, j, k;

#pragma _CRI nopattern
  for (i=0; i < n; ++i) {
    for (j = 0; j < n; ++j) {
      for (k = 0; k < n; ++k) {
        c[i][j] += a[i][k] * b[k][j]
      }
    }
  }
}
```

3.7.3 `novector` Directive

Scope: Local

The novector directive directs the compiler to not vectorize the loop that immediately follows the directive. It overrides any other vectorization-related directives, as well as the -h vector and -h ivdep command line options. The format of this directive is as follows:

```
#pragma _CRI novector
```

The following example illustrates the use of the novector compiler directive:

```
#pragma _CRI novector
for (i = 0; i < h; i++) {   /* Loop not vectorized */
    a[i] = b[i] + c[i];
}
```

3.7.4 novsearch Directive

Scope: Local

The novsearch directive directs the compiler to not vectorize the search loop that immediately follows the directive. A search loop is a loop with one or more early exit statements. It overrides any other vectorization-related directives as well as the -h vector and -h ivdep command line options. The format of this directive is as follows:

```
#pragma _CRI novsearch
```

The following example illustrates the use of the novsearch compiler directive:

```
#pragma _CRI novsearch
for (i = 0; i < h; i++) {   /* Loop not vectorized */
    if (a[i] < b[i]) break;
    a[i] = b[i];
}
```

3.7.5 prefervector Directive

Scope: Local

The prefervector directive tells the compiler to vectorize the loop that immediately follows the directive if the loop contains more than one loop in the nest that can be vectorized. The directive states a vectorization preference and does not guarantee that the loop has no memory dependence hazard.

The format of this directive is as follows:

```
#pragma _CRI prefervector
```

The following example illustrates the use of the `prefervector` directive:

```
#pragma _CRI prefervector
for (i = 0; i < n; i++) {
   #pragma _CRI ivdep
   for (j = 0; j < m; j++)
      a[i] += b[j][i];
}
```

In the preceding example, both loops can be vectorized, but the directive directs the compiler to vectorize the outer `for` loop. Without the directive and without any knowledge of `n` and `m`, the compiler vectorizes the inner `for` loop. In this example, the outer `for` loop is vectorized even though the inner `for` loop had an `ivdep` directive.

3.7.6 `safe_address` Directive

Scope: Local

The `safe_address` directive allows you to tell the compiler that it is safe to speculatively execute memory references within all conditional branches of a loop. In other words, you know that these memory references can be safely executed in each iteration of the loop.

For most code, the `safe_address` directive can improve performance significantly by preloading vector expressions. However, most loops do not require this directive to have preloading performed. The directive is only required when the safety of the operation cannot be determined or index expressions are very complicated.

The `safe_address` directive is an advisory directive. That is, the compiler may override the directive if it determines the directive is not beneficial.

If you do not use the directive on a loop and the compiler determines that it would benefit from the directive, it issues a message indicating such. The message is similar to this:

```
CC-6375 cc: VECTOR File = ctest.c, Line = 6
  A loop would benefit from "#pragma safe_address".
```

If you use the directive on a loop and the compiler determines that it does not benefit from the directive, it issues a message that states the directive is superfluous and can be removed.

To see the messages you must use the `-hreport=v` option.

Incorrect use of the directive can result in segmentation faults, bus errors, or excessive page faulting. However, it should not result in incorrect answers. Incorrect usage can result in very severe performance degradations or program aborts.

This is the syntax of the `safe_address` directive:

```
#pragma safe_address
```

In the example below, the compiler will not preload vector expressions, because the value of `j` is unknown. However, if you know that references to `b[i][j]` is safe to evaluate for all iterations of the loop, regardless of the condition, we can use the `SAFE_ADDRESS` directive for this loop as shown below:

```
void x3( double a[restrict 1000], int j )
{
    int i;
#pragma safe_address
    for ( i = 0; i < 1000; i++ ) {
        if ( a[i] != 0.0 ) {
            b[j][i] = 0.0;
        }
    }
}
```

With the directive, the compiler can load `b[i][j]` with a full vector mask, merge `0.0` where the condition is true, and store the resulting vector using a full mask.

3.7.7 `shortloop` and `shortloop128` Directives

Scope: Local

The `shortloop` and `shortloop128` directives improve performance of a vectorized loop by allowing the compiler to omit the run time test to determine whether it has been completed. The `shortloop` compiler directive identifies vector loops that execute with a maximum iteration count of 64 and a minimum iteration count of 1. The `shortloop128` compiler directive identifies vector

loops that execute with a maximum iteration count of 128 and a minimum iteration count of 1. If the iteration count is outside the range for the directive, results are unpredictable.

These directives are ignored if the loop trip count is known at compile time and is greater than the target machine's vector length. The maximum hardware vector length is 64.

The formats of these directives are as follows:

```
#pragma _CRI shortloop
#pragma _CRI shortloop128
```

The following examples illustrate the use of the `shortloop` and `shortloop128` directives:

```
#pragma _CRI shortloop
for (i = 0; i < n; i++) {  /* 1< = n < = 64 */
    a[i] = b[i] + c[i];
}


#pragma _CRI shortloop128
for (i = 0; i < n; i++) { /* 1 < = n < = 128 */
    a[i] = b[i] + c[i];
}
```

3.8 Multistreaming Processor (MSP) Directives

This section describes the multistreaming processor (MSP) optimization directives. For information about MSP compiler option, refer to Section 2.11, page 23 and for streaming intrinsics, refer to Appendix F, page 217.

The MSP directives work with the `-h stream`*n* command line option to determine whether parts of your program are optimized for the MSP. The level of streaming must be greater than 0 in order for these directives to be recognized. For more information on the `-h stream`*n* command line option, see Section 2.11.1, page 23.

The MSP `#pragma` directives are as follows:

- `#pragma nostream` (see the following section)
- `#pragma preferstream` (see Section 3.8.3, page 75)

3.8.1 `ssp_private` Directive (cc, `c99`)

The `ssp_private` directive allows the compiler to stream loops that contain function calls. By default, the compiler does not stream loops containing function calls, because the function may cause side effects that interfere with correct parallel execution. The `ssp_private` directive asserts that the specified function is free of side effects that inhibit parallelism and that the specified function, and all functions it calls, will run on an SSP.

An implied condition for streaming a loop containing a call to a function specified with the `ssp_private` directive is that the loop body must not contain any data reference patterns that prevent parallelism. The compiler can disregard an `ssp_private` directive if it detects possible loop-carried dependencies that are not directly related to a call inside the loop.

Note: The `SSP_PRIVATE` directive only affects whether or not loops are automatically streamed. It has no effect on loops within CSD parallel regions.

When using the `ssp_private` directive, you must ensure that the function called within the body of the loop follows these criteria:

- The function does not modify an object in one iteration and reference this same data in another iteration of the streamed loop.
- The function does not reference data in one iteration that is defined in another iteration.
- If the function modifies data, the iterations cannot modify data at the same storage location, unless these variables are scoped as `PRIVATE`. Following the streamed loop, the content of private variables are undefined.

  The `ssp_private` directive does not force the master thread to execute the last iteration of the streamed loop.
- If the function uses shared data that can be written to and read, you must protect it with a guard (such as the CSD `critical` directive or the lock command) or have the SSPs access the data disjointedly (where access does not overlap).
- The function calls only other routines that are capable of being called privately.
- The function calls I/O properly.

Note: The preceding list assumes that you have a working knowledge of race conditions.

To use the ssp_private directive, it must placed in the specification part, before any executable statements. This is the syntax of the ssp_private directive:

```
#pragma ssp_private PROC_NAME[, PROC_NAME] ...
```

PROC_NAME is the name of a function. Any number of ssp_private directives may be specified in a function. If a function is specified with the ssp_private directive, the function retains this attribute throughout the entire program unit. Also, the ssp_private directive is considered a declarative directive and must be specified before the start of any executable statements.

The following example demonstrates use of the ssp_private pragma:

```
/* Code in example.c */
extern void poly_eval( float *y, float x, int m, float p[m] );
#pragma _CRI ssp_private poly_eval

void example(int n, int m, float x[n], float y[n], float p[])
{
        int i;

        for (i = 0; i < n; ++i) {
                poly_eval( &y[i], x[i], m, p );
        }
}

/* Code in example poly_eval.c */
void poly_eval( float *y, float x, int m, float p[] )
{
        float result = p[m];
        int   i;

        for (i = m-1; m >= 0; --m) {
                result = x * result + p[i];
        }
        *y = result;
}
```

This example compiles the code:

```
cc -c example.c
cc -c -h gen_private_callee poly_eval.c
cc example.o poly_eval.o -o example
```

Now run the code:

```
% aprun -L1 example
```

SSP private routines are appropriate for user-specified math support functions. Intrinsic math functions, like COS are effectively SSP private routines.

3.8.2 nostream Directive

Scope: Local

The #pragma nostream directive directs the compiler to not perform MSP optimizations on the loop that immediately follows the directive. It overrides any other MSP-related directives as well as the -h stream*n* command line option.

The format of this directive is as follows:

```
#pragma _CRI nostream
```

The following example illustrates the use of the nostream directive:

```
#pragma _CRI nostream
   for ( i = 0; i < n1; i++ ) {
         x[i] = y[i] + z[i]
   }
```

3.8.3 preferstream Directive

Scope: Local

The preferstream directive tells the compiler to multistream the following loop. It can be used when one of these conditions apply:

- The compiler issues a message saying there are too few iterations in the loop to make multistreaming worthwhile.
- The compiler streams a loop in a loop nest, and you want it to stream a different eligible loop in the same nest.

The format of this directive is as follows:

```
#pragma _CRI preferstream
```

The following example illustrates the use of the `preferstream` directive:

```
   for ( j = 0; j< n2; j++ ) {
#pragma _CRI preferstream
      for ( i = 0; i < n1; i++ ) {
         a[j][i] = b[j][i] + c[j][i]
      }
   }
```

3.9 Cray Streaming Directives (CSDs)

The Cray streaming directives (CSDs) consist of 6 non-advisory directives which allow you to more closely control multistreaming for key loops in C and C++ programs. Non-advisory means that the compiler must honor these directives. The intention of these directives is not to create an additional parallel programming style or demand large effort in code development. They are meant to assist the compiler in multistreaming your program. On its own, the compiler should perform multistreaming correctly in most cases. However, if you feel that multistreaming for key loops is not occurring as you desire, then use the CSDs to override the compiler.

CSDs are modeled after the OpenMP directives and are compatible with Pthreads and all distributed-memory parallel programming models on Cray X1 systems. Multistreaming advisory directives (MSP directives) and CSDs cannot be mixed within the same block of code.

Before explaining guidelines and other issues, you need an understanding of these CSD items:

- CSD parallel regions
- CSD `parallel` defines a CSD parallel region.
- CSD `for` multistreams a `for` loop
- CSD `parallel for`, combines the CSD `parallel` and `for` directives into one directive.
- CSD `sync`, synchronizes all SSPs within an MSP
- CSD `critical`, defines a critical section of code.
- `ordered`, specifies SSPs execute in order

When you are familiar with the directives, these topics will be beneficial to you:

- Using CSDs with Cray programming models
- CSD Placement
- Protection of shared data
- Dynamic memory allocation for CSD parallel regions
- Compiler options affecting CSDs

Note: Refer to *Optimizing Applications on the Cray X1 System* for information about how to use the CSDs to optimize your code.

3.9.1 CSD Parallel Regions

CSDs are applied to a block of code (for example a loop), which will be referred to as the CSD parallel region. All CSDs must be used within this region. You must not branch into or out of the region.

Multiple CSD parallel regions can exist within a program, however, only one parallel region will be active at any given time. For example, if a parallel region calls a function containing a parallel region, the function will execute as if it did not contain a parallel region.

The CSD parallel region can contain loops and nonloop constructs, but only loops are multistreamed. Parallel execution of nonloop constructs, such as initializing variables for the targeted loop, are performed redundantly on all SSPs. Functions called from the region will be multistreamed, however you must guarantee that the function does not cause any side effects. Parallel execution of the function is independent and redundant on all SSPs, except for code blocks containing standalone CSDs. Refer to Section 3.9.9, page 85.

3.9.2 `parallel` Directive

The `parallel` directive defines the CSD parallel region, tells the compiler to multistream the region, and specifies private data objects. All other CSDs must be used within the region. You cannot place the `parallel` directive in the middle of a construct.

This is the form of the parallel directives:

```
#pragma csd parallel [private(list)] [ordered]
{
    structured_block
}   /* End of CSD parallel region */
```

The `private` clause allows you to specify data objects that are private to each SSP within the CSD parallel region; that is, each SSP has its own copy of that object and is not shared with other SSPs. The main reason for having private objects is because updating them within the CSD parallel region could cause incorrect updates because of race conditions on their addresses. The *list* argument specifies a comma separated list of objects to make private.

By default the variables used for loop indexing are assumed to be private. Other variables, unless specified in the `private` clause, are assumed to be shared.

You may need to take special steps when using private variables. If a data object existed before the parallel region is entered and the object is made private, the object may not have the same contents inside of the region as it did outside the region. The same is true when exiting the parallel region. This same object may not have the same content outside of the region as it did within the region. Therefore, if you desire that a private object keep the same value when transitioning in and out of the parallel region, copy its value to a protected shared object so you can copy it back into the private object later.

The `ordered` clause is needed if there are within the parallel region, but outside the loops within the region, any call to a function containing a CSD `ordered` directive. That is, if only the loops contain calls to functions that contain the CSD `ordered` directive, the clause is not needed. If the clause is used and there are no called functions containing a CSD `ordered` directive, the results produced by the code encapsulated by the directive will be correct, but performance of that code will be slightly degraded. If the `ordered` clause is missing and there is a called function containing a CSD `ordered` directive, your results will be incorrect. The following example shows when the `ordered` clause is needed:

```
#pragma csd parallel ordered
{
   fun();  /* fun contains ordered directive */

   for_loop_block
   . . .
}
```

The end of the CSD parallel region has an implicit barrier synchronization. The implicit barrier protects an SSP from prematurely accessing shared data.

Note: At the point of the `parallel` directive, all SSPs are enabled and are disabled at the end of the CSD parallel region.

This example shows how to use the `parallel` directive:

```
#pragma csd parallel private(jx)
{
   x = 2 * PI;   /* This line is computed on all SSPs */
   for(i=1; NN; i++)
   {
      jx = y[i] * z[i] * x;  /* jx is private to each SSP */
      ...
   }
} /* End of CSD parallel region */
```

3.9.3 CSD `for` Directive

The compiler distributes among the SSPs the iteration of `for` loops modified by the CSD `for` directive. Iterations of `for` loops not modified by the CSD `for` directives are not distributed among the SSPs, but are all redundantly executed on all SSPs.

Refer to Section 3.9.9, page 85 for placement restrictions of the CSD `for` directive.

This is the syntax of the CSD `for` directive:

```
#pragma csd for [schedule(static [, chunk_size])] [nowait] [ordered]
for_statement
{
       ...
} /* End of for loop and CSD for region */
```

The `schedule` clause specifies how the loop iterations are distributed among the SSPs. This iteration distribution is fixed (`static`) at compile time and cannot be changed by run time events.

The iteration distribution is calculated by you or the compiler. You or the compiler will divide the number of iterations into groups or *chunks*. The compiler will then statically assign the chunks to the 4 SSPs in a round-robin fashion according to iteration order (in other words, from the first iteration to the last iteration). Therefore, an SSP could have one or more chunks. The number of iterations in each chunk is called the *chunk size* which is specified by the *chunk_size* argument.

You can use these tips to calculate the chunk size:

- Balance the parallel work load across all 4 SSPs (the number of SSPs in an MSP) by dividing the number of iterations by 4. If you have a remainder, add one to the chunk size. Using 4 chunks gives you the best performance, because less overhead is incurred when using fewer chunks per SSP.

- The work load distribution among the SSPs will be imbalanced if the chunk size is greater than $1/4^{th}$ of the total number of iterations.
- If the chunk size is greater than the total number of iterations, the first SSP (SSP0) will do all the work.

The compiler calculates the iteration distribution (*chunk_size*) if the `schedule` clause or *chunk_size* argument is not specified. The value used is dependent on the conditions shown in Table 7.

Table 7. Compiler-calculated Chunk Size

Calculated chunk size	Condition
1	When a `sync`, `critical`, or `ordered` CSD directive or a function call appears in the loop.
Iterations / 4	The number of iterations are divided as evenly as possible into four chunks if these are not present in the CSD parallel region: `sync`, `critical`, or `ordered` directive or a function call.

An implicit barrier synchronization occurs at the end of the `for` region, unless the `nowait` clause is also specified. The implicit barrier protects an SSP from prematurely accessing shared data. The `nowait` clause assumes that you are guaranteeing that consumption-before-production cannot occur.

The `ordered` clause is needed if the `for` loop encapsulated by the CSD `for` directive calls any function containing a CSD `ordered` directive. If the clause is used and there are no called functions containing a CSD `ordered` directive, the results produced by the code encapsulated by the directive will be correct, but performance of that code will be slightly degraded. If the `ordered` clause is missing and there is a called function containing a CSD `ordered` directive, the results produced by the code encapsulated by the directive will be incorrect. The following example shows when the `ORDERED` clause is needed:

```
#pragma csd parallel
{
   ...

#pragma csd for ordered
   for(i=1, i<n; i++)
      fun(i) /* fun contains ordered directive */
}
```

The following examples illustrate compiler and user calculated chunk sizes. For this example, the compiler calculates the chunk size as 1, because of the function call (a chunk size of 1 causes SSP0 to perform iterations 1, 5, 9, ... , SSP1 to perform iterations 2, 6, 10, ...):

```
#pragma csd for
for(i=1; num_samples; i++)
{
   process_sample(sample[i]);
} /* End of CSD for region */
```

For this example, because there are no `sync`, `critical`, or `ordered` directives, or subprogram calls, the compiler calculates the chunk size as `(arraySize + 3) / 4)`:

```
#pragma csd for
for(i=1; arraySize; i++)
{
      product[i] = operand[i] * operand[i];
} /* End of CSD for region */
```

Adding 3 to the array size produces an optimal chunk size by grouping the maximum number of iterations into 4 chunks.

This example specifies the `schedule` clause and a chunk size of 128:

```
#pragma csd for schedule(static, 128)
for(i=1; array_size; i++)
{
   product[i] = operand[i] * operand[i];
} /* End of CSD for region */
```

In the above example, the compiler will use the chunk size based on this statement `min(array_size, 128)`. If the chunk size is larger than the array size, the compiler will use the array as the chunk size. If this is the case, then all the work will be done by SSP0.

3.9.4 `parallel for` Directive

The `parallel for` directive combines most of the functionality of the CSD `parallel` and `for` directives into one directive. The `parallel for` directive is used on a single `for` loop that contains or does not contain nested loops and is the equivalent to the following statements:

```
#pragma csd parallel [private(list)]
{
   #pragma csd for [schedule(static [, chunk])]
   for_loop_block
} /* End of CSD parallel for region */
```

The differences between the `parallel for` and its counter parts include the lack of the `nowait` clause, because it is not needed.

This is the form of the `parallel for` directive:

```
#pragma csd parallel for [private(list)] [schedule(static
[, chunk_size])]
for_statement
{
   loop_block
} /* End of CSD parallel for region */
```

For a description the `parallel for` directive, refer to the `parallel` and `for` directives at Section 3.9.2, page 77 and Section 3.9.3, page 79.

3.9.5 `sync` Directive

The `sync` directive synchronizes all SSPs within a multistreaming processor (MSP) and may under certain conditions synchronize memory with physical storage by calling `msync`. The `sync` directive is normally used where additional intra-MSP synchronization is needed to prevent race conditions caused by forced multistreaming.

The `sync` directive can appear anywhere within the CSD parallel region, even within the CSD `for` and `parallel for` directives. If the `sync` directive appears within a CSD parallel region, but outside of an enclosed CSD `for` directive, then it performs an `msync` on all four SSPs.

This example shows how to use the `sync` directive:

```
#pragma csd parallel for private(j)
{
   for(i=1; 4; i++)
   {
      for(j=1; 100000; j++)
      {
         x[j][i] = ...          /* Produce x */
```

```
      }

      #pragma csd sync

      for(j=1; 100000; j++)
      {
         ... = x[j][5-i] * ... /* Consume x */
      }
   }
}
```

The two inner loops provide a producer and consumer pair for array `x`. The `sync` directive prevents the use of the array by the second inner loop before it is completely populated.

3.9.6 `critical` Directive

The `critical` directive specifies a critical region where only one SSP at a time will execute the enclosed region.

This is the form of the `critical` directive:

```
#pragma csd critical
{
   block_of_code
} /* End of critical region */
```

This example performs a streamed sum reduction of `a` and uses the `critical` directive to calculate the complete sum:

```
sum = 0   /* Shared variable */

#pragma csd parallel private(private_sum)
{
   private_sum = 0;

   #pragma csd for
   for(i=1; a_size; i++)
   {
      private_sum = private_sum + a(i);
   }

   #pragma csd critical
   {
```

```
      sum = sum + private_sum;
   }
}
```

3.9.7 CSD `ordered` Directive

The `ordered` directive allows you to have loops with particular dependencies on other loops in the parallel region by ensuring the execution order of the SSPs. That is, SSP0 completes execution of its block of code in the ordered region before SSP1 executes that same block of code; SSP1 completes execution of that block of code before SSP2 can execute it, etc.

If the CSD `ordered` directive is placed in a function that is called from a parallel region, the CSD `parallel`, `parallel for`, or `for` directives that encapsulate the call may also need the `ordered` clause to ensure correct results. See the appropriate CSD directive for more information.

This is the format of the `ordered` directive:

```
#pragma csd ordered
{
   block_of_code
} /* End of ordered region */
```

In following example, successive iterations of the loop depend upon previous iterations, because of `a[i-1]` and `a[i-2]` on the right side of the first assignment statement. The `ordered` directive ensures that each computation of `a[i]` is complete before the next iteration (which occurs on the next SSP) uses this value as its `a[i-1]` and similarly for `a[i-2]`:

```
#pragma csd parallel for schedule(static, 1)
for(i=3; a_size; i++)
{
   #pragma csd ordered
   {
      a[i] = a[i-1] + a[i-2];
   }

   ...    /* other processing */
}
```

If the execution time for the code indicated by the `other processing` comment is larger than the time to compute the assignment within the ordered

directive, then the loop will mostly run concurrently on the 4 SSPs, even if the ordered directive is used.

3.9.8 Nested CSDs Within Cray Parallel Programming Models

CSDs can be mixed with all Cray parallel programming models within the same program on Cray X1 systems. If you nest them, the CSDs must be at the inner most level. These are the nesting levels:

1. Distributed memory models (MPI, SHMEM, UPC, and Fortran co-arrays)
2. Shared memory models (OpenMP and Pthreads)
3. Nonadvisory directives (CSDs)

If the shared or distributed memory model is used, then you can nest the CSDs within either one. These models cannot be nested within the CSDs. If both memory models are nested, then the CSDs must be nested within the distributed memory model, and the distributed memory model nested within the shared memory model.

3.9.9 CSD Placement

CSDs must be used within the CSD parallel region as defined by the `parallel` directive. Some must be used where the parallel directives are used; that is, used within the same block of code. Other CSDs can be used in the same block of code or be placed in a function and called from the parallel region (in effect, appearing as if they were within the parallel region). These CSDs will be referred to as standalone CSDs.

The CSD `for` directive is the only one that must be used within the same block of code as this example shows:

```
#pragma csd parallel
{
   ...
   #pragma csd for
   for_loop_block

   ...
}
```

The standalone CSDs are `sync`, `critical`, and `ordered`. If standalone CSDs are placed in a function and the function is not called from a parallel region, the code will execute as if no CSD code exists.

3.9.10 Protection of Shared Data

Updates to shared data by functions called from a CSD parallel region, must be protected against simultaneous access by SSPs used for the CSD parallel region. Shared data are statically allocated data objects (such as globals or data defined in a static files), dynamically allocated data objects pointed to by more than one SSP, and subprogram formal arguments where corresponding actual arguments are shared. Protecting your shared data includes using the `private` list of these CSD items: `parallel` directive, `critical` directive, or `for` loop indices.

Accesses to shared arrays made within a CSD `for` loop are implicitly private and therefore protected if the accesses use indices that involve the loop control variable of the `for` loop.

This example shows access to the `sum` shared array using loop control variable `i`:

```
...
for(i=1; array_size; i++)
{
   sum[i] = sum[i] + 1
}
```

The `critical` directive can protect writes to shared data by ensuring that only one SSP at any one time can execute the enclosed code that accesses the shared data.

Using the `for` loop indices when accessing array elements is another way to protect your shared data. Within a CSD parallel region, iterations of a `for` loop are distributed among the SSPs. This distribution can be used to divide the array among the SSPs, if the iteration of the `for` loop are used to access the array. If each SSP accesses only its portion of the array, then in a sense, that portion of the array is private to the SSP.

The following example illustrates this principle. The example performs a sum reduction on the entire shared `a` array by doing an intermediate sum reduction on all SSPs to the shared `inter_sum` vector and a final reduction on a single SSP to the `sum` scalar. The `inter_sum` array is the shared array to consider.

```
int a[SIZE1, SIZE2];
int inter_sum[SIZE2];
```

```
int sum;

#pragma csd parallel for private(inter_sum)
for(i=1; SIZE2; i++)
{
   inter_sum[i] = 0;

   for(j=i; SIZE1; j++)
   {
      inter_sum[i] = inter_sum[i] + a[j][i];
   }
}

sum = 0;

for(i=1; SIZE2; i++)
{
   sum = sum + inter_sum[i];
}
```

Although the `inter_sum` array is shared within the parallel region, the accesses to it are private, because all accesses are indexed by the loop control variable of the loop to which the CSD `for` was applied.

3.9.11 Dynamic Memory Allocation for CSD Parallel Regions

There are certain precautions you should remember as you allocate or deallocate dynamic memory for private or shared data objects.

Dynamic memory for private data objects specified by the `private` list of the `parallel` directive must be allocated and deallocated within the CSD parallel region. Dynamic memory cannot be allocated for private objects before entering the CSD parallel region and made private when within the region.

Dynamic memory can be allocated for shared data objects outside or within the CSD parallel region. If memory for the shared object is allocated or deallocated within the CSD parallel region, you must ensure that it is allocated or deallocated by only one SSP.

This example shows how to ensure that only one SSP deallocates the memory for private variable `a`:

```
...
```

```
real *a;
...

#pragma csd parallel private(a)
{
   malloc(a SIZE1);
   ...
   free(a);
}
```

3.9.12 Compiler Options Affecting CSDs

To enable CSDs, compile your code with the `-h streamn` option with *n* set to 1 or greater. Also, specify the `-h gen_private_callee` option to compile procedures called from the CSD parallel region. To disable CSDs, specify the `-h stream0` option.

3.10 Scalar Directives

This section describes the scalar optimization directives, which control aspects of code generation, register storage, and so on.

3.10.1 `concurrent` Directive

Scope: Local

The `concurrent` directive indicates that no data dependence exists between array references in different iterations of the loop that follows the directive. This can be useful for vectorization and multistreaming optimizations.

The format of the `concurrent` directive is as follows:

```
#pragma _CRI concurrent [safe_distance=n]
```

n An integer constant between 1 and 63, specifying that no dependencies exist between any iteration of the loop and *n* subsequent iterations. The `concurrent` directive is not ignored if the `safe_distance` clause is used and MSP optimizations, streaming, or vectorization is requested on the command line.

In the following example, the concurrent directive indicates that the relationship, k>=3, is true. The compiler will safely load all the array references x[i-k], x[i-k+1], x[i-k+2], and x[i-k+3] during i-th loop iteration.

```
#pragma _CRI concurrent safe_distance=3

for (i = k + 1; i < n;i++) {
  x[i] = a[i] + x[i-k]
}
```

3.10.2 nointerchange Directive

Scope: Local

The nointerchange directive inhibits the compiler's ability to interchange the loop that follows the directive with another inner or outer loop.

The format of this directive is as follows:

```
#pragma _CRI nointerchange
```

In the following example, the nointerchange directive prevents the iv loop from being interchanged by the compiler with either the jv loop or the kv loop:

```
    for (jv = 0; jm < 128; jv++) {
#pragma nointerchange
      for (iv = 0; iv < m; iv++) {
          for (kv = 0; kv < n; kv++) {
             p1[iv][jv][kv] = pw[iv][jv][kv] * s;
          }
       }
    }
```

3.10.3 noreduction Directive

Scope: Local

The noreduction compiler directive tells the compiler to not optimize the loop that immediately follows the directive as a reduction loop. If the loop is not a reduction loop, the directive is ignored.

A *reduction loop* is a loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array

elements. This involves including the result of the previous iteration in the expression of the current iteration.

You may choose to use this directive when the loop iteration count is small or when the order of evaluation is numerically significant. It overrides any vectorization-related directives as well as the `-h vector` and `-h ivdep` command line options.

The `noreduction` directive disables vectorization of any loop that contains a reduction. The specific reductions that are disabled are summation and product reductions, and alternating value computations. The directive also prevents the compiler from rewriting loops involving multiplication or exponentiation by an induction variable to be a series of additions or multiplications of a value.

Regardless of platform, however, the format of this directive is as follows:

```
#pragma _CRI noreduction
```

The following example illustrates the use of the `noreduction` compiler directive:

```
sum = 0;
#pragma _CRI noreduction
for (i = 0; i < n; i++) {
    sum += a[i];
}
```

3.10.4 `suppress` Directive

The `suppress` directive suppresses optimization in two ways, determined by its use with either global or local scope.

The global scope `suppress` directive specifies that all associated local variables are to be written to memory before a call to the specified function. This ensures that the value of the variables will always be current. The global `suppress` directive takes the following form:

```
#pragma _CRI suppress func...
```

The local scope `suppress` directive stores current values of the specified variables in memory. If the directive lists no variables, all variables are stored to memory. This directive causes the values of these variables to be reloaded

from memory at the first reference following the directive. The local suppress directive has the following format:

```
#pragma _CRI suppress [var] ...
```

The net effect of the local suppress directive is similar to declaring the affected variables to be volatile except that the volatile qualifier affects the entire program whereas the local suppress directive affects only the block of code in which it resides.

3.10.5 [no]unroll Directive

Scope: Local

The unrolling directive allows the user to control unrolling for individual loops or to specify no unrolling of a loop. Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

The format for this compiler directive is as follows:

```
#pragma _CRI [no]unroll [[n]]
```

The nounroll directive disables loop unrolling for the next loop and does not accept the integer argument *n*. The nounroll directive is equivalent to the unroll 0 and unroll 1 directives.

The *n* argument applies only to the unroll directive and specifies no loop unrolling (*n* = 0 or 1) or the total number of loop body copies to be generated ($2 \leq n \leq 63$).

If you do not specify a value for *n*, the compiler will determine the number of copies to generate based on the number of statements in the loop nest.

Caution: If placed prior to a noninnermost loop, the `unroll` directive asserts that the following loop has no dependencies across iterations of that loop. If dependencies exist, incorrect code could be generated.

The `unroll` compiler directive can be used only on loops with iteration counts that can be calculated before entering the loop. If `unroll` is specified on a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only one loop, and the innermost loop can contain work.

The compiler may do additional unrolling over the amount requested by the user.

In the following example, assume that the outer loop of the following nest will be unrolled by 2:

```
#pragma _CRI unroll 2
for (i = 0; i < 10; i++) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
      }
}
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent to each other:

```
for (i = 0; i < 10; i += 2) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
      }
      for (j = 0; j < 100; j++) {
        a[i+1][j] = b[i+1][j] + 1;
      }
}
```

The compiler then *jams*, or *fuses*, the inner two loop bodies, producing the following nest:

```
for (i = 0; i < 10; i += 2) {
      for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
        a[i+1][j] = b[i+1][j] + 1;
      }
}
```

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program.

For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between `a[i][...]` and `a[i+1][...]`:

```
/* directive will cause incorrect code due to dependencies! */
#pragma _CRI unroll 2
for (i = 0; i < 10; i++) {
      for (j = 1; j < 100; j++) {
        a[i][j] = a[i+1][j-1] + 1;
      }
}
```

3.11 Inlining Directives

Inlining replaces calls to user-defined functions with the code in the calling process that represents the function. This can improve performance by saving the expense of the function call overhead. It also enhances the possibility of additional code optimization and vectorization, especially if the function call was an inhibiting factor.

Inlining is invoked in the following ways:

- Automatic inlining of an entire compilation is enabled by issuing the `-h inline` command line option, as described in Section 2.13.1, page 26.
- Inlining of particular function calls is specified by the `inline` directive, as discussed in the following sections.

Inlining directives can appear in global scope (that is, not inside a function definition). Global inlining directives specify whether all calls to the specified functions should be inlined (`inline` or `noinline`).

Inlining directives can also appear in local scope; that is, inside a function definition. A local inlining directive applies only to the next call to the function specified on the directive. Although the function specified on an inlining directive does not need to appear in the next statement, a call to the function must occur before the end of the function definition.

Inlining directives always take precedence over the automatic inlining requested on the command line. This means that function calls that are associated with inlining directives are inlined before any function calls selected to be inlined by automatic inlining.

Note: A function that contains a variable length array argument is not currently inlined.

The `-h report=i` option writes messages identifying where functions are inlined or briefly explains why functions are not inlined.

3.11.1 `inline` Directive

The `inline` directive specifies functions that are to be inlined. The `inline` directive has the following format:

```
#pragma _CRI inline func,...
```

The *func,...* argument represents the function or functions to be inlined. The list can be enclosed in parentheses. Listed functions must be defined in the compilation unit. You cannot specify objects of type pointer-to-function.

The following example illustrates the use of the `inline` directive:

```
#include <stdio.h>
int f(int a) {
  return a*a;
}
#pragma _CRI inline f   /* Direct the compiler to inline */
                           /* calls to f. */

main() {
  int b = 5;
  printf("%d\n", f(b)); /* f is inlined here */
}
```

3.11.2 `noinline` Directive

The `noinline` directive specifies functions that are not to be inlined. The format of the `noinline` directive is as follows:

```
#pragma _CRI noinline func,...
```

The *func,...* argument represents the function or functions that are not to be inlined. The list can be enclosed in parentheses. Listed functions must be defined in the compilation unit. You cannot specify objects of type pointer-to-function.

The following example illustrates the use of the `noinline` directive:

```
#include <stdio.h>
int f(int a) {
  return a*a;
}
#pragma _CRI noinline f   /* Direct the compiler not to */
                          /* inline calls to f. */

main() {
  int b = 5;
  printf("%d\n", f(b));   /* f is not inlined here */
}
```

(Deferred implementation) OpenMP C and C++ API Directives [4]

This chapter describes the OpenMP directives that the Cray C and Cray C++ Compilers support. These directives are based on the *OpenMP C and C++ Application Program Interface Version 2.0 March 2002* standard. Copyright © 1997–2002 OpenMP Architecture Review Board.

In addition to directives, the OpenMP C and C++ API describes several run time library routines and environment variables. For information on the library routines, see the `omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages. For information on the environment variables, see Section 2.25, page 52.

The sections in this chapter are as follows:

- Using directives (Section 4.1, page 97)
- Conditional compilation (Section 4.2, page 98)
- `parallel` construct (Section 4.3, page 98)
- Work-sharing constructs (Section 4.4, page 101)
- Combined parallel work-sharing constructs (Section 4.5, page 107)
- Master and synchronization directives (Section 4.6, page 108)
- Data environment (Section 4.7, page 113)
- Directive binding (Section 4.8, page 124)
- Directive nesting (Section 4.9, page 124)
- Using the `schedule` clause (Section 4.10, page 125)

4.1 Using Directives

OpenMP directives are based on #pragma directives. Directives are case-insensitive and are of the following form:

```
#pragma omp directive-name [clause[ [,] clause]... ] new-line
```

Each directive starts with `#pragma omp`. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the #, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the `#pragma omp` are subject to macro replacement.

Directives are case sensitive. The order in which clauses appear in directives is not significant. Clauses in directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one *directive-name* can be specified per directive. For example, the following directive is not allowed:

```
/* ERROR - multiple directive names not allowed */
#pragma omp parallel barrier
```

An OpenMP directive applies to at most one succeeding statement, which must be a structured block.

4.2 Conditional Compilation

The `_OPENMP` macro is defined with value 200203 when `-h omp` is specified. This macro must not be the subject of a `#define` or a `#undef` preprocessing directive.

```
#ifdef _OPENMP
iam = omp_get_thread_num() + index;
#endif
```

For details on the `omp_get_thread_num` routine, see the `omp_threads`(3) man page.

4.3 `parallel` Construct

The following directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
        structured-block
```

The *clause* is one of the following:

- `if`*(scalar-expression)*
- `private`*(variable-list)*
- `firstprivate`*(variable-list)*
- `default(shared | none)`
- `shared`*(variable-list)*
- `copyin`*(variable-list)*
- `reduction`*(operator: variable-list)*
- `num_threads`*(integer-expression)*

When a thread encounters a parallel construct, a team of threads is created if one of the following cases is true:

- No `if` clause is present.
- The `if` expression evaluates to a nonzero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads in the team, including the master thread, execute the region in parallel. If the value of the `if` expression is zero, the region is serialized.

To determine the number of threads that are requested, the following rules will be considered in order. The first rule whose condition is met will be applied:

1. If the *num_threads* clause is present, then the value of the integer expression is the number of threads requested.
2. If the `omp_set_num_threads` library function has been called, then the value of the argument in the most recently executed call is the number of threads requested.
3. If the environment variable `OMP_NUM_THREADS` is defined, then the value of this environment variable is the number of threads requested.
4. If none of the methods above were used, then the number of threads requested is implementation-defined.

If the *num_threads* clause is present, then it supersedes the number of threads requested by the `omp_set_num_threads` library function or the `OMP_NUM_THREADS` environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

The number of threads that execute the parallel region also depends upon whether or not dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, then the requested number of threads will execute the parallel region. If dynamic adjustment is enabled, then the requested number of threads is the maximum number of threads that may execute the parallel region.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region exceeds the number that the run time system can supply, the behavior of the program is implementation defined. An implementation may, for example, interrupt the execution of the program, or it may serialize the parallel region.

The `omp_set_dynamic` library function and the `OMP_DYNAMIC` environment variable can be used to enable and disable dynamic adjustment of the number of threads.

The number of physical processors actually hosting the threads at any given time is implementation-defined. Once created, the number of threads in the team remains constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the run time system from one parallel region to another.

The statements contained within the dynamic extent of the parallel region are executed by each thread, and each thread can execute a path of statements that is different from the other threads. Directives encountered outside the lexical extent of a parallel region are referred to as orphaned directives.

There is an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region.

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team, and it becomes the master of that new team. Nested parallel regions are serialized by default. As a result, by default, a nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the run time library function `omp_set_nested` or the environment variable `OMP_NESTED`. However, the number of threads in a team that execute a nested parallel region is implementation defined.

Restrictions to the parallel directive are as follows:

- At most one `if` clause can appear on the directive.
- It is unspecified whether any side effects inside the `if` expression or `num_threads` expression occur.

- A throw executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception. Throw statements are currently not supported with parallel regions.
- Only a single `num_threads` clause can appear on the directive. The `num_threads` expression is evaluated outside the context of the parallel region, and must evaluate to a positive integer value.
- The order of evaluation of the `if` and `num_threads` clauses is unspecified.

4.4 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The sequence of work-sharing constructs and `barrier` directives encountered must be the same for every thread in a team.

OpenMP defines the following work-sharing constructs, and these are described in the sections that follow:

- `for` directive
- `sections` directive
- `single` directive

4.4.1 `for` Construct

The `for` directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the for loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the for construct is as follows:

```
#pragma omp for [clause[[,] clause] ... ] new-line
   for-loop
```

The *clause* is one of the following:

- `private`*(variable-list)*
- `firstprivate`*(variable-list)*
- `lastprivate`*(variable-list)*
- `reduction`*(operator:variable-list)*
- `ordered`
- `schedule`*(kind*[`,`*chunk_size*]*)*
- `nowait`

The `for` directive places restrictions on the structure of the corresponding `for` loop. Specifically, the corresponding `for` loop must have canonical shape:

```
for (init-expr;var logical-op b;incr-expr)
```

Where:

init-expr — One of the following:

- *var = lb*
- *integer-type var = lb*

incr-expr — One of the following:

- *++var*
- *var++*
- *–var*
- *var–*
- *var += incr*
- *var -= incr*
- *var = var + incr*
- *var = incr + var*
- *var = var - incr*

var — A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the `for`. This variable must not

	be modified within the body of the `for` statement. Unless the variable is specified `lastprivate`, its value after the loop is indeterminate.
logical-op	One of the following: • `<` • `<=` • `>` • `>=`
lb, *b*, and *incr*	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Note that the canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is performed with values in the type of *var*, after integral promotions. In particular, if the value of *b* - *lb* + *incr* cannot be represented in that type, the result is indeterminate. Further, if *logical-op* is `<` or `<=`, then *incr-expr* must cause *var* to increase on each iteration of the loop. If *logical-op* is `>` or `>=`, then *incr-expr* must cause *var* to decrease on each iteration of the loop.

The `schedule` clause specifies how iterations of the `for` loop are divided among threads of the team. The correctness of a program must not depend on which thread executes a particular iteration. The value of *chunk_size*, if specified, must be a loop invariant integer expression with a positive value. There is no synchronization during the evaluation of this expression. Thus, any evaluated side effects produce indeterminate results. The `schedule` *kind* can be one of the following:

Table 8. `schedule` clause *kind* values

`static`	When `schedule(static,` *chunk_size*`)` is specified, iterations are divided into chunks of a size specified by *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
`dynamic`	When `schedule(dynamic,` *chunk_size*`)` is specified, the iterations are divided into a series of chunks, each containing *chunk_size* iterations. Each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned. Note that the last chunk to be assigned may have a smaller number of iterations. When no *chunk_size* is specified, it defaults to 1.
`guided`	When `schedule(guided,` *chunk_size*`)` is specified, the iterations are assigned to threads in chunks with decreasing sizes. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remain. For a *chunk_size* of 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads. These sizes decrease approximately exponentially to 1. For a *chunk_size* with value *k* greater than 1, the sizes decrease approximately exponentially to *k*, except that the last chunk may have fewer than *k* iterations. When no *chunk_size* is specified, it defaults to 1.
`runtime`	When `schedule(runtime)` is specified, the decision regarding scheduling is deferred until run time. The `schedule` *kind* and size of the chunks can be chosen at run time by setting the environment variable `OMP_SCHEDULE`. If this environment variable is not set, the resulting schedule is implementation-defined. When `schedule(runtime)` is specified, *chunk_size* must not be specified.

In the absence of an explicitly defined schedule clause, the default schedule is implementation defined.

An OpenMP-compliant program should not rely on a particular schedule for correct execution. A program should not rely on a `schedule` *kind* conforming precisely to the description given above, because it is possible to have variations in the implementations of the same schedule kind across different compilers. The descriptions can be used to select the schedule that is appropriate for a particular situation.

The `ordered` clause must be present when `ordered` directives bind to the `for` construct.

There is an implicit barrier at the end of a `for` construct unless a `nowait` clause is specified.

Restrictions to the `for` directive are as follows:

- The `for` loop must be a structured block, and, in addition, its execution must not be terminated by a `break` statement.
- The values of the loop control expressions of the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a signed integer type.
- Only a single `schedule` clause can appear on a `for` directive.
- Only a single `ordered` clause can appear on a `for` directive.
- Only a single `nowait` clause can appear on a `for` directive.
- It is unspecified if or how often any side effects within the *chunk_size*, *lb*, *b*, or *incr* expressions occur.
- The value of the *chunk_size* expression must be the same for all threads in the team.

4.4.2 `sections` Construct

The `sections` directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. The syntax of the `sections` directive is as follows:

```
#pragma omp sections [clause[ [,] clause]... ] new-line

  {

      [#pragma omp section new-line]
            structured-block
      [#pragma omp section new-line
            structured-block ]
        ...
  }
```

The *clause* is one of the following:

- `private`*(variable-list)*

- `firstprivate`*(variable-list)*
- `lastprivate`*(variable-list)*
- `reduction`*(operator: variable-list)*
- `nowait`

Each section is preceded by a `section` directive, although the `section` directive is optional for the first section. The `section` directives must appear within the lexical extent of the `sections` directive. There is an implicit barrier at the end of a `sections` construct, unless a `nowait` is specified.

Restrictions to the sections directive are as follows:

- A `section` directive must not appear outside the lexical extent of the `sections` directive.
- Only a single `nowait` clause can appear on a `sections` directive.

4.4.3 `single` Construct

The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The syntax of the `single` directive is as follows:

```
#pragma omp single [clause[[,] clause] ...] new-line
            structured-block
```

The *clause* is one of the following:

- `private`*(variable-list)*
- `firstprivate`*(variable-list)*
- `copyprivate`*(variable-list)*
- `nowait`

There is an implicit barrier after the `single` construct unless a `nowait` clause is specified.

Restrictions to the `single` directive are as follows:

- Only a single `nowait` clause can appear on a `single` directive.
- The `copyprivate` clause must not be used with the `nowait` clause.

4.5 Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a `parallel` directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing constructs:

- the `parallel for` directive
- the `parallel sections` directive

4.5.1 `parallel for` Construct

The `parallel for` directive is a shortcut for a parallel region that contains only a single `for` directive. The syntax of the `parallel for` directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
        for-loop
```

This directive allows all the clauses of the `parallel` directive and the `for` directive, except the `nowait` clause, with identical meanings and restrictions. The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `for` directive.

4.5.2 `parallel sections` Construct

The `parallel sections` directive provides a shortcut form for specifying a parallel region containing only a single `sections` directive. The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `sections` directive. The syntax of the `parallel sections` directive is as follows:

```
#pragma omp parallel sections [clause[[,] clause] ...] new-line
        {
        [#pragma omp section new-line]
             structured-block
        [#pragma omp section new-line
             structured-block ]
        ...
        }
```

The *clause* can be one of the clauses accepted by the `parallel` and `sections` directives, except the `nowait` clause.

4.6 Master and Synchronization Directives

The following sections describe the:

- `master` construct
- `critical` construct
- `barrier` directive
- `atomic` construct
- `flush` directive
- `ordered` construct

4.6.1 `master` Construct

The `master` directive identifies a construct that specifies a structured block that is executed by the master thread of the team. The syntax of the `master` directive is as follows:

```
#pragma omp master new-line
            structured-block
```

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the `master` construct.

4.6.2 `critical` Construct

The `critical` directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the `critical` directive is as follows:

```
#pragma omp critical [(name)] new-line
            structured-block
```

An optional name may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed `critical` directives map to the same unspecified name.

4.6.3 `barrier` Directive

The `barrier` directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. The syntax of the `barrier` directive is as follows:

```
#pragma omp barrier new-line
```

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the `barrier` directive in parallel.

Note that because the `barrier` directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The example below illustrates these restrictions.

```
/* ERROR - The barrier directive cannot be the immediate
 * substatement of an if statement
 */
if (x!=0)
        #pragma omp barrier
...
/* OK - The barrier directive is enclosed in a
 * compound statement.
 */
if (x!=0) {
        #pragma omp barrier
}
```

4.6.4 `atomic` Construct

The `atomic` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of the `atomic` directive is as follows:

```
#pragma omp atomic new-line
      expression-stmt
```

The expression statement must have one of the following forms:

- *x binop= expr*
- *x*++
- ++*x*
- *x*–
- –*x*

In the preceding expressions:

- *x* is an lvalue expression with scalar type
- *expr* is an expression with scalar type, and it does not reference the object designated by *x*
- *binop* is not an overloaded operator and is one of +, *, -, /, &, ^, |, <<, or >>

Although it is implementation-defined whether an implementation replaces all `atomic` directives with `critical` directives that have the same unique name, the `atomic` directive permits better optimization. Often hardware instructions are available that can perform the atomic update with the least overhead.

Only the load and store of the object designated by *x* are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location in parallel should be protected with the `atomic` directive, except those that are known to be free of race conditions.

Restrictions to the `atomic` directive are as follows:

- All atomic references to the storage location *x* throughout the program are required to have a compatible type

Examples:

```
extern float a[], *p = a, b;
/* Protect against races among multiple updates. */
#pragma omp atomic
a[index[i]] += b;
/* Protect against races with updates through a. */
#pragma omp atomic
p[i] -= 1.0f;

extern union {int n; float x;} u;
/* ERROR - References through incompatible types. */
#pragma omp atomic
u.n++;
#pragma omp atomic
u.x -= 1.0f;
```

4.6.5 `flush` Directive

The `flush` directive, whether explicit or implied, specifies a *cross-thread* sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory. This means that previous evaluations of expressions that reference those objects are complete and subsequent evaluations have not yet begun. For example, compilers must restore the values of the objects from registers to memory, and hardware may need to flush write buffers to memory and reload the values of the objects from memory.

The syntax of the `flush` directive is as follows:

```
#pragma omp flush [(variable-list)] new-line
```

If the objects that require synchronization can all be designated by variables, then those variables can be specified in the optional *variable-list*. If a pointer is present in the *variable-list*, the pointer itself is flushed, not the object the pointer refers to.

A `flush` directive without a *variable-list* synchronizes all shared objects except inaccessible objects with automatic storage duration. (This is likely to have more overhead than a `flush` with a *variable-list*.) A `flush` directive without a *variable-list* is implied for the following directives:

- `barrier`
- At entry to and exit from `critical`

- At entry to and exit from `ordered`
- At entry to and exit from `parallel`
- At exit from `for`
- At exit from `sections`
- At exit from `single`
- At entry to and exit from `parallel for`
- At entry to and exit from `parallel sections`

The directive is not implied if a `nowait` clause is present. It should be noted that the `flush` directive is not implied for any of the following:

- At entry to `for`
- At entry to or exit from `master`
- At entry to `sections`
- At entry to `single`

A reference that accesses the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the previous sequence point. A reference that modifies the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the subsequent sequence point.

Note that because the `flush` directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The example below illustrates these restrictions.

```
/* ERROR - The flush directive cannot be the immediate
 * substatement of an if statement.
 */
if (x!=0)
        #pragma omp flush (x)
...
/* OK - The flush directive is enclosed in a
 * compound statement
 */
if (x!=0) {
        #pragma omp flush (x)
}
```

Restrictions to the `flush` directive are as follows:

- A variable specified in a `flush` directive must not have a reference type.

4.6.6 `ordered` Construct

The structured block following an `ordered` directive is executed in the order in which iterations would be executed in a sequential loop. The syntax of the `ordered` directive is as follows:

```
#pragma omp ordered new-line
      structured-block
```

An `ordered` directive must be within the dynamic extent of a `for` or `parallel for` construct. The `for` or `parallel for` directive to which the `ordered` construct binds must have an `ordered` clause specified as described in Section 4.4.1, page 101. In the execution of a `for` or `parallel for` construct with an `ordered` clause, `ordered` constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

There is one restriction to the `ordered` directive. An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive.

4.7 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of parallel regions, as follows:

- A `threadprivate` directive (see Section 4.7.1, page 113) is provided to make filescope, namespace-scope, or static block-scope variables local to a thread.
- Clauses that may be specified on the directives to control the sharing attributes of variables for the duration of the parallel or work-sharing constructs are described in Section 4.7.2, page 115.

4.7.1 `threadprivate` Directive

The `threadprivate` directive makes the named file-scope, namespace-scope, or static block-scope variables specified in the *variable-list* private to a thread. *variable-list* is a comma-separated list of variables that do not have an incomplete type. The syntax of the `threadprivate` directive is as follows:

```
#pragma omp threadprivate(variable-list) new-line
```

Each copy of a `threadprivate` variable is initialized once, at an unspecified point in the program prior to the first reference to that copy, and in the usual manner (that is, as the master copy would be initialized in a serial execution of the program). Note that if an object is referenced in an explicit initializer of a `threadprivate` variable, and the value of the object is modified prior to the first reference to a copy of the variable, then the behavior is unspecified.

As with any private variable, a thread must not reference another thread's copy of a `threadprivate` object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

After the first parallel region executes, the data in the `threadprivate` objects is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads remains unchanged for all parallel regions.

The restrictions to the `threadprivate` directive are as follows:

- A `threadprivate` directive for file-scope or namespace-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a `threadprivate` directive at file or namespace scope must refer to a variable declaration at file or namespace scope that lexically precedes the directive.
- A `threadprivate` directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a `threadprivate` directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a `threadprivate` directive in one translation unit, it must be specified in a `threadprivate` directive in every translation unit in which it is declared.
- A `threadprivate` variable must not appear in any clause except the `copyin`, `copyprivate`, `schedule`, `num_threads`, or the `if` clause.
- The address of a `threadprivate` variable is not an address constant.

- A `threadprivate` variable must not have an incomplete type or a reference type.
- A `threadprivate` variable with non-POD class type must have an accessible, unambiguous copy constructor if it is declared with an explicit initializer.

The following example illustrates how modifying a variable that appears in an initializer can cause unspecified behavior, and also how to avoid this problem by using an auxiliary object and a copy-constructor.

```
int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
    x++;
    #pragma omp parallel for
    /* In each thread:
     * Object a is constructed from x (with value 1 or 2?)
     * Object b is copy-constructed from b_aux
     */
    for (int i=0; i<n; i++) {
        g(a, b); /* Value of a is unspecified. */
    }
}
```

4.7.2 Data-Sharing Attribute Clauses

Several directives accept clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described with the directive.

If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a sharing attribute clause or `threadprivate` directive, then the variable is shared. Static variables declared within the dynamic extent of a parallel region are shared. Heap allocated memory (for example, using `malloc()` in C or C++ or the `new` operator in C++) is shared. (The pointer to this memory, however, can be either private or shared.) Variables

with automatic storage duration declared within the dynamic extent of a parallel region are private.

Most of the clauses accept a *variable-list* argument, which is a comma-separated list of variables that are visible. If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, the behavior is undefined.

All variables that appear within directive clauses must be visible. Clauses may be repeated as needed, but no variable may be specified in more than one clause, except that a variable can be specified in both a `firstprivate` and a `lastprivate` clause.

The following sections describe the data-sharing attribute clauses:

- `private`, (Section 4.7.2.1, page 116)
- `firstprivate`, (Section 4.7.2.2, page 117)
- `lastprivate`, (Section 4.7.2.3, page 118)
- `shared`, (Section 4.7.2.4, page 118)
- `default`, (Section 4.7.2.5, page 119)
- `reduction`, (Section 4.7.2.6, page 120)
- `copyin`, (Section 4.7.2.7, page 123)
- `copyprivate`, (Section 4.7.2.8, page 123)

4.7.2.1 `private`

The `private` clause declares the variables in *variable-list* to be private to each thread in a team. The syntax of the `private` clause is as follows:

```
private(variable-list)
```

The behavior of a variable specified in a `private` clause is as follows. A new object with automatic storage duration is allocated for the construct. The size and alignment of the new object are determined by the type of the variable. This allocation occurs once for each thread in the team, and a default constructor is invoked for a class object if necessary; otherwise the initial value is indeterminate. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within

the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

In the lexical extent of the directive construct, the variable references the new private object allocated by the thread.

The restrictions to the `private` clause are as follows:

- A variable with a class type that is specified in a `private` clause must have an accessible, unambiguous default constructor.
- A variable specified in a `private` clause must not have a `const`-qualified type unless it has a class type with a `mutable` member.
- A variable specified in a `private` clause must not have an incomplete type or a reference type.
- Variables that appear in the `reduction` clause of a `parallel` directive cannot be specified in a `private` clause on a work-sharing directive that binds to the parallel construct.

4.7.2.2 `firstprivate`

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `firstprivate` clause is as follows:

```
firstprivate(variable-list)
```

Variables specified in *variable-list* have `private` clause semantics, as described in Section 4.7.2.1, page 116. The initialization or construction happens as if it were done once per thread, prior to the thread's execution of the construct. For a `firstprivate` clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it. For a `firstprivate` clause on a work-sharing construct, the initial value of the new private object for each thread that executes the work-sharing construct is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct. In addition, for C++ objects, the new private object for each thread is copy constructed from the original object.

The restrictions to the `firstprivate` clause are as follows:

- A variable specified in a `firstprivate` clause must not have an incomplete type or a reference type.

- A variable with a class type that is specified as `firstprivate` must have an accessible, unambiguous copy constructor.
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive cannot be specified in a `firstprivate` clause on a work-sharing directive that binds to the parallel construct.

4.7.2.3 `lastprivate`

The `lastprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `lastprivate` clause is as follows:

```
lastprivate(variable-list)
```

Variables specified in the *variable-list* have `private` clause semantics. When a `lastprivate` clause appears on the directive that identifies a work-sharing construct, the value of each `lastprivate` variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable's original object. Variables that are not assigned a value by the last iteration of the `for` or `parallel for`, or by the lexically last section of the `sections` or `parallel sections` directive, have indeterminate values after the construct. Unassigned subobjects also have an indeterminate value after the construct.

The restrictions to the `lastprivate` clause are as follows:

- All restrictions for `private` apply
- A variable with a class type that is specified as `lastprivate` must have an accessible, unambiguous copy assignment operator
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive cannot be specified in a `lastprivate` clause on a work-sharing directive that binds to the parallel construct

4.7.2.4 `shared`

This clause shares variables that appear in the *variable-list* among all the threads in a team. All threads within a team access the same storage area for `shared` variables. The syntax of the `shared` clause is as follows:

```
shared(variable-list)
```

4.7.2.5 default

The default clause allows the user to affect the data-sharing attributes of variables. The syntax of the default clause is as follows:

```
default(shared | none)
```

Specifying default(shared) is equivalent to explicitly listing each currently visible variable in a shared clause, unless it is threadprivate or const-qualified. In the absence of an explicit default clause, the default behavior is the same as if default(shared) were specified.

Specifying default(none) requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct:

- The variable is explicitly listed in a data-sharing attribute clause of a construct that contains the reference
- The variable is declared within the parallel construct
- The variable is threadprivate
- The variable has a const-qualified type
- The variable is the loop control variable for a for loop that immediately follows a for or parallel for directive, and the variable reference appears inside the loop

Specifying a variable on a firstprivate, lastprivate, or reduction clause of an enclosed directive causes an implicit reference to the variable in the enclosing context. Such implicit references are also subject to the requirements listed above.

Only a single default clause may be specified on a parallel directive.

A variable's default data-sharing attribute can be overridden by using the private, firstprivate, lastprivate, reduction, and shared clauses, as demonstrated by the following example:

```
#pragma omp parallel for default(shared)
   firstprivate(i) private(x) private(r) lastprivate(i)
```

4.7.2.6 `reduction`

This clause performs a reduction on the scalar variables that appear in *variable-list*, with the operator *op*. The syntax of the `reduction` clause is as follows:

```
reduction(op:variable-list)
```

A reduction is typically specified for a statement with one of the following forms:

x = x op expr
x binop= expr
x = expr op x (except for subtraction)
x++
++*x*
x–
–*x*

where:

x	One of the reduction variables specified in the *list*
variable-list	A comma-separated list of scalar reduction variables
expr	An expression with scalar type that does not reference *x*
op	Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, `|`, `&&`, or `||`
binop	Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, or `|`

The following is an example of the `reduction` clause:

```
#pragma omp parallel for reduction(+: a, y) reduction(||: am)
for (i=0; i<n; i++) {
    a += b[i];
    y = sum(y, c[i]);
    am = am || b[i] == c[i];
}
```

As shown in the example, an operator may be hidden inside a function call. The user should be careful that the operator specified in the `reduction` clause matches the reduction operation.

Although the right operand of the || operator has no side effects in this example, they are permitted, but should be used with care. In this context, a side effect that is guaranteed not to occur during sequential execution of the loop may occur during parallel execution. This difference can occur because the order of execution of the iterations is indeterminate.

The operator is used to determine the initial value of any private variables used by the compiler for the reduction and to determine the finalization operator. Specifying the operator explicitly allows the reduction statement to be outside the lexical extent of the construct. Any number of `reduction` clauses may be specified on the directive, but a variable may appear in at most one `reduction` clause for that directive.

A private copy of each variable in *variable-list* is created, one for each thread, as if the `private` clause had been used. The private copy is initialized according to the operator (see Table 9, page 122).

At the end of the region for which the `reduction` clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler may freely reassociate the computation of the final value. (The partial results of a subtraction reduction are added to form the final value.)

The value of the original object becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. Normally, the computation will be complete at the end of the construct; however, if the `reduction` clause is used on a construct to which `nowait` is also applied, the value of the original object remains indeterminate until a barrier synchronization has been performed to ensure that all threads have completed the `reduction` clause.

The following table lists the operators that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 9. Private Copy Initialization

Operator	Initialization
+	0
*	1
–	0
&	–0
|	0
^	0
&&	1
||	0

The restrictions to the `reduction` clause are as follows:

- The type of the variables in the `reduction` clause must be valid for the reduction operator except that pointer types and reference types are never permitted.
- A variable that is specified in the `reduction` clause must not be `const`-qualified.
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive cannot be specified in a `reduction` clause on a work-sharing directive that binds to the parallel construct.

```
#pragma omp parallel private(y)
{ /* ERROR - private variable y cannot be specified
             in a reduction clause */
  #pragma omp for reduction(+: y)
  for (i=0; i<n; i++)
    y += b[i];
}

/* ERROR - variable x cannot be specified in both
           a shared and a reduction clause */
#pragma omp parallel for shared(x) reduction(+: x)
```

4.7.2.7 `copyin`

The `copyin` clause provides a mechanism to assign the same value to `threadprivate` variables for each thread in the team executing the parallel region. For each variable specified in a `copyin` clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region. The syntax of the `copyin` clause is as follows:

```
copyin(variable-list)
```

The restrictions to the `copyin` clause are as follows:

- A variable that is specified in the `copyin` clause must have an accessible, unambiguous copy assignment operator.
- A variable that is specified in the `copyin` clause must be a `threadprivate` variable.

4.7.2.8 `copyprivate`

The `copyprivate` clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The `copyprivate` clause can appear only on the `single` directive.

The syntax of the `copyprivate` clause is as follows:

```
copyprivate(variable-list)
```

The effect of the `copyprivate` clause on the variables in its *variable-list* occurs after the execution of the structured block associated with the `single` construct, and before any of the threads in the team have left the barrier at the end of the construct. Then, in all other threads in the team, for each variable in the *variable-list*, that variable becomes defined (as if by assignment) with the value of the corresponding variable in the thread that executed the construct's structured block.

Restrictions to the `copyprivate` clause are as follows:

- A variable that is specified in the `copyprivate` clause must not appear in a `private` or `firstprivate` clause for the same `single` directive.
- If a `single` directive with a `copyprivate` clause is encountered in the dynamic extent of a parallel region, all variables specified in the `copyprivate` clause must be private in the enclosing context.
- A variable that is specified in the `copyprivate` clause must have an accessible unambiguous copy assignment operator.

4.8 Directive Binding

Dynamic binding of directives must adhere to the following rules:

- The `for`, `sections`, `single`, `master`, and `barrier` directives bind to the dynamically enclosing `parallel`, if one exists, regardless of the value of any `if` clause that may be present on that directive. If no parallel region is currently being executed, the directives are executed by a team composed of only the master thread.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `atomic` directive enforces exclusive access with respect to `atomic` directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest dynamically enclosing `parallel`.

4.9 Directive Nesting

Dynamic nesting of directives must adhere to the following rules:

- A `parallel` directive dynamically inside another `parallel` logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- `for`, `sections`, and `single` directives that bind to the same `parallel` are not allowed to be nested inside each other.
- `critical` directives with the same *name* are not allowed to be nested inside each other. Note this restriction is not sufficient to prevent deadlock.

- `for`, `sections`, and `single` directives are not permitted in the dynamic extent of `critical`, `ordered`, and `master` regions if the directives bind to the same `parallel` as the regions.
- `barrier` directives are not permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master`, and `critical` regions if the directives bind to the same `parallel` as the regions.
- `master` directives are not permitted in the dynamic extent of `for`, `sections`, and `single` directives if the `master` directives bind to the same `parallel` as the work-sharing directives.
- `ordered` directives are not allowed in the dynamic extent of `critical` regions if the directives bind to the same `parallel` as the regions.
- Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed by a team composed of only the master thread.

4.10 Using the `schedule` Clause

A parallel region has at least one barrier, at its end, and may have additional barriers within it. At each barrier, the other members of the team must wait for the last thread to arrive. To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time. If some of that shared work is contained in `for` constructs, the `schedule` clause can be used for this purpose.

When there are repeated references to the same objects, the choice of schedule for a `for` construct may be determined primarily by characteristics of the memory system, such as the presence and size of caches and whether memory access times are uniform or nonuniform. Such considerations may make it preferable to have each thread consistently refer to the same set of elements of an array in a series of loops, even if some threads are assigned relatively less work in some of the loops. This can be done by using the `static` schedule with the same bounds for all the loops. In the following example, note that zero is used as the lower bound in the second loop, even though `k` would be more natural if the schedule were not important.

```
#pragma omp parallel
{
#pragma omp for schedule(static)
   for(i=0; i<n,i++)
```

```
     a[i] = work1(i);
#pragma omp for schedule(static)
  for(i=0; i<n, i++)
    if(i>=k) a[i] += work2(i);
}
```

In the remaining examples, it is assumed that memory access is not the dominant consideration, and, unless otherwise stated, that all threads receive comparable computational resources. In these cases, the choice of schedule for a `for` construct depends on all the shared work that is to be performed between the nearest preceding barrier and either the implied closing barrier or the nearest subsequent barrier, if there is a `nowait` clause. For each kind of schedule, a short example shows how that schedule kind is likely to be the best choice. A brief discussion follows each example.

The `static` schedule is also appropriate for the simplest case, a parallel region containing a single `for` construct, with each iteration requiring the same amount of work.

```
#pragma omp parallel for schedule(static)
for(i=0; i>n; i++){
  invariant_amount_of_work(i);
}
```

The `static` schedule is characterized by the properties that each thread gets approximately the same number of iterations as any other thread, and each thread can independently determine the iterations assigned to it. Thus no synchronization is required to distribute the work, and, under the assumption that each iteration requires the same amount of work, all threads should finish at about the same time.

For a team of *p* threads, let *ceiling(n/p)* be the integer *q*, which satisfies *n = p*q - r* with *0 <= r < p*. One implementation of the `static` schedule for this example would assign *q* iterations to the first *p–1* threads, and *q-r* iterations to the last thread. Another acceptable implementation would assign *q* iterations to the first *p-r* threads, and *q-1* iterations to the remaining *r* threads. This illustrates why a program should not rely on the details of a particular implementation.

The `dynamic` schedule is appropriate for the case of a `for` construct with the iterations requiring varying, or even unpredictable, amounts of work.

```
#pragma omp parallel for schedule(dynamic)
  for(i=0; i>n; i++) {
    unpredictable_amount_of_work(i);
}
```

The `dynamic` schedule is characterized by the property that no thread waits at the barrier for longer than it takes another thread to execute its final iteration. This requires that iterations be assigned one at a time to threads as they become available, with synchronization for each assignment. The synchronization overhead can be reduced by specifying a minimum chunk size *k* greater than 1, so that threads are assigned *k* at a time until fewer than *k* remain. This guarantees that no thread waits at the barrier longer than it takes another thread to execute its final chunk of (at most) *k* iterations.

4.11 Compiling Code for OpenMP

These Cray C and Cray C++ Compiler options enable or disable the compiler recognition of OpenMP directives:

- Enable OpenMP directive recognition: `-h omp`
- Disable OpenMP directive recognition: `-h noomp`

4.12 Cray Implementation Differences

The Cray C and Cray C++ implementation of OpenMP differs slightly from the *OpenMP C and C++ Application Program Interface Version 2.0 March 2002* in the following areas:

- The use of `throw` statements in parallel regions is not supported.
- `Threadprivate` variables may not have static storage class.
- Nesting of parallel regions is not supported. Nested parallel directives will result in the inner directive being ignored and that code being executed in serial.
- Restrictions for `default` (`none`) sharing listed in Section 4.7.2.5, page 119 are currently not enforced.

Cray Unified Parallel C (UPC) [5]

Unified Parallel C (UPC) is a C language extension for parallel program development. UPC allows you to explicitly specify parallel programming through language syntax rather than library functions such as used in MPI and SHMEM by allowing you to read and write memory of other processes with simple assignment statements. Program synchronization occurs only when you say so, because there is no implied synchronization. These methods map very well onto the Cray X1 systems and enable users to achieve high performance.

UPC allows you to maintain a view of your program as a collection of threads operating in a common global address space without burdening you with details of how parallelism is implemented on the machine (for example, as shared memory or as a collection of physically distributed memories).

UPC data objects are private to a single thread or shared among all threads of execution. Each thread has a unique memory space that holds its private data objects, and access to a globally shared memory space that is distributed across the threads. Thus, every part of a shared data object has an affinity to a single thread.

Cray UPC is compatible with SHMEM, Cray Fortran co-arrays, and MPI.

> **Note:** Currently, the UPC model does not define an I/O model. Therefore, you must supply the controls as needed to remove race conditions. File I/O under UPC is very similar to standard C because one thread opens a file and shares the file handle, and multiple threads may read or write to the same file.

The remainder of this information assumes that you are familiar with UPC and will therefore continue with the differences between the published UPC Introduction and Language Specification paper and the current UPC specification, differences between the UPC draft and the implementation of UPC by Cray, and compilation of your UPC code on the Cray X1 system.

If you are not familiar with UPC, refer to the UPC home page at `http://upc.gwu.edu/`and find under the Publications link the *Introduction to UPC and Language Specification* paper. This paper is slightly outdated, but contains valuable information about understanding and using UPC.

The UPC home page also contains, under the Documentation link, the *UPC Language Specification* paper, which is up to date. This paper assumes that you are familiar with the terminology and methodology for defining the C programming language. For your convenience, we have documented the

differences between the *Introduction to UPC and Language Specification* and the *UPC Language Specification* here.

After familiarizing yourself with UPC, refer to this chapter about Cray UPC:

- Changes to UPC specification (Section 5.1, page 130)
- Cray implementation differences (Section 5.2, page 131)
- Compiling and executing UPC code Section 5.3, page 134

For more information about improving UPC code performance, refer to *Optimizing Applications on the Cray X1 System*.

5.1 Changes to UPC Specification

Since the publication of the *UPC Introduction and Language Specification* paper in 1999, the UPC working group altered or added to the UPC specifications. These modifications are reflected in the *UPC Language Specification* paper version 1.0. These components of UPC were changed or added:

- Changed the memory consistency pragmas:
  - Replaced `#pragma upc strict global` and `#pragma upc relaxed global` with `#pragma upc strict` and `#pragma upc relaxed`, respectively. No change in functionality
  - `#pragma upc strict next` and `#pragma upc relaxed next` were removed
- Replaced barrier functions with barrier statements of the same name, as Table 10 shows.

Table 10. Barrier Function Replacements

Barrier function	Corresponding barrier statement
`upc_barrier();`	`upc_barrier;`
`upc_notify();`	`upc_notify;`
`upc_wait();`	`upc_wait;`
`upc_fence();`	`upc_fence;`

- Added a new blocking factor specifier `[*]`, which distributes a shared array among all threads so that each thread has only one block.
- Added new built-in memory functions:
  - `upc_global_exit`
  - `upc_phaseof`
  - `upc_addrfield`
  - `upc_global_lock_alloc`
  - `upc_all_lock_alloc`
  - `upc_memcpy`
  - `upc_memget`
  - `upc_memput`

5.2 Cray Implementation Differences

Implementation, by Cray, of the UPC draft as described by the *UPC Language Specification* paper differs slightly in the following areas:

- Declaration of shared array dimensions and blocking sizes must follow Cray-defined requirements
- Declaration of pointers to shared types have a maximum blocking size of 1
- Implementation of some UPC memory functions are deferred
- Implementation of the `upc_forall` statement is deferred

5.2.1 Requirements for Declaration of Shared Arrays Dimensions and Blocking Sizes

The implementation of shared arrays conforms to the UPC draft, but currently, Cray imposes certain requirements for defining their dimensions. This requirement is dependent upon the value of `THREADS` and the blocking factor used by the object. Declarations that stray from the requirements will cause a compile-time error.

These are the requirements for defining the dimensions of your shared arrays:

- If a shared array has a blocking factor of 1, its rightmost dimension must be an exact multiple of `THREADS` as these examples show:

```
/* Legal, because the blocking size is 1
   and rightmost dimension is an exact multiple of THREADS */
shared int a[10][THREADS];

/* Legal because blocking size is 1 and rightmost dimension
   is an exact multiple of THREADS. Equivalent is
   a[10][10][THREADS] */
shared int a[10][10*THREADS];

/* Legal if THREADS is defined at compile time as an integral
   divisor of the rightmost extent (100).
   (refer to the -X npes option )*/
shared int a[10][100];
```

- If a shared array has a blocking factor of `[*]`, its leftmost dimension extent must be an exact multiple of `THREADS` as this example shows:

```
/* shared array of ints with one b[5] per thread.
   blocking size is 5 */
shared[*] int b[THREADS][5];
```

- For all other finite blocking sizes, the block size must be the product of the extents of all but the leftmost dimension. The leftmost dimension must be an exact multiple of `THREADS` as these examples show:

```
/* Legal */
shared [10] int a[THREADS][10];

/* Legal */
shared [10] int a[10*THREADS][10];

/* Static THREADS compilation environment only.
   Legal if THREADS is an integral divisor of 100
   (leftmost dimension must be a multiple of THREADS) */
shared [10] int a[100][10];

/* Illegal: The block size must be an exact multiple
   of the product of all but leftmost extent
   (in this case, 100) */
shared [10] int a[THREADS][100];
```

As a consequence of the previously mentioned restrictions, some declarations are only supported in the static `THREADS` compilation environment, and then only for certain values of `THREADS`.

5.2.2 Maximum Blocking Size for Pointers to Shared Types

Currently, the declaration of pointer to shared types can specify a maximum blocking size of 1. For example, the following declarations are not supported:

```
shared [100] int *p;
shared [100] int *test();
```

However, `ap` in the following example is supported:

```
void example( shared [100] int ap[THREADS][100] );
```

even though the standard says that it is equivalent to this:

```
shared [100] int (*ap)[100]
```

5.2.3 (Deferred implementation) UPC Memory Functions

These UPC memory functions are deferred:

- `upc_global_alloc`
- `upc_free`
- `upc_global_exit`

5.2.4 (Deferred implementation) `upc_forall` Statement

The implementation of the `upc_forall` statement is currently deferred. In the meantime, you can rewrite `upc_forall` statements as the following examples show. Consider this `upc_forall` statement:

```
upc_forall( expr1; expr2; expr3; affinity ) {
   code;
}
```

If `affinity` is an address of a shared object, the statement can be rewritten as follows:

```
for( expr1; expr2; expr3 ) {
   if ( upc_threadof( affinity ) == MY_THREAD ) {
      code;
   }
}
```

If `affinity` is an integer expression, the statement can be rewritten as follows:

```
for( expr1; expr2; expr3 ) {
   if ( pmod( affinity, THREADS ) == MY_THREAD ) {
      code;
   }
}
```

where pmod(*a*,*b*) is evaluated as (*a*>=0)?(*a*%*b*):(((*a*%*b*)+*b*)%*b*).

The previous code construct will not perform at optimal efficiency. When the upc_forall statement is implemented, you should change your code back to the original form.

5.2.5 Cray UPC Functions

Cray UPC functions are provided in addition to the functions defined by the UPC draft. These Cray UPC functions supplement the UPC memory and lock functions:

- upc_all_free deallocates memory allocated by the upc_all_alloc function
- upc_local_free deallocates memory allocated by the upc_local_alloc function
- upc_all_lock_free frees a lock allocated by the upc_all_lock_alloc function
- upc_global_lock_free frees a lock allocated by the upc_all_lock_alloc function

For more information about the Cray UPC functions, refer to the man pages.

5.3 Compiling and Executing UPC Code

In order to compile UPC code, you must load the programming environment module (PrgEnv) and specify the -h upc option to the cc, c89, or c99 command. To execute your compiled code you can use the aprun or mpirun command depending on whether SHMEM, Cray Fortran co-arrays, or MPI is used with your UPC code.

The -X *npes* option can optionally be used to define the number of threads to use and statically set the value of the THREADS constant.

This example enables UPC and allows the THREADS symbol to be defined dynamically for the examp1 application:

```
cc -h upc -o multupc examp1.c
```

This example enables UPC and statically defines the THREADS symbol as 15 for the examp1 application:

```
cc -h upc -X 15 -o multupc examp1.c
```

The processing elements in *npes* (number of processing elements) are either MSPs or SSPs. To run programs on SSPs, you must specify the `-h ssp` compiler option. The default is to run on MSPs. See Section 2.10.10, page 21 for more information about using UPC in SSP mode.

After compiling the UPC code, you can run the program using the aprun command when the code contains UPC code only, or a mixture of UPC and SHMEM, and/or Cray Fortran Co-array code. If the code has a mixture of UPC and MPI code, use the mpirun command.

If you use the `-X` *npes* compiler option, you must specify the same number of threads in the aprun command.

Note: For more information about improving UPC code performance, refer to *Optimizing Applications on the Cray X1 System*.

Cray C++ Libraries [6]

The Cray C++ compiler together with the Dinkum C++ Libraries support the C++ 98 standard (ISO/IEC FDIS 14882) and continues to support existing Cray extensions. Most of the standard C++ features are supported, except for the few mentioned in Section 6.1. The Dinkum C++ Library is described in Section 6.2.

For information about C++ language conformance and exceptions, refer to Appendix D, page 195.

6.1 Unsupported Standard C++ Library Features

The Cray C++ compiler supports the C++ standard except for wide characters and multiple locales as follows:

- String classes using basic string class templates with wide character types or that use the `wstring` standard template class
- I/O streams using wide character objects
- File-based streams using file streams with wide character types (`wfilebuf`, `wifstream`, `wofstream`, and `wfstream`)
- Multiple localization libraries; Cray C++ supports only one locale

Note: The C++ standard provides a standard naming convention for library routines. Therefore, classes or routines that use wide characters are named appropriately. For example, the `fscanf` and `sprintf` functions do not use wide characters, but the `fwscanf` and `swprintf` function do.

6.2 Dinkum C++ Libraries

The Cray C++ compiler uses the Dinkum C++ libraries, which support standard C++. The Dinkum C++ Library documentation describes the library and are provided only in HTML and is accessible through CrayDoc. You can also find other references to tutorials and advanced user materials for the standard C++ library in the preface of this document.

Cray C++ Template Instantiation [7]

A *template* describes a class or function that is a model for a family of related classes or functions. The act of generating a class or function from a template is called *template instantiation*.

For example, a template can be created for a stack class, and then a stack of integers, a stack of floats, and a stack of some user-defined type can be used. In source code, these might be written as `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed during a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (template entities) are not necessarily done immediately for the following reasons:

- The preferred end result is one copy of each instantiated entity across all object files in a program. This applies to entities with external linkage.
- A specialization of a template entity is allowed. For example, a specific version of `Stack<int>`, or of just `Stack<int>::push` could be written to replace the template-generated version and to provide a more efficient representation for a particular data type.

  Because the compiler does not know about specializations of entities provided in future compilations when compiling a reference to a template entity, it cannot automatically instantiate the template in source files that contain references to the template.
- If a template function is not referenced, it should not be compiled because such functions could contain semantic errors that would prevent compilation. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

  Note: Certain template entities, such as inline functions, are always instantiated when they are used.

If the compiler is responsible for doing all instantiations automatically, it can only do so for the entire program. That is, the compiler cannot make decisions about instantiation of template entities until all source files of the complete program have been read.

The Cray C++ compiler provides an instantiation mechanism that performs automatic instantiation at linkage and provides command line options and `#pragma` directives that give the programmer more explicit control over instantiation.

7.1 Automatic Instantiation

The goal of an automatic instantiation mode is to provide trouble-free instantiation. The programmer should be able to compile source files to object code, link them and run the resulting program, without questioning how the necessary instantiations are done.

In practice, this is difficult for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses.

The Cray C++ compiler requires a normal, top-level, explicitly compiled source file that contains the definition of both the template entity and of any types required for the particular instantiation. This requirement is met in one of the following ways:

- Each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- When the compiler sees a template declaration in a .h file and discovers a need to instantiate that entity, implicit inclusion gives the compiler permission to search for an associated definition file having the same base name and a different suffix and implicitly include that file at the end of the compilation (see Section 7.5, page 146).
- The programmer makes sure that the files that define template entities also have the definitions of all the available types and adds code or directives in those files to request instantiation of those entities.

Automatic instantiation is accomplished by the Cray C++ compiler as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation. For any source file that makes use of a template instantiation, an associated `.ti` file is created, if one does not already exist (for example, the compilation of `abc.C` results in the creation of `abc.ti`).
2. When the object files are linked together, a program called the *prelinker* is run. It examines the object files, looking for references and definitions of

template entities and for any additional information about entities that could be instantiated.

Caution: The prelinker does not examine the object files in a library (`.a`) file.

3. If the prelinker finds a reference to a template entity for which there is no definition in the set of object files, it looks for a file that indicates that it could instantiate that template entity. Upon discovery of such a file, it assigns the instantiation to that file. The set of instantiations assigned to a given file (for example, `abc.C`) is recorded in an associated file that has a `.ii` suffix (for example, `abc.ii`).

4. The prelinker then executes the compiler to again recompile each file for which the `.ii` was changed.

5. During compilation, the compiler obeys the instantiation requests contained in the associated `.ii` file and produces a new object file that contains the requested template entities and the other things that were already in the object file.

6. The prelinker repeats steps 3 through 5 until there are no more instantiations to be adjusted.

7. The object files are linked together.

Once the program has been linked correctly, the `.ii` files contain a complete set of instantiation assignments. If source files are recompiled, the compiler consults the `.ii` files and does the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. This is true even if the entire program is recompiled. Because the `.ii` file contains information on how to recompile when instantiating, it is important that the `.o` and `.ii` files are not moved between the first compilation and linkage.

The prelinker cannot instantiate into and from library files (`.a`), so if a library is to be shared by many applications its templates should be expanded. You may find that creating a directory of objects with corresponding `.ii` files and the use of `-h prelink_copy_if_nonlocal` (see Section 2.7.8, page 16) will work as if you created a library (`.a`) that is shared.

The `-h prelink_local_copy` option indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations. This option is useful when you are sharing some common

relocatables but do not want them updated. Another way to ensure that shared `.o` files are not updated is to use the `-h remove_instantiation_flags` option when compiling the shared `.o` files. This also makes smaller resulting shared `.o` files.

An easy way to create a library that instantiates all references of templates within the library is to create an empty `main` function and link it with the library, as shown in the following example. The prelinker will instantiate those template references that are within the library to one of the relocatables without generating duplicates. The empty `dummy_main.o` file is removed prior to creating the `.a` file.

```
CC a.C b.C c.C dummy_main.C
ar cr mylib.a a.o b.o c.o
```

Another alternative to creating a library that instantiates all references of templates, is to use the `-h one_instantiation_per_object` option. This option directs the prelinker to instantiate each template referenced within a library in its own object file. The following example shows how to use the option:

```
CC -h one_instantiation_per_object a.C b.C c.C dummy_main.C
ar cr mylib.a a.o b.o c.o myInstantiationsDir/*.int.o
```

For more information about this alternative see Section 7.3, page 144 and Section 2.7.2, page 15.

If a specialization of a template entity is provided somewhere in the program, the specialization is seen as a definition by the prelinker. Because that definition satisfies the references to that entity, the prelinker will not request an instantiation of the entity. If a specialization of a template is added to a previously compiled program, the prelinker removes the assignment of the instantiation from the proper `.ii` file.

The `.ii` files do not, in general, require any manual intervention. The exception occurs when a definition is changed in such a way that some instantiation no longer compiles (it receives errors) and at the same time a specialization is added to another file and the first file is recompiled before the specialization file. If this exception occurs, the `.ii` file that corresponds to the file that generated the errors must be deleted manually to allow the prelinker to regenerate it.

Automatic instantiation can coexist with partial explicit control of instantiation by the programmer through the use of `#pragma` directives or the `-h instantiate=`*mode* option.

Automatic instantiation mode can be disabled by issuing the `-h noautoinstantiate` command line option. If automatic instantiation is

disabled, the information about template entities that could be instantiated in a file is not included in the object file.

7.2 Instantiation Modes

Normally, during compilation of a source file, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by issuing the `-h instantiate=`*mode* command line option. The *mode* argument can be specified as follows:

mode	Description
`none`	Do not automatically create instantiations of any template entities. This is the most appropriate mode when automatic instantiation is enabled. This is the default instantiation mode.
`used`	Instantiate those template entities that were used in the compilation. This includes all static data members that have template definitions.
`all`	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members are instantiated, regardless of whether they were used. Nonmember template functions are instantiated even if the only reference was a declaration.
`local`	Similar to `used` mode, except that the functions are given internal linkage. This mode provides a simple mechanism for those who are not familiar with templates. The compiler instantiates the functions used in each compilation unit as local functions, and the program links and runs correctly (barring problems due to multiple copies of local static variables). This mode may generate multiple copies of the instantiated functions and is not suitable for production use. This mode cannot be used in conjunction with automatic template instantiation. Automatic instantiation is disabled by this mode.

In the case where the `CC`(1) command is given a single source file to compile and link, all instantiations are done in the single source file and, by default, the `used` mode is used and automatic instantiation is suppressed.

7.3 One Instantiation Per Object File

You can direct the prelinker to instantiate each template referenced in the source into its own object file. This method is preferred over other template instantiation object file generation options because:

- The user of a library pulls in only the instantiations that are needed.
- Multiple libraries with the same template can link. If each instantiation is not placed in its own object file, linking a library with another library that also contains the same instantiations will generate warnings on some platforms.

Use the `-h one_instantiation_per_object` option to generate one object file per instantiation. For more information about this option, see Section 2.7.2, page 15.

7.4 Instantiation `#pragma` Directives

Instantiation `#pragma` directives can be used in source code to control the instantiation of specific template entities or sets of template entities. There are three instantiation `#pragma` directives:

- The `#pragma _CRI instantiate` directive causes a specified entity to be instantiated.
- The `#pragma _CRI do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.
- The `#pragma _CRI can_instantiate` directive indicates that a specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

The argument to the `#pragma _CRI instantiate` directive can be any of the following:

- A template class name. For example: `A<int>`
- A template class declaration. For example: `class A<int>`
- A member function name. For example: `A<int>::f`
- A static data member name. For example: `A<int>::i`
- A static data declaration. For example: `int A<int>::i`

- A member function declaration. For example: `void A<int>::f(int, char)`
- A template function declaration. For example: `char* f(int, float)`

A `#pragma` directive in which the argument is a template class name (for example, `A<int>` or `class A<int>`) is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class, a given member function or static data member may be excluded using the `#pragma _CRI do_not_instantiate` directive. For example:

```
#pragma _CRI instantiate A<int>
#pragma _CRI do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the `#pragma instantiate` directive and no template definition is available or a specific definition is provided, an error is issued.

The following example illustrates the use of the `#pragma _CRI instantiate` directive:

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
void main()
{
 int    i;
 double  d;
 f1(i);
 f1(d);
 g1(i);
 g1(d);
}
#pragma _CRI instantiate void f1(int) // error-specific definition
#pragma _CRI instantiate void g1(int) // error-no body provided
```

In the preceding example, `f1(double)` and `g1(double)` are not instantiated because no bodies are supplied, but no errors will be produced during the compilation. If no bodies are supplied at link time, a linker error is issued.

A member function name (such as `A<int>::f`) can be used as a `#pragma` directive argument only if it refers to a single, user-defined member function (that is, not an overloaded function). Compiler-generated functions are

not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in the following example:

```
#pragma _CRI instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

7.5 Implicit Inclusion

The implicit inclusion feature implies that if the compiler needs a definition to instantiate a template entity declared in a `.h` file, it can implicitly include the corresponding `.C` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation, but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will look to see if a file `xyz.C` exists and, if so, it processes it as if it were included at the end of the main source file.

To find the template definition file for a given template entity, the Cray C++ compiler must know the full path name to the file in which the template was declared and whether the file was included using the system include syntax (such as `#include <file.h>`). This information is not available for preprocessed source code containing `#line` directives. Consequently, the Cray C++ compiler does not attempt implicit inclusion for source code that contains `#line` directives.

The set of definition-file suffixes that are tried by default, is `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, and `.cc`.

Implicit inclusion works well with automatic instantiation, however, they are independent. They can be enabled or disabled independently, and implicit inclusion is still useful without automatic instantiation.

Cray C Extensions [8]

The Cray C compiler supports these extensions, developed by Cray, to the C standard:

- Complex data extensions (Section 8.1, page 147)
- `fortran` keyword (Section 8.2, page 148)
- Hexadecimal floating-point constants (Section 8.3, page 148)

A program that uses one or more extensions does not strictly conform to the standard. These extensions are not available in strict conformance mode.

8.1 Complex Data Extensions

Cray C extends the complex data facilities defined by standard C with these extensions:

- Imaginary constants
- Incrementing or decrementing `_Complex` data

The Cray C compiler supports the Cray imaginary constant extension and is defined in the `<complex.h>` header file. This imaginary constant has the following form:

R`i`

R is either a floating constant or an integer constant; no space or other character can appear between *R* and `i`. If you are compiling in strict conformance mode (`-h conform`), the Cray imaginary constants are not available.

The following example illustrates imaginary constants:

```
#include <complex.h>
double complex z1 = 1.2 + 3.4i;
double complex z2 = 5i;
```

The other extension to the complex data facility allows the prefix– and postfix-increment and decrement operators to be applied to the `_Complex` data type. The operations affect only the real portion of a complex number.

8.2 `fortran` Keyword

In extended mode, the identifier `fortran` is treated as a keyword. It specifies a storage class that can be used to declare a Fortran-coded external function. The use of the `fortran` keyword when declaring a function causes the compiler to verify that the arguments used in each call to the function are pass by addresses; any arguments that are not addresses are converted to addresses.

As in any function declaration, an optional *type-specifier* declares the type returned, if any. Type `int` is the default; type `void` can be used if no value is returned (by a Fortran subroutine). The `fortran` storage class causes conversion of lowercase function names to uppercase, and, if the function name ends with an underscore character, the trailing underscore character is stripped from the function name. (Stripping the trailing underscore character is in keeping with UNIX practice.)

Functions specified with a `fortran` storage class must not be declared elsewhere in the file with a `static` storage class.

Note: The `fortran` keyword is not allowed in Cray C++.

An example using the `fortran` keyword is shown in Section 11.3.7, page 166.

8.3 Hexadecimal Floating-point Constants

The Cray C compiler supports the standard hexadecimal floating constant notations and the Cray hexadecimal floating constant notation. The standard hexadecimal floating constants are portable and have sizes that are dependent upon the hardware. The remainder of this section discusses the Cray hexadecimal floating constant.

The Cray hexadecimal floating constant feature is not portable, because identical hexadecimal floating constants can have different meanings on different systems. It can be used whenever traditional floating-point constants are allowed.

The hexadecimal constant has the usual syntax: `0x` (or `0X`) followed by hexadecimal characters. The optional floating suffix has the same form as for normal floating constants: `f` or `F` (for float), `l` or `L` (for long), optionally followed by an `i` (imaginary).

The constant must represent the same number of bits as its type, which is determined by the suffix (or the default of double). The constant's bit length is four times the number of hexadecimal digits, including leading zeros.

The following example illustrates hexadecimal constant representation:

`0x7f7fffff.f`	32-bit float
`0x0123456789012345.`	64-bit double

The value of a hexadecimal floating constant is interpreted as a value in the specified floating type. This uses an unsigned integral type of the same size as the floating type, regardless of whether an object can be explicitly declared with such a type. No conversion or range checking is performed. The resulting floating value is defined in the same way as the result of accessing a member of floating type in a union after a value has been stored in a different member of integral type.

The following example illustrates hexadecimal floating-point constant representation that use Cray floating-point format:

```
int main(void)
{
   float f1, f2;
   double g1, g2;

   f1 = 0x3ec00000.f;
   f2 = 0x3fc00000.f;
   g1 = 0x40fa400100000000.;
   g2 = 0x40fa400200000000.;

   printf("f1 = %8.8g\n", f1);
   printf("f2 = %8.8g\n", f2);
   printf("g1 = %16.16g\n", g1);
   printf("g2 = %16.16g\n", g2);
   return 1;
}
```

This is the output for the previous example:

```
f1 =   0.375
f2 =    1.5
g1 =    107520.0625
g2 =     107520.125
```

Predefined Macros [9]

Predefined macros can be divided into the following categories:

- Macros required by the C and C++ standards
- Macros based on the host machine
- Macros based on the target machine
- Macros based on the compiler

Predefined macros provide information about the compilation environment. In this chapter, only those macros that begin with the underscore (_) character are defined when running in strict-conformance mode (see the `-h conform` command line option in Section 2.6.2, page 13).

> **Note:** Any of the predefined macros except those required by the standard (see Section 9.1, page 151) can be undefined by using the `-U` command line option; they can also be redefined by using the `-D` command line option.

A large set of macros is also defined in the standard header files.

9.1 Macros Required by the C and C++ Standards

The following macros are required by the C and C++ standards:

Macro	Description
`__TIME__`	Time of translation of the source file.
`__DATE__`	Date of translation of the source file.
`__LINE__`	Line number of the current line in your source file.
`__FILE__`	Name of the source file being compiled.
`__STDC__`	Defined as the decimal constant 1 if compilation is in strict conformance mode; defined as the decimal constant 2 if the compilation is in extended mode. This macro is defined for Cray C and C++ compilations.
`__cplusplus`	Defined as 1 when compiling Cray C++ code and undefined when compiling Cray C code. The `__cplusplus` macro is required by the ISO C++ standard, but not the ISO C standard.

9.2 Macros Based on the Host Machine

The following macros provide information about the environment running on the host machine:

Macro	Description
`__unix`	Defined as 1 if the machine uses the UNIX OS.
`unix`	Defined as 1 if the machine uses the UNIX OS. This macro is not defined in strict-conformance mode.
`_UNICOSMP`	Defined as 1 if the operating system is UNICOS/mp. This macro is not defined in strict-conformance mode.

9.3 Macros Based on the Target Machine

The following macros provide information about the characteristics of the target machine:

Macro	Description
`_ADDR64`	Defined as 1 if the targeted CPU has 64-bit address registers; if the targeted CPU does not have 64-bit address registers, the macro is not defined.
`__sv`	Defined as 1 on all Cray X1 systems.
`__sv2`	Defined as 1 and indicates that the current system is a Cray X1 system.
`_CRAY`	Defined as 1 on UNICOS/mp systems.
`_CRAYIEEE`	Defined as 1 if the targeted CPU type uses IEEE floating-point format.
`_CRAYSV2`	Defined as 1 and indicates that the current system is a Cray X1 system.
`__crayx1`	Defined as 1 and indicates that the current system is a Cray X1 system.
`_MAXVL`	Defined as the maximum hardware vector length, which is 64.
`cray`	Defined as 1 on UNICOS/mp. This macro is not defined in strict-conformance mode.

CRAY	Defined as 1 on UNICOS/mp systems. This macro is not defined in strict-conformance mode.

9.4 Macros Based on the Compiler

The following macros provide information about compiler features:

Macro	Description
_RELEASE	Defined as the major release level of the compiler.
_RELEASE_MINOR	Defined as the minor release level of the compiler.
_RELEASE_STRING	Defined as a string that describes the version of the compiler.
_CRAYC	Defined as 1 to identify the Cray C and C++ compilers.

Debugging Cray C and C++ Code [10]

The Etnus TotalView symbolic debugger is available to help you debug C and C++ codes (refer to *Etnus TotalView Users Guide*). In addition, the Cray C and C++ compilers provide the following features to help you in debugging codes:

- The `-G` and `-g` compiler options provide symbol information about your source code for use by the Etnus TotalView debugger. For more information on these compiler options, see Section 2.16.1, page 31.
- The `-h [no]bounds` option and the `#pragma _CRI [no]bounds` directive let you check pointer and array references. The `-h [no]bounds` option is described in Section 2.16.2, page 32. The `#pragma _CRI [no]bounds` directive is described in Section 3.5.1, page 57.
- The `#pragma _CRI message` directive lets you add warning messages to sections of code where you suspect problems. The `#pragma _CRI message` directive is described in Section 3.5.3, page 61.
- The `#pragma _CRI [no]opt` directive lets you selectively isolate portions of your code to optimize, or to toggle optimization on and off in selected portions of your code. The `#pragma _CRI [no]opt` directive is described in Section 3.5.5, page 62.

10.1 Etnus TotalView Debugger

Some of the functions available in the TotalView debugger allow you to perform the following actions:

- Set and clear breakpoints, which can be conditional, at both the source code level and the assembly code level
- Examine core files
- Step through a program, including across function calls
- Reattach to the executable file after editing and recompiling
- Edit values of variables and memory locations
- Evaluate code fragments

10.2 Compiler Debugging Options

To use the TotalView debugger in debugging your code, you must first compile your code using one of the debugging options (`-g` or `-G`). These options are specified as follows:

- `-Gf`

  If you specify the `-Gf` debugging option, the TotalView debugger allows you to set breakpoints at function entry and exit and at labels.

- `-Gp`

  If you specify the `-Gp` debugging option, the TotalView debugger allows you to set breakpoints at function entry and exit, labels, and at places where execution control flow changes (for example, loops, `switch`, and `if...else` statements).

- `-Gn` or `-g`

  If you specify the `-Gn` or `-g` debugging option, the TotalView debugger allows you to set breakpoints at function entry and exit, labels, and executable statements. These options force all compiler optimizations to be disabled as if you had specified `-O0`.

Users of the Cray C and C++ compilers do not have to sacrifice run time performance to debug codes. Many compiler optimizations are inhibited by breakpoints generated for debugging. By specifying a higher debugging level, fewer breakpoints are generated and better optimization occurs.

However, consider the following cases in which optimization is affected by the `-Gp` and `-Gf` debugging options:

- Vectorization can be inhibited if a label exists within the vectorizable loop.
- Vectorization can be inhibited if the loop contains a nested block and the `-Gp` option is specified.
- When the `-Gp` option is specified, setting a breakpoint at the first statement in a vectorized loop allows you to stop and display at each vector iteration. However, setting a breakpoint at the first statement in an unrolled loop may not allow you to stop at each vector iteration.

Interlanguage Communication [11]

In some situations, it is necessary or advantageous to make calls to assembly or Fortran functions from C or C++ programs. This chapter describes how to make such calls. It also discusses calls to C and C++ functions from Fortran and assembly language. For additional information on interlanguage communication, see *Interlanguage Programming Conventions*. The calling sequence is described in detail on the `callseq`(3) man page.

The C and C++ compilers provide a mechanism for declaring external functions that are written in other languages. This allows you to write portions of an application in C, C++, Fortran, or assembly language. This can be useful in cases where the other languages provide performance advantages or utilities that are not available in C or C++.

This section describes how to call assembly language and Fortran programs from a C or C++ program. It also discusses the issues related to calling C or C++ programs from other languages.

11.1 Calls between C and C++ Functions

The following requirements must be considered when making calls between functions written in C and C++:

- In Cray C++, the `extern "C"` linkage is required when declaring an external function that is written in Cray C or when declaring a Cray C++ function that is to be called from Cray C. Normally the compiler will mangle function names to encode information about the function's prototype in the external name. This prevents direct access to these function names from a C function. The `extern "C"` keyword will prevent the compiler from performing name mangling.
- The program must be linked using the `CC`(1) command.

Objects can be shared between C and C++. There are some Cray C++ objects that are not accessible to Cray C functions (such as classes). The following object types can be shared directly:

- Integral and floating types.
- Structures and unions that are declared identically in C and C++. In order for structures and unions to be shared, they must be declared with identical members in the identical order.

- Arrays and pointers to the above types.

In the following example, a Cray C function (C_add_func) is called by the Cray C++ main program:

```
#include <iostream.h>

extern "C" int C_add_func(int, int);
int global_int = 123;

main()
{
  int res, i;

  cout << "Start C++ main" << endl;

   // Call C function to add two integers and return result.

  cout << "Call C C_add_func" << endl;
  res = C_add_func(10, 20);
  cout << "Result of C_add_func = " << res << endl;
  cout << "End C++ main << endl;
}
```

The Cray C function (C_add_func) is as follows:

```
#include <stdio.h>

extern int global_int;

int C_add_func(int p1, int p2)
{
    printf("\tStart C function C_add_func.\n");
    printf("\t\tp1     = %d\n", p1);
    printf("\t\tp2     = %d\n", p2);
    printf("\t\tglobal_int = %d\n", global_int);
    return p1 + p2;
}
```

The output from the execution of the calling sequence illustrated in the preceding example is as follows:

```
Start C++ main
Call C C_add_func
        Start C function C_add_func.
                p1     = 10
                p2     = 20
                global_int = 123
Result of C_add_func = 30
End C++ main
```

11.2 Calling Assembly Language Functions from a C or C++ Function

You can sometimes avoid bottlenecks in programs by rewriting parts of the program in assembly language, maximizing performance by selecting instructions to reduce machine cycles. When writing assembly language functions that will be called by C or C++ functions, use the standard UNICOS/mp program linkage macros. When using these macros, you do not need to know the specific registers used by the C or C++ program or by the calling sequence of the assembly coded routine.

In Cray C++, use `extern "C"` to declare the assembly language function.

11.2.1 (Deferred implementation) Cray Assembly Language (CAL) Functions

The use of Cray Assembly Language (CAL) is described in the *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual*.

The `ALLOC`, `DEFA`, `DEFS`, `ENTER`, `EXIT`, and `MXCALLEN` macros can be used to define the calling list; A and S register use; temporary storage; and entry and exit points.

11.3 Calling Fortran Functions and Subroutines from a C or C++ Function

This subsection describes the following aspects of calling Fortran from C or C++. Topics include requirements and guidelines, argument passing, array storage, logical and character data, accessing named common, and accessing blank common.

11.3.1 Requirements

Keep the following points in mind when calling Fortran functions from C or C++:

- Fortran uses the call-by-address convention. C and C++ use the call-by-value convention, which means that only pointers should be passed to Fortran subprograms. See Section 11.3.2, page 160.
- Fortran arrays are in column-major order. C and C++ arrays are in row-major order. This indicates which dimension is indicated by the first value in an array element subscript. See Section 11.3.3, page 161.
- Single-dimension arrays of signed 32-bit integers and single dimension arrays of 32-bit floating-point numbers are the only aggregates that can be passed as parameters without changing the arrays.
- Fortran character pointers and character pointers from Cray C and C++ are incompatible. See Section 11.3.4, page 162.
- Fortran logical values and the Boolean values from C and C++ are not fully compatible. See Section 11.3.4, page 162.
- External C and C++ variables are stored in common blocks of the same name, making them readily accessible from Fortran programs if the C or C++ variable is in uppercase.
- When declaring Fortran functions or objects in C or C++, the name must be specified in all uppercase letters, digits, or underscore characters and consist of 31 or fewer characters.
- In Cray C, Fortran functions can be declared using the `fortran` keyword (see Section 8.2, page 148). The `fortran` keyword is not available in Cray C++. Instead, Fortran functions must be declared by specifying `extern "C"`.

11.3.2 Argument Passing

Because Fortran subroutines expect arguments to be passed by pointers rather than by value, C and C++ functions called from Fortran subroutines must pass pointers rather than values.

All argument passing in Cray C is strictly by value. To prepare for a function call between two Cray C functions, a copy is made of each actual argument. A function can change the values of its formal parameters, but these changes cannot affect the values of the actual arguments. It is possible, however, to pass a pointer. (All array arguments are passed by this method.) This capability is analogous to the Fortran method of passing arguments.

In addition to passing by value, Cray C++ also provides passing by reference.

11.3.3 Array Storage

C and C++ arrays are stored in memory in row-major order. Fortran arrays are stored in memory in column-major order. For example, the C or C++ array declaration `int A[3][2]` is stored in memory as:

`A[0][0]`	`A[0][1]`
`A[1][0]`	`A[1][1]`
`A[2][0]`	`A[2][1]`

The previously defined array is viewed linearly in memory as:

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

The Fortran array declaration `INTEGER A(3,2)` is stored in memory as:

`A(1,1)`	`A(2,1)`	`A(3,1)`
`A(1,2)`	`A(2,2)`	`A(3,2)`

The previously defined array is viewed linearly in memory as:

```
A(1,1)  A(2,1)  A(3,1)  A(1,2)  A(2,2)  A(3,2)
```

When an array is shared between Cray C, C++, and Fortran, its dimensions are declared and referenced in C and C++ in the opposite order in which they are declared and referenced in Fortran. Arrays are zero-based in C and C++ and are one-based in Fortran, so in C and C++ you should subtract 1 from the array subscripts that you would normally use in Fortran.

For example, using the Fortran declaration of array `A` in the preceding example, the equivalent declaration in C or C++ is:

```
int a[2][3];
```

The following list shows how to access elements of the array from Fortran and from C or C++:

Fortran	C or C++
A(1,1)	A[0][0]
A(2,1)	A[0][1]
A(3,1)	A[0][2]
A(1,2)	A[1][0]
A(2,2)	A[1][1]
A(3,2)	A[1][2]

11.3.4 Logical and Character Data

Logical and character data need special treatment for calls between C or C++ and Fortran. Fortran has a character descriptor that is incompatible with a character pointer in C and C++. The techniques used to represent logical (Boolean) values also differ between Cray C, C++, and Fortran.

Mechanisms you can use to convert one type to the other are provided by the standard header file and conversion utilities shown in the following list:

Header file or utility	Description
_btol	Conversion utility that converts a 0 to a Fortran logical .FALSE. and a nonzero value to a Fortran logical .TRUE.
_ltob	Conversion utility that converts a Fortran logical .FALSE. to a 0 and a Fortran logical .TRUE. to a 1.

11.3.5 Accessing Named Common from C and C++

The following example demonstrates how external C and C++ variables are accessible in Fortran named common blocks. It shows a C or C++ C function calling a Fortran subprogram, the associated Fortran subprogram, and the associated input and output.

In this example, the C or C++ structure ST is accessed in the Fortran subprogram as common block ST. The name of the structure and the Fortran common block must match. Note that this requires that the structure name be uppercase. The C and C++ C structure member names and the Fortran common block member names do not have to match, as is shown in this example.

The following Cray C main program calls the Fortran subprogram FCTN:

```
#include <stdio.h>
struct
{
   int i;
   double a[10];
   long double d;
} ST;

main()
{
   int i;

   /* initialize struct ST */
   ST.i = 12345;

   for (i = 0; i < 10; i++)
      ST.a[i] = i;

   ST.d = 1234567890.1234567890L;

   /* print out the members of struct ST */
   printf("In C: ST.i = %d, ST.d = %20.10Lf\n", ST.i, ST.d);
   printf("In C: ST.a = ");
   for (i = 0; i < 10; i++)
      printf("%4.1f", ST.a[i]);
   printf("\n\n");

   /* call the fortran function */
   FCTN();
}
```

The following example is the Fortran subprogram FCTN called by the previous Cray C main program:

```
C *********** Fortran subprogram (f.f): ***********

      SUBROUTINE FCTN

      COMMON /ST/STI, STA(10), STD
      INTEGER STI
      REAL STA
      DOUBLE PRECISION STD
```

```
      INTEGER I

      WRITE(6,100) STI, STD
  100 FORMAT ('IN FORTRAN: STI = ', I5, ', STD = ', D25.20)
      WRITE(6,200) (STA(I), I = 1,10)
  200 FORMAT ('IN FORTRAN: STA =', 10F4.1)
      END
```

The previous Cray C and Fortran examples are executed by the following commands, and they produce the output shown:

```
%cc -c c.c
%ftn -c f.f
%ld c.o f.o
%a.out
 ST.i = 12345, ST.d = 1234567890.1234567890
 In C: ST.a =  0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

 IN FORTRAN: STI = 12345, STD = .12345678901234567889D+10
 IN FORTRAN: STA = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

11.3.6 Accessing Blank Common from C or C++

Fortran includes the concept of a common block. A *common block* is an area of memory that can be referenced by any program unit in a program. A *named common block* has a name specified in names of variables or arrays stored in the block. A *blank common block*, sometimes referred to as blank common, is declared in the same way, but without a name.

There is no way to access blank common from C or C++ similar to accessing a named common block. However, you can write a simple Fortran function to return the address of the first word in blank common to the C or C++ program and then use that as a pointer value to access blank common.

The following example shows how Fortran blank common can be accessed using C or C++ source code:

```
#include <stdio.h>

struct st
{
  float a;
  float b[10];
} *ST;

#ifdef __cplusplus
  extern "C" struct st *MYCOMMON(void);
  extern "C" void FCTN(void);
#else
  fortran struct st *MYCOMMON(void);
  fortran void FCTN(void);
#endif

main()
{
    int i;

    ST = MYCOMMON();
    ST->a = 1.0;
    for (i = 0; i < 10; i++)
        ST->b[i] = i+2;
    printf("\n In C and C++\n");
    printf("     a = %5.1f\n", ST->a);
    printf("     b = ");
    for (i = 0; i < 10; i++)
        printf("%5.1f ", ST->b[i]);
    printf("\n\n");

    FCTN();
}
```

This Fortran source code accesses blank common and is accessed from the C or C++ source code in the preceding example:

```
SUBROUTINE FCTN
COMMON // STA,STB(10)
PRINT *, "IN FORTRAN"
PRINT *, "    STA = ",STA
PRINT *, "    STB = ",STB
STOP
END

FUNCTION MYCOMMON()
COMMON // A
MYCOMMON = LOC(A)
RETURN
END
```

This is the output of the previous C or C++ source code:

```
a =  1.0
b = 2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0
```

This is the output of the previous Fortran source code:

```
STA = 1.
STB = 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
```

11.3.7 Cray C and Fortran Example

Here is an example of a Cray C function that calls a Fortran subprogram. The Fortran subprogram example follows the Cray C function example, and the input and output from this sequence follows the Fortran subprogram example.

Note: This example assumes that the Cray Fortran function is compiled with the `-s default32` option enabled. The examples will not work if the `-s default64` option is enabled.

```
/*                C program (main.c):                   */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

/* Declare prototype of the Fortran function. Note the last */
/* argument passes the length of the first argument. */
fortran double FTNFCTN (char *, int *, int);

double FLOAT1 = 1.6;
double FLOAT2;  /* Initialized in FTNFCTN */

main()
{
     int clogical, ftnlogical, cstringlen;
     double rtnval;
     char *cstring = "C Character String";


/* Convert clogical to its Fortran equivalent */
     clogical = 1;
     ftnlogical = _btol(clogical);

/* Print values of variables before call to Fortran function */
     printf(" In main: FLOAT1 = %g; FLOAT2 = %g\n",
          FLOAT1, FLOAT2);
     printf(" Calling FTNFCTN with arguments:\n");
     printf(" string = \"%s\"; logical = %d\n\n", cstring, clogical);
     cstringlen = strlen(cstring);
     rtnval = FTNFCTN(cstring, &ftnlogical, cstringlen);

/* Convert ftnlogical to its C equivalent */
     clogical = _ltob(&ftnlogical);

/* Print values of variables after call to Fortran function */
     printf(" Back in main: FTNFCTN returned %g\n", rtnval);
     printf(" and changed the two arguments:\n");
     printf(" string = \"%.*s\"; logical = %d\n",
     cstringlen, cstring, clogical);
}
```

```
C                 Fortran subprogram (ftnfctn.f):


      FUNCTION FTNFCTN(STR, LOG)

      REAL FTNFCTN
      CHARACTER*(*) STR
      LOGICAL LOG

      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT2/2.4/          ! FLOAT1 INITIALIZED IN MAIN

C       PRINT CURRENT STATE OF VARIABLES
        PRINT*, '     IN FTNFCTN: FLOAT1 = ', FLOAT1,
       1                          ';FLOAT2 = ', FLOAT2
        PRINT*, '     ARGUMENTS:    STR = "', STR, '"; LOG = ', LOG

C       CHANGE THE VALUES FOR STR(ING) AND LOG(ICAL)
        STR = 'New Fortran String'
        LOG = .FALSE.

        FTNFCTN = 123.4
        PRINT*, '    RETURNING FROM FTNFCTN WITH ', FTNFCTN
        PRINT*
        RETURN
        END
```

The previous Cray C function and Fortran subprogram are executed by the following commands and produce the following output:

```
$cc -c main.c
$ftn -c ftnfctn.f
$ld main.o ftnfctn.o
$a.out
In main: FLOAT1 = 1.6;  FLOAT2 = 2.4
Calling FTNFCTN with arguments:
string = "C Character String"; logical = 1

IN FTNFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
ARGUMENTS:   STR = "C Character String"; LOG = T
RETURNING FROM FTNFCTN WITH 123.4

Back in main: FTNFCTN returned 123.4
and changed the two arguments:
string = "New Fortran String"; logical = 0
```

11.3.8 Calling a Fortran Program from a Cray C++ Program

The following example illustrates how a Fortran program can be called from a Cray C++ program:

```
#include <iostream.h>
extern "C" int FORTRAN_ADD_INTS(int *arg1, int &arg2);

main()
{
    int num1, num2, res;
    cout << "Start C++ main" << endl << endl;

    //Call FORTRAN function to add two integers and return result.
    //Note that the second argument is a reference parameter so
    //it is not necessary to take the address of the
    //variable num2.

    num1 = 10;
    num2 = 20;
    cout << "Before Call to FORTRAN_ADD_INTS" << endl;
    res = FORTRAN_ADD_INTS(&num1, num2);
    cout << "Result of FORTRAN Add = " << res << endl << endl;
    cout << "End C++ main" << endl;
}
```

The Fortran program that is called from the Cray C++ `main` function in the preceding example is as follows:

```
INTEGER FUNCTION FORTRAN_ADD_INTS(Arg1, Arg2)
INTEGER Arg1, Arg2

PRINT *," FORTRAN_ADD_INTS, Arg1,Arg2 = ", Arg1, Arg2
FORTRAN_ADD_INTS = Arg1 + Arg2
END
```

The output from the execution of the preceding example is as follows:

```
Start C++ main

Before Call to FORTRAN_ADD_INTS
  FORTRAN_ADD_INTS, Arg1,Arg2 =  10,  20
Result of FORTRAN Add = 30

End C++ main
```

11.4 Calling a C or C++ Function from a Fortran or Assembly Language Program

A C or C++ function can be called from a Fortran or (Deferred implementation) assembly language program. One of two methods can be used to call C functions from Fortran: the C interoperability feature provided by the Fortran 2000 facility or the method documented in this section. C interoperability provides a standard portable interoperability mechanism for Fortran and C programs. Refer to *Fortran Language Reference Manual, Volume 2* for more information about C interoperability. If you are using the method documented in this section to call C functions from Fortran, keep in mind the information in Section 11.3, page 159.

When calling a Cray C++ function from a Fortran or (Deferred implementation) assembly language program, observe the following rules:

- The Cray C++ function must be declared with `extern "C"` linkage.
- The program must be linked with the `CC`(1) command.

The example that follows illustrates a Fortran program that calls a Cray C function. The Cray C function being called, the commands required, and the associated input and output are also included.

Note: This example assumes that the Cray Fortran program is compiled with the `-s default32` option enabled. The examples will not work if the `-s default64` option is enabled.

```
C   Fortran program (main.f):

      PROGRAM MAIN

      REAL CFCTN
      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT1/1.6/      ! FLOAT2 INITIALIZED IN cfctn
      LOGICAL LOG
      CHARACTER*24 STR
      REAL RTNVAL

C  INITIALIZE VARIABLES STR(ING) AND LOG(ICAL)
      STR = 'Fortran Character String'
      LOG = .TRUE.

C  PRINT VALUES OF VARIABLES BEFORE CALL TO C FUNCTION
      PRINT*, ' IN MAIN: FLOAT1 = ', FLOAT1,
     1                '; FLOAT2 = ', FLOAT2
      PRINT*, ' CALLING CFCTN WITH ARGUMENTS: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      PRINT*

      RTNVAL = CFCTN(STR, LOG)

C  PRINT VALUES OF VARIABLES AFTER CALL TO C FUNCTION
      PRINT*, ' BACK IN MAIN: CFCTN RETURNED ', RTNVAL
      PRINT*, ' AND CHANGED THE TWO ARGUMENTS: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      END
```

The following example illustrates the associated Cray C function that is being called:

```
/*              C function (cfctn.c):                          */
#include <fortran.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

double FLOAT1;        /* Initialized in MAIN */
double FLOAT2 = 2.4;

/* The slen argument passes the length of string in str */
double CFCTN(char * str, int *log, int slen)
{
     int clog;
     float returnval;
     char *cstring;

/* Convert log passed from Fortran MAIN */
/* into its C equivalent */
    cstring = malloc(slen+1);
    strncpy(cstring, str, slen);
    cstring[slen] = '\0';
    clog = _ltob(log);

/* Print the current state of the variables */
    printf("     In CFCTN: FLOAT1 = %.1f; FLOAT2 = %.1f\n",
               FLOAT1, FLOAT2);
    printf("     Arguments: str = "%s"; log = %d\n",
    cstring, clog);

/* Change the values  for str and log */
    strncpy(str, "C Character String     ", 24);
    *log = 0;

    returnval = 123.4;
    printf("     Returning from CFCTN with %.1f\n\n", returnval);
    return(returnval);
}
```

The previous Fortran program and Cray C function are executed by the following commands and produce the following output:

```
%cc -c cfctn.c
% ftn -c main.f
%ftn cfctn.o main.o
%a.out
IN MAIN: FLOAT1 = 1.6; FLOAT2 = 2.4
CALLING CFCTN WITH ARGUMENTS:
STR = "Fortran Character String"; LOG = T

     In CFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
     Arguments: str = "Fortran Character String"; log = 1
     Returning from CFCTN with 123.4

BACK IN MAIN: CFCTN RETURNED 123.4
AND CHANGED THE TWO ARGUMENTS:
STR = "C Character String  "; LOG = F
```

Implementation-defined Behavior [12]

This chapter describes compiler behavior that is defined by the implementation according to the C and/or C++ standards. The standards require that the behavior of each particular implementation be documented.

12.1 Implementation-defined Behavior

The C and C++ standards define implementation-defined behavior as behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation. The behavior of the Cray C and C++ compilers for these cases is summarized in this section.

12.1.1 Messages

All diagnostic messages issued by the compilers are reported through the UNICOS/mp message system. For information on messages issued by the compilers and for information about the UNICOS/mp message system, see Appendix E, page 211.

12.1.2 Environment

When `argc` and `argv` are used as parameters to the `main` function, the array members `argv[0]` through `argv[argc-1]` contain pointers to strings that are set by the command shell. The shell sets these arguments to the list of words on the command line used to invoke the compiler (the argument list). For further information on how the words in the argument list are formed, refer to the documentation on the shell in which you are running. For information on UNICOS/mp shells, see the `sh`(1) or `csh`(1) man page.

A third parameter, `char **envp`, provides access to environment variables. The value of the parameter is a pointer to the first element of an array of null-terminated strings, that matches the output of the `env`(1) command. The array of pointers is terminated by a null pointer.

The compiler does not distinguish between interactive devices and other, noninteractive devices. The library, however, may determine that `stdin`, `stdout`, and `stderr` (`cin`, `cout`, and `cerr` in Cray C++) refer to interactive devices and buffer them accordingly.

12.1.2.1 Identifiers

The *identifier* (as defined by the standards) is merely a sequence of letters and digits. Specific uses of identifiers are called *names*.

The Cray C compiler treats the first 255 characters of a name as significant, regardless of whether it is an internal or external name. The case of names, including external names, is significant. In Cray C++, all characters of a name are significant.

12.1.2.2 Types

Table 11, page 176 summarizes Cray C and C++ types and the characteristics of each type. *Representation* is the number of bits used to represent an object of that type. *Memory* is the number of storage bits that an object of that type occupies.

In the Cray C and C++ compilers, *size*, in the context of the `sizeof` operator, refers to the size allocated to store the operand in memory; it does not refer to representation, as specified in Table 11, page 176. Thus, the `sizeof` operator will return a size that is equal to the value in the *Memory* column of Table 11, page 176 divided by 8 (the number of bits in a byte).

Table 11. Data Type Mapping

	UNICOS/mp	
Type	Representation (bits)	Memory (bits)
`bool` (C++)	8	8
`_Bool` (C)		
`char`	8	8
`wchar_t`	32	32
`short`[1]	16	16
`int`	32	32
`long`	64	64
`long long`	64	64
`float`	32	32

1 We do not recommend using shorts because of performance penalties.

	UNICOS/mp	
Type	Representation (bits)	Memory (bits)
`double`	64	64
`long double`	128	128
`float complex`	64 (each part is 32 bits)	64
`double complex`	128 (each part is 64 bits)	128
`long double complex`	256 (each part is 128 bits)	256
Pointers	64	64

12.1.2.3 Characters

The full 8-bit ASCII code set can be used in source files. Characters not in the character set defined in the standard are permitted only within character constants, string literals, and comments. The `-h [no]calchars` option allows the use of the `@` sign and `$` sign in identifier names. For more information on the `-h [no]calchars` option, see Section 2.9.3, page 18.

A character consists of 8 bits. Up to 8 characters can be packed into a 64-bit word. A plain `char` type, one that is declared without a `signed` or `unsigned` keyword, is treated as an unsigned type.

Character constants and string literals can contain any characters defined in the 8-bit ASCII code set. The characters are represented in their full 8-bit form. A character constant can contain up to 8 characters. The integer value of a character constant is the value of the characters packed into a word from left to right, with the result right-justified, as shown in the following table:

Table 12. Packed Characters

Character constant	Integer value
`'a'`	`0x61`
`'ab'`	`0x6162`

In a character constant or string literal, if an escape sequence is not recognized, the \ character that initiates the escape sequence is ignored, as shown in the following table:

Table 13. Unrecognizable Escape Sequences

Character constant	Integer value	Explanation
'\a'	0x7	Recognized as the ASCII BEL character
'\8'	0x38	Not recognized; ASCII value for 8
'\['	0x5b	Not recognized; ASCII value for [
'\c'	0x63	Not recognized; ASCII value for c

12.1.2.4 Wide Characters

Wide characters are treated as signed 64-bit integer types. Wide character constants cannot contain more than one multibyte character. Multibyte characters in wide character constants and wide string literals are converted to wide characters in the compiler by calling the mbtowc(3) function. The current locale in effect at the time of compilation determines the method by which mbtowc(3) converts multibyte characters to wide characters, and the shift states required for the encoding of multibyte characters in the source code. If a wide character, as converted from a multibyte character or as specified by an escape sequence, cannot be represented in the extended execution character set, it is truncated.

12.1.2.5 Integers

All integral values are represented in a twos complement format. For representation and memory storage requirements for integral types, see Table 11, page 176.

When an integer is converted to a shorter signed integer, and the value cannot be represented, the result is the truncated representation treated as a signed quantity. When an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented, the result is the original representation treated as a signed quantity.

The bitwise operators (unary operator ~ and binary operators <<, >>, &, ^, and |) operate on signed integers in the same manner in which they operate on unsigned integers. The result of E1 >> E2, where E1 is a negative-valued signed integral value, is E1 right-shifted E2 bit positions; vacated bits are filled with 1s. This behavior can be modified by using the -h nosignedshifts option (see Section 2.9.4, page 18). Bits higher than the sixth bit are not ignored. Values higher than 31 cause the result to be 0 or all 1s for right shifts.

The result of the / operator is the largest integer less than or equal to the algebraic quotient when either operand is negative and the result is a nonnegative value. If the result is a negative value, it is the smallest integer greater than or equal to the algebraic quotient. The / operator behaves the same way in C and C++ as in Fortran.

The sign of the result of the percent (%) operator is the sign of the first operand.

Integer overflow is ignored. Because some integer arithmetic uses the floating-point instructions, floating-point overflow can occur during integer operations. Division by 0 and all floating-point exceptions, if not detected as an error by the compiler, can cause a run time abort.

12.1.2.6 Arrays and Pointers

An `unsigned int` value can hold the maximum size of an array. The type `size_t` is defined to be a `typedef` name for `unsigned long` in the headers: `malloc.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`. If more than one of these headers is included, only the first defines `size_t`.

A type `int` can hold the difference between two pointers to elements of the same array. The type `ptrdiff_t` is defined to be a `typedef` name for `long` in the header `stddef.h`.

If a pointer type's value is cast to a signed or unsigned `long int`, and then cast back to the original type's value, the two pointer values will compare equal.

Pointers on UNICOS/mp systems are byte pointers. Byte pointers use the same internal representation as integers; a byte pointer counts the numbers of bytes from the first address.

A pointer can be explicitly converted to any integral type large enough to hold it. The result will have the same bit pattern as the original pointer. Similarly, any value of integral type can be explicitly converted to a pointer. The resulting pointer will have the same bit pattern as the original integral type.

12.1.2.7 Registers

Use of the register storage class in the declaration of an object has no effect on whether the object is placed in a register. The compiler performs register assignment aggressively; that is, it automatically attempts to place as many variables as possible into registers.

12.1.2.8 Classes, Structures, Unions, Enumerations, and Bit Fields

Accessing a member of a union by using a member of a different type results in an attempt to interpret, without conversion, the representation of the value of the member as the representation of a value in the different type.

Members of a class or structure are packed into words from left to right. Padding is appended to a member to correctly align the following member, if necessary. Member alignment is based on the size of the member:

- For a member bit field of any size, alignment is any bit position that allows the member to fit entirely within a 64–bit word.
- For a member with a size less than 64 bits, alignment is the same as the size. For example, a `char` has a size and alignment of 8 bits; a `float` has a size and alignment of 32 bits.
- For a member with a size equal to or greater than 64 bits, alignment is 64 bits.
- For a member with array type, alignment is equal to the alignment of the element type.

A plain `int` type bit field is treated as an `signed int` bit field.

The values of an enumeration type are represented in the type `signed int` in C; they are a separate type in C++.

12.1.2.9 Qualifiers

When an object that has `volatile`-qualified type is accessed, it is simply a reference to the value of the object. If the value is not used, the reference need not result in a load of the value from memory.

12.1.2.10 Declarators

A maximum of 12 pointer, array, and/or function declarators are allowed to modify an arithmetic, structure, or union type.

12.1.2.11 Statements

The compiler has no fixed limit on the maximum number of case values allowed in a `switch` statement.

The Cray C++ compiler parses `asm` statements for correct syntax, but otherwise ignores them.

12.1.2.12 Exceptions

In Cray C++, when an exception is thrown, the memory for the temporary copy of the exception being thrown is allocated on the stack and a pointer to the allocated space is returned.

12.1.2.13 System Function Calls

See the `exit`(3) man page for a description of the form of the unsuccessful termination status that is returned from a call to `exit`(3).

12.1.3 Preprocessing

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character in the execution character set. No such character constant has a negative value. For each, `'a'` has the same value in the two contexts:

```
#if 'a' == 97
if ('a' == 97)
```

The `-I` option and the method for locating included source files is described in Section 2.19.4, page 38.

The source file character sequence in a `#include` directive must be a valid UNICOS/mp file name or path name. A `#include` directive may specify a file name by means of a macro, provided the macro expands into a source file character sequence delimited by double quotes or < and > delimiters, as follows:

```
#define myheader "./myheader.h"
#include myheader

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

The macros `__DATE__` and `__TIME__` contain the date and time of the beginning of translation. For more information, see the description of the predefined macros in Chapter 9, page 151.

The `#pragma` directives are described in Chapter 3, page 55.

Possible Requirements for non-C99 Code [A]

In order to use C code, developed under previous C compilers of the Cray C++ Programming Environment, with the `c99` command, your code may require one or more of the following modifications:

- Include necessary header files for complete function prototyping.
- Add return statements to all non-`void` functions.
- Ensure that all strings in any macro that begins with an underscore are literals. These macros cannot contain other types of strings.
- Follow C99 conventions

Previous Cray C compilers did not require you to explicitly include header files in many situations because they allowed functions to be implicitly declared. In C99, functions cannot be implicitly declared.

Libraries and Loader [B]

This appendix describes the libraries that are available with the Cray C and C++ compilers and the loader (`ld`).

B.1 Cray C and C++ Libraries Current Programming Environments

Libraries that support Cray C and C++ are automatically available when you use the `CC`, `cc`, `c89`, or `c99` command to compile your programs. These commands automatically issue the appropriate directives to load the program with the appropriate functions. If your program strictly conforms to the C or C++ standards, you do not need to know library names and locations. If your program requires other libraries or if you want direct control over the loading process, more knowledge of the loader and libraries is necessary.

The Standard Template Library (STL) is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. Be sure you have a complete understanding of templates and how they work before using them.

B.2 Loader

When you issue the `cc`(1), `CC`, `c89`, or `c99` command to invoke the compiler, and the program compiles without errors, the loader is called. Specifying the `-c` option on the command line produces relocatable object files without calling the loader. These relocatable object files can then be used as input to the loader command by specifying the file names on the appropriate loader command line.

For example, the following command line compiles a file called `target.c` and produces the relocatable object file called `target.o` in your current working directory:

```
cc -c target.c
```

You can then use file `target.o` as input to the loader or save the file to use with other relocatable object files to compile and create a linked executable file (`a.out` by default).

Because of the special code needed to handle templates, constructors, destructors, and other C++ language features, object files generated by using the CC(1) command should be linked using the CC command. To link C++ object files using the loader command (ld), the -h keep=*files* option (see Section 2.9.1, page 17) must be specified on the command line when compiling source files.

The ld command can be accessed by using one of the following methods:

- You can access the loader directly by using the ld command.
- You can let the cc, CC, c89, or c99 command choose the loader. This method has the following advantages:
  - You do not need to know the loader command line interface.
  - You do not need to worry about the details of which libraries to load, or the order in which to load them.
  - When using CC, you need not worry about template instantiation requirements or about loading the compiler-generated static constructors and destructors.

You can control the operation of the loader with the ld command line options. Refer to the ld(1) man page.

Compatibility with Older C++ Code [C]

A key feature of the Cray C++ Programming Environment 5.0 release is the Standard C++ Library. C++ code developed under the C++ Programming Environment 4.2 release or earlier can still be used with the Programming Environment 5.0 release. If your code uses nonstandard Cray C++ header files, you can continue to use your code without modification by using the `CRAYOLDCPPLIB` environment variable. Another way to use your pre-4.x code with the current Programming Environment release is to make changes to your existing code. The following sections explain how to use either of these methods.

Note: Other changes to your existing C++ code may be required because of differences between the Cray SV1 or Cray T3E systems and the Cray X1 systems. Refer to the *Cray X1 User Environment Differences*.

C.1 Use of Nonstandard Cray C++ Header Files

The Cray C++ Programming Environment 5.0 release continues to support some of the nonstandard Cray C++ header files. This allows pre-5.0 code that use these header files to be compiled without modification. These header files are available in the Standard C++ Library at the same location as they were in previous releases.

Here are the Cray nonstandard header files that can be used in Programming Environment 5.0:

- `common.h`
- `complex.h`
- `fstream.h`
- `generic.h`
- `iomanip.h`
- `iostream.h`
- `stdiostream.h`
- `stream.h`
- `strstream.h`
- `vector.h`

The nonstandard header files can be used when you set the CRAYOLDCPPLIB environment variable to a nonzero value. How to set the variable depends on the shell you are using. If you are using ksh or sh, set the variable as this example shows:

```
%export CRAYOLDCPPLIB=1
```

If you are using csh, set the variable as this example shows:

```
%setenv CRAYOLDCPPLIB 1
```

C.2 When to Update Your C++ Code

You are not required to modify your existing C++ codes in order to compile it with the Cray C++ compiler version 5.0, unless you wish to use the Standard C++ Library. One reason for migrating your code to the Standard C++ Library is that future Cray systems, such as the Cray X1 system, are unlikely to support the nonstandard Cray C++ header files of Programming Environment 3.5. Another reason for migrating is your C++ code may already contain support for the Standard C++ Library. Often, third-party code contain a configuration script that tests the features of the compiler and system before building a makefile. This script can determine whether the C++ compiler supports the Standard C++ Library.

You can use the following steps to migrate your C++ code:

1. Use the proper header files
2. Add namespace declarations
3. Reconcile header definition differences
4. Recompile all C++ files

C.2.1 Use the Proper Header Files

The first step in migrating your C++ code to use the Standard C++ Library is to ensure that it uses the correct Standard C++ Library header files. The following tables show each header file used by the C++ library version 3.5 and its likely corresponding header file in the current Standard C++ Library. The older header files do not always map directly to the new files. For example, most of the definitions of the Cray C++ version 3.5 STL alloc.h header file are contained

in the Standard C++ Library header files `memory` and `xmemory`. Anomalies, such as this are noted in the tables.

The tables divide the header files into three groups:

- Run time support library header files
- Stream and class library header files
- Standard Template Library header files

The older header file used by the run time support library originated from Edison Design Group and perform functions such as exception handling and memory allocation and deallocation. Table 14 shows the old and new header files.

Table 14. Run time Support Library Header Files

Cray C++ 3.5 header file	Standard C++ library header file
`exception.h`	`exception`
`new.h`	`new`
`stdexcept.h`	`stdexcept`
`typeinfo.h`	`typeinfo`

The header files in the stream and class library originate from AT&T and define the I/O stream classes along the string, complex, and vector classes. Table 15 shows the old and new header files.

Table 15. Stream and Class Library Header Files

Cray C++ 3.5 header file	Standard C++ Library header file
`common.h`	No equivalent header file
`complex.h`	`complex`
`fstream.h`	`fstream`
`iomanip.h`	`iomanip`
`iostream.h`	`iostream`
`stdiostream.h`	`iosfwd`
`stream.h`	Not available

Cray C++ 3.5 header file	Standard C++ Library header file
strstream.h	strstream
vector.h	vector

Note: The use of any of the stream and class library header files from Cray C++ Programming Environment 3.5 requires that you set the CRAYOLDCPPLIB environment variable. Refer to Section C.1, page 187.

Table 16 shows the old and new Standard Template Library (STL) header files.

Note: The older STL originated from Silicon Graphics Inc.

Table 16. Standard Template Library Header Files

Cray C++ 3.5 header file	Standard C++ header file
algo.h	algorithm
algobase.h	algorightm
alloc.h	memory
bvector.h	vector
defalloc.h[1]	Not available
deque.h	deque
function.h	functional
hash_map.h	hash_map
hash_set.h	hash_set
hashtable.h	xhash
heap.h	algorithm
iterator.h	iterator
list.h	list
map.h	map
mstring.h	string
multimap.h	map
multiset.h	set

1 This header file was deprecated in the Cray C++ Programming Environment 3.5 release.

Cray C++ 3.5 header file	Standard C++ header file
pair.h	pair
pthread_alloc.h	No equivalent header file
rope.h	rope
ropeimpl.h	rope
set.h	set
slist.h	slist
stack.h	stack
stl_config.h	The Standard C++ Library does not need the STL configuration file.
tempbuf.h	memory
tree.h	xtree
vector.h	vector

C.2.2 Add Namespace Declarations

The second step in migrating to the Standard C++ Library is adding namespace declarations. Most classes of the Standard C++ Library are declared under the `std` namespace, so this usually requires that you add this statement to the existing code: `using namespace std`. For example, the following program returns an error when it is compiled with previous versions of the Standard C++ Library:

```
%cat hello.C
#include <iostream>

main() { cout << "hello world\n"; }

%CC hello.C

  CC-20 CC: ERROR File = hello.C, line = 2
  The identifier "cout" is undefined.
  main() { cout <<"hello world\n" ; }
  ^
  Total errors detected in hello.C: 1
%
```

When you add `using namespace std;` to the example program, it compiles without error:

```
%cat hello.C
     #include <iostream>
     using namespace std;
     main() { cout << "hello world\n"; }

%CC hello.C
%a.out
hello world
%
```

C.2.3 Reconcile Header Definition Differences

The most difficult process of migrating to the Standard C++ Library is reconciling the differences between the definitions of the Cray C++ version 3.5 header files and the Standard Cray C++ library header files. For example, the definitions for the complex class differs. In Cray C++ version 3.5, the complex class has real and imaginary components of type double. The Standard C++ Library defines the complex class as a template class, where the user defines the data type of the real and imaginary components.

For example, here is a program written with the Cray C++ version 3.5 header files:

```
%cat complex.C
     #include <iostream.h>
     #include <complex.h>

     main() {
        complex C(1.0, 2.0);
        cout << "C = " << C << endl;
     }

%env CRAYOLDCPPLIB=1 CC complex.C
%a.out
C = ( 1, 2)
```

An equivalent program that uses the Standard C++ Library appears as:

```
%cat complex.C
     #include <iostream>
```

```
     #include <complex.h>
     using namespace std;

     main() {
        complex<double> C(1.0, 2.0);

        cout << "C = " << C << endl;
     }
%CC complex.C
%a.out
C = (1,2)
```

C.2.4 Recompile All C++ Files

Finally, when all of the source files that use the Standard C++ Library header files can be built, you must recompile all C++ source files that belong to the program using only the Standard C++ Library.

Cray C and C++ Dialects [D]

This appendix details the features of the C and C++ languages that are accepted by the Cray C and C++ compilers, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other environments.

D.1 C++ Language Conformance

The Cray C++ compiler accepts the C++ language as defined by the *ISO/IEC 14882:1998* standard, with the exceptions listed in Section D.1.1, page 195.

The Cray C++ compiler also has a `cfront` compatibility mode, which duplicates a number of features and bugs of `cfront`. Complete compatibility is not guaranteed or intended. The mode allows programmers who have used `cfront` features to continue to compile their existing code (see Section 3.5, page 57). Command line options are also available to enable and disable anachronisms (see Section D.2, page 199) and strict standard-conformance checking (see Section D.3, page 200, and Section D.4, page 201). The command line options are described in Chapter 2, page 7.

D.1.1 Unsupported and Supported C++ Language Features

The `export` keyword for templates is not supported. It is defined in the *ISO/IEC 14882:1998* standard, but is not in traditional C++ .

The following features, which are in the ISO / IEC 14882:1998 standard but not in traditional C++[1], are supported:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a `?` operator, or as an operand of the `&&`, `||`, or `!` operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.

1 As defined in *The Annotated C++ Reference Manual* (ARM), by Ellis and Stroustrup, Addison Wesley, 1990.

- A global-scope qualifier is allowed in member references of the form `x.::A::B and p->::A::B`.
- The precedence of the third operand of the `?` operator is changed.
- If control reaches the end of the `main()` routine, and the `main()` routine has an integral return type, it is treated as if a `return 0;` statement was executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary that is created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an `rvalue`.
- Qualification conversions such as conversion from `T**` to `T const * const` are allowed.
- Digraphs are recognized.
- Operator keywords (for example, `and` or `bitand`) are recognized.
- Static data member declarations can be used to declare member constants.
- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (run time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (within `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.

- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including using declarations and directives. Access declarations are broadened to match the corresponding using declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare nonconverting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using `template <>`) is implemented.
- Cv qualifiers are retained on `rvalues` (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between process overlay directives (PODs) and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- A `typedef` name can be used in an explicit destructor call.
- Placement delete is supported.
- An array allocated via a placement new can be deallocated via delete.
- `enum` types are considered to be nonintegral types.
- Partial specification of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.

- It is possible to overload operators using functions that take `enum` types and no class types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `::  template` (and `->template`, etc.) is supported.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated. Likewise for a similar reference to a static data member. The ARM specifies that the left operand is not evaluated in such cases.
- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- `reinterpret_cast` allows casting a pointer to a member of one class to a pointer to a member of another class even when the classes are unrelated.
- Two-phase name binding in templates as described in the *Working Paper* is implemented.
- Putting a `try/catch` around the initializers and body of a constructor is implemented.
- Template `template` parameters are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- `extern inline` functions are supported.
- Covariant return types on overriding virtual functions are supported.

D.2 C++ Anachronisms Accepted

C++ anachronisms are enabled by using the `-h anachronisms` command line option (see Section 2.6.7, page 14). When anachronisms are enabled, the following anachronisms are accepted:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized by using the default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is allowed. This is only allowed if anachronisms are enabled and the `assignment to this` configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a nonnested class name if no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an `rvalue` of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when checking for compatibility, therefore, the following statements declare the overloading of two functions named `f`:

```
int f(int);

int f(x) char x; { return  x; }
```

Note: In C, this code is legal, but has a different meaning. A tentative declaration of `f` is followed by its definition.

D.3 Extensions Accepted in Normal C++ Mode

The following C++ extensions are accepted (except when strict standard conformance mode is enabled, in which case a warning or caution message may be issued):

- A `friend` declaration for a class can omit the `class` keyword, as shown in the following example:

```
class B;
class A  {
   friend B;   // Should be "friend class B"
};
```

- Constants of scalar type can be defined within classes, as shown in the following example:

```
class A {
   const int size=10;
   int a[size];
};
```

- In the declaration of a class member, a qualified name can be used, as shown in the following example:

```
struct A {
   int A::f();   // Should be int f();
}
```

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a “default” assignment operator; that is, such a declaration blocks the implicit generation of a copy assignment operator. This is `cfront` behavior that is known to be relied upon in at least one widely used library. Here is an example:

```
struct A { };
struct B : public A {
   B& operator=(A&);
};
```

By default, as well as in `cfront` compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode, `B::operator=(A&)` is not a copy assignment operator and `B::operator=(const B&)` is implicitly declared.

- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. The following is an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)()         // pf points to an extern "C++" function
             = &f;   // error unless implicit conversion allowed
```

- The `?` operator, for which the second and third operands are string literals or wide string literals, can be implicitly converted to one of the following:

```
char *
wchar_t *
```

In C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension:

```
char *p = x ? "abc" : "def";
```

D.4 Extensions Accepted in C or C++ Mode

The following extensions are accepted in C or C++ mode except when strict standard conformance modes is enabled, in which case a warning or caution message may be issued.

- The special `lint` comments `/*ARGSUSED*/`, `/*VARARGS*/` (with or without a count of nonvarying arguments), and `/*NOTREACHED*/` are recognized.
- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.
- Bit fields can have base types that are `enum` or integral types in addition to `int` and `unsigned int`. This corresponds to A.6.5.8 in the ANSI Common Extensions appendix.

- `enum` tags can be incomplete as long as the tag name is defined and resolved by specifying the brace-enclosed list later.
- An extra comma is allowed at the end of an `enum` list.
- The final semicolon preceding the closing of a `struct` or `union` type specifier can be omitted.
- A label definition can be immediately followed by a right brace (}). (Normally, a statement must follow a label definition.)
- An empty declaration (a semicolon preceded by nothing) is allowed.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union does not need to be enclosed in braces. ANSI C requires braces.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it.
- The address of a variable with register storage class may be taken.
- In an integral constant expression, an integer constant can be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (for example, `short short` or `unsigned unsigned`) the redundancy is ignored.
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name can be redeclared in the same scope with the same type.
- Dollar sign ($) and at sign (@) characters can be accepted in identifiers by using the `-h calchars` command line option. This is not allowed by default.
- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one token that is not valid. If the `-h conform` option is specified, the pp-number syntax is used.
- Assignment and pointer differences are allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size (for example, `int *` and `long *`). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **`

to `const int **`). Comparisons and pointer difference of such pairs of pointer types are also allowed.

- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, these are allowed by some operators, and not by others (generally, where it does not make sense).
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. This extension is not allowed in C++ mode.
- A pointer to `void` can be implicitly converted to or from a pointer to a function type.
- External entities declared in other scopes are visible:

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */ }
```

- In C mode, end-of-line comments (`//`) are supported.
- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.
- The `fortran` keyword. For more information, see Section 8.2, page 148.
- Cray hexadecimal floating point constants. For more information, see Section 8.3, page 148.

D.5 C++ Extensions Accepted in `cfront` Compatibility Mode

The `cfront` compatibility mode is enabled by the `-h cfront` command-line option. The following extensions are accepted in `cfront` compatibility mode:

- Type qualifiers on the `this` parameter are dropped in contexts such as in the following example:

```
struct A {
   void f() const;
};
void (A::*fp)() = &A::f;
```

  This is a safe operation. A pointer to a `const` function can be put into a pointer to non-`const`, because a call using the pointer is permitted to modify

the object and the function pointed to will not modify the object. The opposite assignment would not be safe.

- Conversion operators that specify a conversion to `void` are allowed.
- A nonstandard `friend` declaration can introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in default mode, however, in `cfront` mode the declaration can also introduce a new type name. An example follows:

```
struct A {
   friend B;
};
```

- The third operator of the `?` operator is a conditional expression instead of an assignment expression.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example:

```
int *p;
const int *&r = p;   // No temporary used
```

- A reference can be initialized to `NULL`.
- Because `cfront` does not check the accessibility of types, access errors for types are issued as warnings instead of errors.
- When matching arguments of an overloaded function, a `const` variable with a value of 0 is not considered to be a null pointer constant. In general, in overload resolution, a null pointer constant must be spelled “0” to be considered a null pointer constant (e.g., ’\0’ is not considered a null pointer constant).
- An alternate form of declaring pointer-to-member-function variables is supported, as shown in the following example:

```
struct A {
 void f(int);
 static void sf(int);
 typedef void A::T3(int);  // nonstd typedef decl
 typedef void T2(int);     // std typedef
};
typedef void A::T(int);   // nonstd typedef decl
T* pmf = &A::f;            // nonstd ptr-to-member decl
A::T2* pf = A::sf;         // std ptr to static mem decl
A::T3* pmf2 = &A::f;       // nonstd ptr-to-member decl
```

In this example, `T` is construed to name a function type for a nonstatic member function of class `A` that takes an `int` argument and returns `void`; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to the following single standard pointer-to-member declaration:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally not valid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. `cfront` version 2.1 accepts declarations, such as `T`, even when `A` is an incomplete type; so this case is also accepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:

```
class B { protected: int i; };
class D : public B { void mf()};

void D::mf() {
 int B::* pmi1 = &B::i;  // error, OK in cfront mode
 int D::* pmi2 = &D::i;  // OK
}
```

  Note: Protected member access checking for other operations (such as everything except taking a pointer-to-member address) is done normally.

- The destructor of a derived class can implicitly call the private destructor of a base class. In default mode, this is an error but in `cfront` mode it is reduced to a warning. For example:

```
class A {
   ~A();
};
class B : public A {
   ~B();
};
B::~B(){}              // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword*(*identifier* ...) is treated as an argument. For example:

  ```
  class A { A(); };
  double d;
  A x(int(d));
  A(x2);
  ```

  By default, `int(d)` is interpreted as a parameter declaration (with redundant parentheses), and so `x` is a function; but in `cfront` compatibility mode `int(d)` is an argument and `x` is a variable.

  The declaration `A(x2)` is also misinterpreted by `cfront`. It should be interpreted as the declaration of an object named `x2`, but in `cfront` mode it is interpreted as a function style cast of `x2` to the type `A`.

  Similarly, the following declaration declares a function named `xzy`, that takes a parameter of type function taking no arguments and returning an `int`. In `cfront` mode, this is interpreted as a declaration of an object that is initialized with the value `int()`, which evaluates to 0.

  ```
  int xyz(int());
  ```

- A named bit field can have a size of 0. The declaration is treated as though no name had been declared.
- Plain bit fields (such as bit fields declared with a type of `int`) are always signed.
- The name given in an elaborated type specifier can be a `typedef` name that is the synonym for a class name. For example:

  ```
  typedef class A T;
  class T *pa;            // No error in cfront mode
  ```

- No warning is issued on duplicate size and sign specifiers, as shown in the following example:

```
short short int i;  // No warning in cfront mode
```

- Virtual function table pointer-update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further derived class. For example:

```
struct A {
  virtual void f() {}
  A() {}
  ~A() {}
};
struct B : public A {
  B() {}
  ~B() {f();}        // Should call A::f according to ARM 12.7
};
struct C : public B {
  void f() {}
} c;
```

  In `cfront` compatibility mode, `B::~B` calls `C::f`.

- An extra comma is allowed after the last argument in an argument list. For example:

```
f(1, 2, );
```

- A constant pointer-to-member function can be cast to a pointer-to-function, as in the following example. A warning is issued.

```
struct A {int f();};
main () {
  int (*p)();
  p = (int (*)())A::f;          // Okay, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value like C structures, and the destructor is not called on the copy. In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Because the argument is passed by value instead of by address, code like this compiled in `cfront` mode is not calling-sequence compatible with the same code

compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.
- When an unnamed class appears in a `typedef` declaration, the `typedef` name may appear as the class name in an elaborated type specifier. For example:

```
typedef struct { int i, j; } S;
struct S x;                 // No error in cfront mode
```

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier. For example:

```
class A {
  void f(int) const;
  static void f(int); // No error in cfront mode
};
```

- The scope of a variable declared in the `for-init-statement` is the scope to which the `for` statement belongs. For example:

```
int f(int i) {
  for (int j = 0; j < i; ++j) { /* ... */ }
  return j; // No error in cfront mode
}
```

- Function types differing only in that one is declared `extern "C"` and the other `extern "C++"` can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
void f(PF);
void f(PCF);
```

  By contrast, in standard C++, PF and PCF are different and incompatible types; PF is a pointer to an `extern "C++"` function whereas PCF is a pointer to an `extern "C"` function; and the two declarations of `f` create an overload set.

- Functions declared `inline` have internal linkage.
- `enum` types are regarded as integral types.

- An uninitialized const object of non-POD class type is allowed even if its default constructor is implicitly declared as in the following example:

  ```
  struct A { virtual void f(); int i; };
  const A a;
  ```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.

- If the user declares an `operator=` function in a class, but not one that can serve as the default `operator=`, and bitwise assignment could be done on the class, a default `operator=` is not generated. Only the user-written `operator=` functions are considered for assignments, so bitwise assignment is not done.

Compiler Messages [E]

This appendix describes how to use the message system to control and use messages issued by the compiler. Explanatory texts for messages can be displayed online through the use of the `explain` command.

E.1 Expanding Messages with the `explain` Command

You can use the `explain` command to display an explanation of any message issued by the compiler. The command takes as an argument, the message number, including the number's prefix. The prefix for Cray C and C++ is `CC`.

In the following sample dialog, the `cc`(1) command invokes the compiler on source file `bug.c`. Message `CC-24` is displayed. The `explain` command displays the expanded explanation for this message.

```
> cc bug.c
CC-24 cc: ERROR File = bug.c, Line = 1
   An invalid octal constant is used.

   int i = 018;
             ^

1 error detected in the compilation of "bug.c".
> explain CC-24

An invalid octal constant is used.

Each digit of an octal constant must be between 0 and 7,
inclusive. One or more digits in the octal constant on the
indicated line are outside of this range. To avoid issuing
an error for each erroneous digit, the constant will be treated
as a decimal constant. Change each digit in the octal constant
to be within the valid range.
```

E.2 Controlling the Use of Messages

This section summarizes the command line options that affect the issuing of messages from the compiler.

E.2.1 Command Line Options

Option	Description
`-h errorlimit[=`*n*`]`	Specifies the maximum number of error messages the compiler prints before it exits.
`-h [no]message=`*n*`[:...]`	Enables or disables the specified compiler messages, overriding `-h msglevel`.
`-h msglevel_`*n*	Specifies the lowest severity level of messages to be issued.
`-h report=`*args*	Generates optimization report messages.

E.2.2 Environment Options for Messages

The following environment variables are used by the message system.

Variable	Description
`NLSPATH`	Specifies the default value of the message system search path environment variable.
`LANG`	Identifies your requirements for native language, local customs, and coded character set with regard to the message system.
`MSG_FORMAT`	Controls the format in which you receive error messages.

E.2.3 `ORIG_CMD_NAME` Environment Variable

You can override the command name printed in the message. If the environment variable `ORIG_CMD_NAME` is set, the value of `ORIG_CMD_NAME` is used as the command name in the message. This functionality is provided for use with shell scripts that invoke the compiler. By setting `ORIG_CMD_NAME` to the name of the

script, any message printed by the compiler appears as though it was generated by the script. For example, the following C shell script is named newcc:

```
#
setenv ORIG_CMD_NAME 'basename $0'
cc $*
```

A message generated by invoking newcc resembles the following:

```
CC-8 newcc: ERROR File = x.c, Line = 1
  A new-line character appears inside a string literal.
```

Because the environment variable ORIG_CMD_NAME is set to newcc, this appears as the command name instead of cc(1) in this message.

Caution: The ORIG_CMD_NAME environment variable is not part of the message system. It is supported by the Cray C and C++ compilers as an aid to programmers. Other products, such as the Fortran compiler and the loader, may support this variable. However, you should not rely on support for this variable in any other product.

You must be careful when setting the environment variable ORIG_CMD_NAME. If you set ORIG_CMD_NAME inadvertently, the compiler may generate messages with an incorrect command name. This may be particularly confusing if, for example, ORIG_CMD_NAME is set to newcc when the Fortran compiler prints a message. The Fortran message will look as though it came from newcc.

E.3 Message Severity

Each message issued by the compiler falls into one of the following categories of messages, depending on the severity of the error condition encountered or the type of information being reported.

Category	Meaning
COMMENT	Inefficient programming practices.
NOTE	Unusual programming style or the use of outmoded statements.
CAUTION	Possible user error. Cautions are issued when the compiler detects a condition that may cause the program to abort or behave unpredictably.

WARNING	Probable user error. Indicates that the program will probably abort or behave unpredictably.
ERROR	Fatal error; that is, a serious error in the source code. No binary output is produced.
INTERNAL	Problems in the compilation process. Please report internal errors immediately to the system support staff, so a Software Problem Report (SPR) can be filed.
LIMIT	Compiler limits have been exceeded. Normally you can modify the source code or environment to avoid these errors. If limit errors cannot be resolved by such modifications, please report these errors to the system support staff, so that an SPR can be filed.
INFO	Useful additional information about the compiled program.
INLINE	Information about inline code expansion performed on the compiled code.
SCALAR	Information about scalar optimizations performed on the compiled code.
VECTOR	Information about vectorization optimizations performed on the compiled code.
STREAM	Information about the MSP optimizations performed on the compiled code.
OPTIMIZATION	Information about general optimizations.

E.4 Common System Messages

The errors in the following list can occur during the execution of a user program. The operating system detects them and issues the appropriate message. These errors are not detected by the compiler and are not unique to C and C++ programs; they may occur in any application program written in any language.

- `Operand Range Error`

  An operand range error occurs when a program attempts to load or store in an area of memory that is not part of the user's area. This usually occurs when an invalid pointer is dereferenced.

- `Program Range Error`

A program range error occurs when a program attempts to jump into an area of memory that is not part of the user's area. This may occur, for example, when a function in the program mistakenly overwrites the internal program stack. When this happens, the address of the function from which the function was called is lost. When the function attempts to return to the calling function, it jumps elsewhere instead.

- `Error Exit`

  An error exit occurs when a program attempts to execute an invalid instruction. This error usually occurs when the program's code area has been mistakenly overwritten with words of data (for example, when the program stores in a location pointed to by an invalid pointer).

Intrinsic Functions [F]

The C and C++ intrinsic functions either allow for direct access to some hardware instructions or result in generation of inline code to perform some specialized functions. These intrinsic functions are processed completely by the compiler. In many cases, the generated code is one or two instructions. These are called *functions* because they are invoked with the syntax of function calls.

To get access to the intrinsic functions, the Cray C++ compiler requires that either the `intrinsics.h` file be included or that the intrinsic functions that you want to call be explicitly declared. If you explicitly declare an intrinsic function, the declaration must agree with the documentation or the compiler treats the call as a call to a normal function, not the intrinsic function. The `-h nointrinsics` command line option causes the compiler to treat these calls as regular function calls and not as intrinsic function calls.

The types of the arguments to intrinsic functions are checked by the compiler, and if any of the arguments do not have the correct type, a warning message is issued and the call is treated as a normal call to an external function. If your intention was to call an external function with the same name as an intrinsic function, you should change the external function name. The names used for the Cray C intrinsic functions are in the name space reserved for the implementation.

> **Note:** Several of these intrinsic functions have both a vector and a scalar version. If a vector version of an intrinsic function exists and the intrinsic is called within a vectorized loop, the compiler uses the vector version of the intrinsic. See the appropriate intrinsic function man page for details on whether it has a vector version.

The following sections groups the C and C++ intrinsics according to function and provides a brief description of each intrinsic in that group. See the corresponding man page for more information.

F.1 Atomic Memory Operations

The following intrinsics perform various atomic memory operations:

> **Note:** In this discussion, an object is an entity that is referred to by a pointer. A value is an actual number, bit mask, etc. that is not referred to by a pointer.

Intrinsic	Description
_amo_aadd	Adds a value to an object that is referred to by a pointer and stores the results in the object.
_amo_aax	ANDs a value and an object that is referred to by a pointer, XORs the result with a third value, and stores the results in the object.
_amo_afadd	Adds a value to an object that is referred to by a pointer and stores the result in the object. The intrinsic returns the original value of the object.
_amo_afax	ANDs a value with an object that is referred to by a pointer, XORs the result with a second value, and stores the result in the object. The intrinsic returns the original value of the object.
_amo_acswap	(Compare and swap) Compares an object that is referenced by a pointer against a value. If equal, a specified value is stored in the object. The intrinsic returns the original value of object.

F.2 BMM Operations

The following intrinsics perform operations on the BMM:

_mtilt	Inverts a bit matrix
_mclr	Logically undefines the BMM unit.
_mld	Loads the BMM functional unit with a matrix vector in transposed form.
_mldmx	Combines the load and multiply functions.
_mmx	Performs a bit matrix multiply.
_mul	Unloads the bit matrix function unit.

F.3 Bit Operations

The following intrinsics copy, count, or shift bits or computes the parity bit:

_dshiftl	Move the left most n bits of an integer into the right side of another integer, and return that integer.

_dshiftr	Move the right most n bits of an integer into the left side of another integer and return that integer.
_pbit	Copies the rightmost bit of a word to the n^{th} bit, from the right, of another word.
_pbits	Copies the rightmost m bits of a word to another word beginning at bit n.
_poppar	Computes the parity bit for a variable.
_popcnt _popcnt32 _popcnt64	Counts the number of set bits in 32-bit and 64-bit integer words.
_leadz _leadz32 _leadz64	Counts the number of leading 0 bits in 32-bit and 64-bit integer words.
_gbit	_gbit returns the value of the n^{th} bit from the right.
_gbits	Returns a value consisting of m bits extracted from a variable, beginning at n^{th} bit from the right.

F.4 Function Operations

These intrinsics return information about function arguments:

_argcount	Returns the number of arguments explicitly passed to a function, excluding any "hidden" arguments added by the compiler.
_numargs	Returns the total number of words in the argument list passed to the function including any "hidden" arguments added by the compiler.

F.5 Mask Operations

These intrinsics create bit masks:

_mask	Creates a left-justified or right-justified bit mask with all bits set to 1.
_maskl	Returns a left-justified bit mask with i bits set to 1.

`_maskr`	Returns a right-justified bit mask with `i` bits set to 1.

F.6 Memory Operations

These intrinsics assures that memory references complete or synchronizes memory:

`_cmr`	Assures completion of memory references.
`_gsync`	Performs global synchronization of all memory.

F.7 Miscellaneous Operations

The following intrinsics perform various functions:

`_EX`	Exits normally.
`_int_mult_upper`	Multiplies integers and returns the uppermost bits. Refer to the `int_mult_upper`(3i) man page.
`_ranf`	`_ranf`, compute a pseudo-random floating-point number ranging from 0.0 through 1.0.
`_rtc`	Return a real-time clock value expressed in clock ticks.

F.8 Streaming Operations

These intrinsics return streaming information:

`__sspid`	Indicates which SSP is being used by the code.
`__streaming`	Indicates whether the code is capable of multistreaming.

Glossary

address

Identification (such as a label, number, or name) that designates a particular location in storage or any other data destination or source, including a node on a network. Computer memory addresses may be considered either as relative (to some value contained in an offsetting register) or absolute (counting from the beginning of memory).

address space

A program's location in memory.

application node

A node that is used to run user applications. Application nodes are best suited for executing parallel applications and are managed by the strong political scheduling and gang scheduling mechanism `psched`. See also *OS node*; *support node*.

assembler

A computer program that creates a machine language program from a symbolic language program. The assembler substitutes machine operation codes for symbolic operation codes and substitutes absolute or relocatable addresses for symbolic instructions. See also *interpreter*.

assembly language

A programming language that uses symbolic notation to represent machine language instructions. It is a low-level language that is closely related to the internal architecture of the computer on which it runs.

atomic

Not interruptible. An atomic operation occurs as a logical unit.

atomic update

When multiple tasks may update the same data location at the same time, an atomic update ensures that more than one task cannot update an element simultaneously.

barrier

An obstacle within a program that provides a mechanism for synchronizing tasks. When a task encounters a barrier, it must wait until all specified tasks reach the barrier.

barrier synchronization

1. An event initiated by software that prevents cooperating tasks from continuing to issue new program instructions until all of the tasks have reached the same point in the program. 2. A feature that uses a barrier to synchronize the processors within a partition. All processors must reach the barrier before they can continue the program.

bit mask

An arrangement of bits in a word or register that assigns logical significance to corresponding bits in other words.

cache hit

A memory reference to a data object already in cache. Such references are closer and faster than references to data objects in main memory.

code

1. A system of characters and rules that represent information in a form the computer can interpret. 2. Translation of a problem into a computer language.

command

1. A directive to the shell to perform a particular task. In the classic sense, a command (or command line) can be made up of the name of an executable file followed by one or more operands (options or other arguments) that supply information to that executable file necessary to complete its processing. 2. The name of an executable file, which can be a compiled program, shell built-in command, or shell script.

command line

A text string that is composed of the command name itself, followed by options and arguments to the command. It can be a sequence of nonblank arguments separated by blanks or tabs entered by a user.

CPU

A general term for the part of a computer that executes programs. For Cray X1 systems, a multistreaming processor (MSP) by default or a single-streaming processor (SSP) if a program is compiled for an SSP. See also MSP; *processor*; SSP.

Cray C

The Cray implementation of the C language conforming to American National Standard Institute (ANSI) standard X3.159-1989. Standard C is the language in which most Cray system software is written. C is described as a middle-level language, having many characteristics of high-level languages, such as Fortran, while offering direct manipulation of binary data in the manner of assembly code.

Cray Fortran Compiler

The compiler that translates Fortran programs into Cray object files. The Cray Fortran Compiler fully supports the Fortran language through the Fortran 95 Standard, ISO/IEC 1539-1:1997. Selected features from the proposed Fortran 2000 Standard are also supported.

Cray X1

The Cray system that combines the single-processor performance and single-shared address space of Cray parallel vector processor (PVP) systems with the highly scalable microprocessor-based architecture that is used in Cray T3E systems. Cray X1 systems utilize powerful vector processors, shared memory, and a modernized vector instruction set in a highly scalable configuration that provides the computational power required for advanced scientific and engineering applications.

CrayDoc

Cray's documentation system for accessing and searching Cray books, man pages, and glossary terms from a web browser. CrayDoc runs on any operating system based on a UNIX or Linux operating system.

CRInform

An online technical-assistance and problem-reporting service for subscribing Cray customers.

deferred implementation

The label used to introduce information about a feature that will not be implemented until a later release.

distributed memory

1. Memory in which each processor has a separate share of the total memory. 2. Memory that is physically distributed among several modules.

element

1. One entry in an array. 2. A discrete item (a word) within a vector register.

environment variable

A variable that stores a string of characters for use by your shell and the processes that execute under the shell. Some environment variables are predefined by the shell, and others are defined by an application or user. Shell-level environment variables let you specify the search path that the shell uses to locate executable files, the shell prompt, and many other characteristics of the operation of your shell. Most environment variables are described in the ENVIRONMENT VARIABLES section of the man page for the affected command.

Etnus TotalView

A symbolic source-level debugger designed for debugging the multiple processes of parallel Fortran, C, or C++ programs.

executable file

A file in a format that can be executed by a computer.

folding

A basic compiler optimization that converts operations on constants to simpler forms (such as folding 1*3 into 3)

function

In C, any called code; in Fortran, a sequence of instructions that returns a value.

functional unit

1. A combination of elements grouped to perform a generally elementary computer function. 2. Hardware within a processor that performs specialized functions. Functional units perform arithmetic, logical, shift, and other functions.

header file

A file in a specific format that more than oneprogram (such as compilers, assemblers, and system utilities) uses. A header file contains definitions of symbols, variables, types, and so on. You can include this file in the compilation of a program. See also *include file*.

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the HEAP directive in the `ld`(1) man page.

include file

When an application (such as a compiler) reads a file, it may be instructed to include (or read in) another file and treat this include file as if it were part of the original named file.

induction variable

A variable in a loop that controls the execution of that loop. For example, in Fortran, I is the induction variable in `DO I = 1, 100`, and in C++, i is the induction variable in `for (i=0;i<100;i++)`.

inlining

Replacing a user subroutine call (or function call) with the code for the procedure itself to eliminate calling overhead.

inner loop

See *stripmining*.

instruction set

The low-level, basic set of the instructions executed by a computer.

interpreter

A program that translates each source language statement into a sequence of machine instructions, executes the instructions, and then translates the next source statement.

intrinsic function

A function is a self-contained bit of code that can be called from a point in other code to do a particular job. Callable code exists so that any job that is done repeatedly in a program is specified only once. If the needed code is particular to your application, you write it yourself and include it in the compilation. If the function is needed by many programs, it is put into a library. These two categories are called external routines, because they are outside the calling routine and require an elaborate mechanism for calling. If the function is almost universally needed, it is put right into the compiler, and the compiler generates code inline when it encounters that call. It saves the overhead of an external call. An example of an intrinsic function is SQRT (square root) in Fortran.

IRIX

The version of the UNIX System V operating system that is produced by Silicon Graphics, Inc.

kind

Data representation (for example, single precision, double precision). The kind of a type is referred to as a kind parameter or kind type parameter of the type. The kind type parameter KIND indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types.

LAPACK

A public-domain library of subroutines for solving dense linear algebra problems, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems.

leading zero

A functional unit that counts from left to right the number of zeroes before the first set bit (1 bit) in a word.

library function

A small routine (not an operating system function) that is called from within a program and performs a specific task; for example, the SQRT library routine finds a square root value of an argument.

link

The action of loaders. Executable files and the libraries and data they access are loaded into memory during the load step. Links are created among modules that must access each other. See also *load*.

load

To create a binary executable file (an executable) from a binary relocatable object file (the object). This process adds library subprograms to the object and resolves the external references among subprograms. Executable files and the libraries and data they access are loaded into memory during the load step. Links are created among modules that must access each other. The command that performs a load is called a link-edit loader, or simply a loader.

loader

A generic term for the system software product that loads a compiled or assembled program into memory and prepares it for execution.

loading

The placement of instructions and data in memory so that execution can begin. Loader input is obtained from one or more object files and libraries. On load completion, the execution of the program in the memory field is optionally initiated. Loading also may involve load-related services such as generating a loader map, presetting unused memory to a user-specified value, and generating overlays.

local

A type of scope in which variables are accessible only to a particular part of a program (usually one module).

loop fusion

An optimization that takes the bodies of loops with identical iteration counts and fuses them into a single loop with the same iteration count.

loop interchange

An optimization that changes the order of loops within a loop nest, to achieve stride minimization or eliminate data dependencies.

loop invariant

A value that does not change between iterations of a loop.

loop unrolling

An optimization that increases the step of a loop and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and reduce memory access time.

loopmark listing

A listing that is generated by invoking the Cray Fortran Compiler with the `-rm` option or invoking the C or C++ compiler with the `-h list` option. The loopmark listing displays what optimizations were performed by the compiler and tells you which loops were vectorized, streamed, unrolled, interchanged, and so on.

macro

An assembly language statement from which a series of machine language instructions is generated.

makefile

A file that tells the `make` command what to do. The `make` utility executes commands in the makefile to update one or more targets, which are typically programs. The `make` utility examines time relationships and updates those derived files that have modified times earlier than the modified times of the files (called prerequisites) from which they are derived. A description file (`makefile`) contains a description of the relationships between files and the commands that must be executed to update the targets to reflect changes in their prerequisites. Each specification, or rule, consists of a target, optional prerequisites, and optional commands to be executed when a prerequisite is newer than the target.

man page

A form of online documentation about a command, library, system call, file format, or topic; contains information such as the syntax for invoking a

command, a list of options and descriptions, examples, and sources for more information.

mapping

An allocation of processes to processors. (Allocating work to processes is usually called scheduling.)

mask

A machine word that specifies the parts of another machine word on which to operate. Bit masks are commonly used with the AND, OR, and exclusive OR instructions.

message passing

A programming method in which explicit messages (containing data) are sent between tasks. Cray has implemented the message-passing programming method through the Message Passing Interface and the shared memory (SHMEM) routines. See also MPI; *SHMEM*.

node

The configurable scalable building block for a Cray X1 mainframe. The actual hardware contents of a node are housed in four multichip modules (MCMs). This is the conceptual or software configuration view of a hardware unit called a node module. Physically, all nodes are the same; software controls how a node is used, such as for an OS node, application node, or support node. See also *application node*; MCM, MSP, *node module*; *OS node*; SSP; *support node*.

node module

The physical node in a Cray X1 system.

OpenMP

An industry-standard, portable model for shared memory parallel programming.

OS node

The node that provides kernel-level services, such as system calls, to all support nodes and application nodes. See also *application node*; *node*; *support node*.

overindexing

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not always, leads to referencing a storage location outside of the entire array.

page size

The smallest unit of memory addressable through the Translation Lookaside Buffer (TLB). On a Cray X1 system, the base page size is 65,536 bytes, but larger page sizes (up to 4,294,967,296 bytes) are also available.

parallel region

See *serial region*.

pattern matching

Recogizing a common code pattern and replacing it with a call to a functionally equivalent library routine.

processing element

A software concept pertaining to a process within an application.

processor

A part of a computer that executes programs. See also MSP; SSP.

reduction

The process of transforming an expression according to certain reduction rules. The most important forms are beta reduction (application of a lambda abstraction to one or more argument expressions) and delta reduction (application of a mathematical function to the required number of arguments). An evaluation strategy (or reduction strategy) determines which part of an expression to reduce first. There are many such strategies. Also called contraction.

reduction loop

A loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

relocatable binary module

A binary module that cannot be executed because absolute machine addresses have not yet been set by the loader/linker; addresses are still only relative to others in the module, so they must be relocated by the operating system to actual hardware memory addresses before execution can occur.

scalable

Capable of being increased in size, or capable of delivering an increase in performance that is proportional to an increase in size.

scalar processing

A form of fine-grain serial processing whereby iterative operations are performed sequentially on the elements of an array, with each iteration producing one result. Compare to vector processing .

scope

The region of a program in which a variable is defined and can be referenced.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

secondary cache

A cache for data or instructions that is one step more distant from a processor than the primary cache. The secondary cache is generally larger but has higher latency than the primary cache.

serial region

An area within a program in which only the master task is executing. Its opposite is a parallel region.

serialize

To put potentially concurrent operations in a strictly sequential order.

SHMEM

A library of optimized subroutines that take advantage of shared memory to move data between the memories of processors. The routines can either be used by themselves or in conjunction with another programming style such as Message Passing Interface.

shortloop

A loop that is vectorized but that has been determined by the compiler to have trips less than or equal to the maximum vector length. In this case, the compiler deletes the loop to the top of the loop. If the shortloop directive is used or the trip count is constant, the top test for number of trips is deleted. A shortloop is more efficient than a conventional loop.

stack

1. A data structure that provides a dynamic, sequential data list that can be accessed from either end; a last-in, first-out (push-down, pop-up) stack is accessed from just one end. 2. The call stack consists of stack frames that hold return locations for called routines, routine arguments, local variables, and saved registers.

store

A hardware operation in which a processor transfers data into central memory from one or more processor registers. A remote store operation transfers data into a remote processor's memory from the local processor's processor registers or memory.

strength reduction

Replacing expressions by algebraic equivalents that require less time to compute. For example, a multiplication do loop is converted by strength reduction into a series of additions, because addition requires less time to compute than multiplications. Similar transformations are performed with exponentiations and other operators.

stripmining

A single-processor optimization technique in which arrays, and the program loops that reference them, are split into optimally sized blocks termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be

addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

support node

The node that is used to run serial commands, such as shells, editors, and other user commands (`ls`, for example). See also *application node*; *OS node*; *node*; *support node*.

synchronization

The act of bringing two or more processes to known points in their execution at the same clock time. See also *barrier synchronization*.

template

In C and C++ programming, a class or function that is a model for a family of related classes or functions. The act of generating a class or function from a template is called template instantiation.

template instantiation

See *template*.

thread

The active entity of execution. A sequence of instructions together with machine context (processor registers) and a stack. On a parallel system, multiple threads can be executing parts of a program at the same time.

UNICOS

The operating system for Cray SV1 series systems.

UNICOS/mk

The operating system for Cray T3E systems.

UNICOS/mp

The operating system for Cray X1 systems.

unrolling

A single-processing-element optimization technique in which the statements within a loop are copied. For example, if a loop has two statements, unrolling might copy those statements four times, resulting in eight statements. The loop control variable would be incremented for each copy, and the stride through the array would also be increased by the number of copies. This technique is often performed directly by the compiler, and the number of copies is usually between two and four.

vector

An array, or a subset of an array, on which a computer operates. When arithmetic, logical, or memory operations are applied to vectors, it is referred to as *vector processing*.

vector length

The number of elements in a vector.

vector processing

A technique whereby operations are performed simultaneously on the elements of an array instead of iteratively. Compare to *scalar processing*.

vectorizable loop

A loop that contains only vectorizable expressions (that is, expressions for which the compiler can produce vector code).

vectorization

A form of processing that uses one instruction for the simultaneous performance of iterative operations on elements in sets of ordered data.

vectorized loop

A source code loop that is processed with hardware vector registers.

Index

D