Cray Standard C/C++ Reference Manual

S–2179–36

New Features

Cray Standard C/C++ Reference Manual S–2179–36

The Cray Standard C / C++ Reference Manual supports the Cray Standard C / C++ compiler 3.6 release. Documentation supporting the following features were added to this manual:

Standard C++	The Cray Standard C++ compiler now supports the C++98 standard (ISO / IEC FDIS 14882:1998). Refer to Chapter 4, page 103 for more information.
Cray C++ STL compiler options	Added two new options that allow existing code that define templates using the Cray C++ STL to compile successfully with the `-h conform` option. The options allow you to compile existing code without recoding it to use the current Cray standard C++ STL. The new options are: `-h [no]parse_templates` and `-h [no]dep_name`. Refer to Section 2.4.4, page 10 and Section 2.4.5, page 10 for more information.
C++ programming environment compatibility option	The `CRAYOLDCPPLIB` environment variable allows C++ code developed under programming environment 3.4 and 3.5 to compile successfully without modification. Refer to Section 2.24, page 43 for more information.
Template instantiation	Two compiler options were added to aid template instantiation: `-h one_instantiation_per_object` and `-h instantiation_dir`. Refer to Section 2.5.2, page 12 for more information.
Standard C Conformance	The Cray Standard C compiler now supports the C99 standard (ISO / IEC 9899:1999) through the `-h [no]c99` command line option. This option enables language features new to the C99 standard and Cray Standard 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. Hexadecimal floating point constants are an example of this behavior.
Vectorization Optimization	(Cray SV1 series only) The new `-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.
Display current optimization selections	Added the `-h display_opt` option which displays the current optimization settings for the current compilation.
`bte_move` intrinsic (Cray SV1 Series systems only)	Added a reference to the new `bte_move` intrinsic command that allows a program to move data within main memory or the SSD, from main memory to SSD, and from SSD to main memory. Refer to the `bte_move`(3i) man page for more information.
Miscellaneous changes	Other miscellaneous technical changes were made to this document.

Many Cray C++ extensions were adopted by the C++ standard, and therefore corresponding sections have been removed from this manual. Information about these features can be found in various books that document the C++ standard.

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.
36	June 2002 This revision supports the Cray C 6.6 and Cray C++ 3.6 releases running on UNICOS and UNICOS/mk operating systems.

Contents

Page

Page

Page

Page

Page

Preface

This publication describes the Standard C and C++ languages as implemented by the Cray Standard C 6.6 release and the Cray Standard C++ 3.6 release compilers. These compilers are supported on the following systems:

- Cray SV1 series systems running UNICOS 10.0.1.0 or later. The use of the `bte_move` intrinsic requires UNICOS 10.0.1.1.
- Cray T3E systems running UNICOS/mk 2.0.5 or later.

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

This publication contains the following chapters:

- Chapter 1, page 1, which contains introductory information.
- Chapter 2, page 3, which contains information on the `CC`(1), `cc`(1), `c89`(1), and `cpp`(1) commands.
- Chapter 3, page 45, which contains information on the `#pragma` directives supported by the Cray Standard C/C++ compilers.
- Chapter 4, page 103, which contains information about supported and unsupported standard C++ features and about the Dinkum C++ library.
- Chapter 5, page 105, which contains information on the extensions to the C and C++ languages.
- Chapter 6, page 109, which contains information on Cray Standard C++ template instantiation.
- Chapter 7, page 117, which contains information on predefined macros.
- Chapter 8, page 121, which contains information on debugging Cray Standard C/C++ code.
- Chapter 9, page 123, which contains information on interlanguage communication.
- Chapter 10, page 141, which contains information on implementation-defined behavior.
- Appendix A, page 153, which contains information on the libraries and the loaders.

- Appendix B, page 157, which contains information on the Cray Standard C/C++ dialects.
- Appendix C, page 173, which contains information on how to extract information on compiler messages and how to manipulate the message system.
- Appendix D, page 179, which contains information on intrinsic functions.

Related Publications

The following documents contain additional information that may be helpful:

- *UNICOS User Commands Reference Manual*
- *UNICOS/mk User Commands Reference Manual*
- *UNICOS System Libraries Reference Manual*
- *UNICOS/mk System Libraries Reference Manual*
- *Introducing CrayLibs*
- *Intrinsic Procedures Reference Manual*
- *Scientific Library Reference Manual*
- *Scientific Libraries User's Guide*
- *Application Programmer's Library Reference Manual*
- *Introducing the Cray TotalView Debugger*
- *Optimizing Application Code on UNICOS Systems*
- *CRAY T3E C and C++ Optimization Guide*
- *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

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

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.

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 Standard C++ Programming Environment contains both the Cray Standard C and C++ compilers. The Cray Standard C compiler conforms to the International Standards Organization (ISO) standard ISO / IEC 9899:1999 (C99). The Cray Standard C++ compiler conforms to the ISO / IEC 14882:1998 standard, with some exceptions. The exceptions are noted in Appendix B, page 157.

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

1.1 Setting up the Programming Environment

The installation and configuration of the programming environment uses a utility called `modules`, which is provided and installed as part of the release package. The `/opt/ct1/doc/README` file is distributed in the release package. It contains information on initializing the `module` command and initializing the environment.

The default programming environment is available to you after you have entered the following command:

```
module load modules PrgEnv
```

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

1.2 General Compiler Description

Both the Cray Standard C / C++ compilers are contained within the same programming environment. If you are compiling code written in C, use the `cc`(1) or `c89`(1) commands to compile source files. If you are compiling code written in C++, use the `CC`(1) command.

1.2.1 Cray Standard C++ Compiler

The Cray Standard C++ compiler consists of a preprocessor, a language parser, a prelinker, an optimizer, and a code generator. The Cray Standard 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 4, and on the CC(1) man page. Command line examples are shown in Section 2.23, page 42.

1.2.2 Cray Standard C Compiler

The Cray Standard C compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. The Cray Standard C compiler is invoked by a command called cc(1) or c89(1) in this manual, but it may be renamed at individual sites. The cc(1) is discussed in Section 2.2, page 4 the and c89(1) command is described in Section 2.2.1, page 5, and in the CC(1) man page. Command line examples are shown in Section 2.23, page 42.

Compiler Commands [2]

This section describes the compiler commands and the environment variables necessary to execute the Cray Standard C and C++ compilers. The commands for these compilers are as follows:

- `CC`, which invokes the Cray Standard C++ compiler.
- `cc`, which invokes the Cray Standard C compiler.
- `c89`, which invokes the Cray Standard 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`, and `c89` commands invoke the preprocessor automatically. The `cpp` command provides a way for you to invoke only the preprocessor component of the Cray Standard 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`, or `c89` commands, with other files or libraries included as necessary.

By default, the `CC`, `cc`, and `c89` 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 Standard 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] [-F] [-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..."]] [-Xnpes] [-Yphase,dirname] [-#] [-##] [-###]
   file [file]...
```

See Section 2.3, page 6 for an explanation of the command line options.

2.2 cc Command

The cc command invokes the Cray Standard C compiler. The cc command accepts 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 command format is as follows:

```
cc [-c] [-C] [-d string] [-D macro[=def]] [-E] [-F] [-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...
```

See Section 2.3, page 6 for an explanation of the command line options.

2.2.1 c89 Command

The c89 command invokes the Cray Standard 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...
```

See Section 2.3, page 6 for an explanation of the command line options.

2.2.2 cpp Command

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

The cpp command format is as follows:

```
cpp [-C] [-D macro[=def]] [-E] [-I incldir] [-M] [-N] [-nostdinc] [-P]
   [-U macro] [-V] [-Yp,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 2.3, page 6 for an explanation of the command line options.

2.3 Command Line Options

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

- Language options:
  - The standard conformance options are described in the following sections:

Section	Option
Section 2.4.1, page 8	`-h [no]c99`
Section 2.4.2, page 9	`-h [no]conform` and `-h [no]stdc`
Section 2.4.3, page 10	`-h cfront`
Section 2.4.4, page 10	`-h [no]parse_templates`
Section 2.4.5, page 10	`-h [no]dep_name`
Section 2.4.6, page 10	`-h [no]exceptions`
Section 2.4.7, page 11	`-h [no]anachronisms`
Section 2.4.8, page 11	`-h new_for_init`
Section 2.4.9, page 11	`-h [no]tolerant`

  - The template options are described in the following sections:

Section	Option
Section 2.5.1, page 12	`-h [no]autoinstantiate`
Section 2.5.2, page 12	`-h one_instantiation_per_object`
Section 2.5.3, page 12	`-h instantiation_dir =` *dirname*
Section 2.5.4, page 12	`-h instantiate=`*mode*
Section 2.5.5, page 13	`-h [no]implicitinclude`
Section 2.5.6, page 13	`-h remove_instantiation_flags`
Section 2.5.7, page 13	`-h prelink_local_copy`
Section 2.5.8, page 13	`-h prelink_copy_if_nonlocal`

  - The virtual function options, `-h forcevtbl` and `-h suppressvtbl`, are described in Section 2.6.1, page 13.

  - General language options are described in the following subsections:

Section	Options
Section 2.7.1, page 14	`-h keep=`*file*
Section 2.7.2, page 14	`-h restrict=`*args*
Section 2.7.3, page 15	`-h [no]calchars`
Section 2.7.4, page 15	`-h [no]signedshifts`

- Optimization options:
  - General optimization options, described in Section 2.8, page 16.
  - Multi-Streaming Processor (MSP) options, described in Section 2.9, page 17.
  - Vectorization options, described in Section 2.10, page 18.
  - Tasking options, described in Section 2.11, page 20.
  - Inlining options, described in Section 2.12, page 22.
  - Scalar optimization options, described in Section 2.13, page 23.
  - UNICOS/mk options, described in Section 2.14, page 25.
- Math options, described in Section 2.15, page 26.
- Analysis tools options, described in Section 2.16, page 28.
- Debugging options, described in Section 2.17, page 29.
- Message control options, described in Section 2.18, page 30.
- Compilation phase control options, described in Section 2.19, page 32.
- Preprocessing options, described in Section 2.20, page 35.
- Loader options, described in Section 2.21, page 38.
- Miscellaneous options, described in Section 2.22, page 40.

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

At the beginning of each subsection, information is included about exceptions to the default option, the systems that use the option, the type of floating-point representation used (IEEE and/or Cray floating point), and the commands that

will accept the option. Unless otherwise noted, the following default information applies to each option:

Default option:	None
Operating System:	UNICOS and UNICOS/mk
Floating-point:	IEEE and Cray floating point

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 nofastmd` command line options can be specified as `-h parse_templates,nofastmd`.

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 nofastmd` overrides `-h fastmd`.

  ```
  CC -h fastmd,nofastmd 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.

2.4 Standard Language Conformance Options

The following subsections describe standard conformance language options.

2.4.1 `-h` [no]c99 (cc)

Default option: `-h noc99`

The `-h c99` option enables language features new to the C99 standard and Cray Standard 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.

The `-h noc99` option allows support for C99 features that were previously implemented as Cray extensions, while disabling C99 defined behavior of these same features.

For example, hexadecimal floating point constants can use the Cray format only when the `-h noc99` option is selected, but can use both the Cray and C99 formats when the `-h c99` option is selected.

The following C99 features are supported:

- `_Bool`
- `complex` data type
- Compound literals
- Designated initializers
- End of line comments
- Flexible array members of `struct`
- Hexadecimal floating point constants
- `inline`
- `long long` data type
- `_Pragma`
- `restrict` keyword
- VLAs

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

Default option: `-h noconform`, `-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 nostdc` 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 Standard 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.

2.4.3 `-h cfront` (`CC`)

The `-h cfront` option causes the Cray Standard 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.

2.4.4 `-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 PE 3.6) to compile successfully with the `-h conform` option. Consequently, this allows you to compile existing code without having to use the Cray Standard 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 Standard C++ STL properly, use the `parse_templates` option.

2.4.5 `-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.

2.4.6 `-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`.

2.4.7 -h [no]anachronisms (CC)

Default option: -h noanachronisms

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

2.4.8 -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.

The following 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
```

2.4.9 -h [no]tolerant (cc)

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.

This option can be specified on the same line with -O3 or any combination of -h scalar3, -h vector3, or -h task3. The combination of -h tolerant with these options causes the compiler to tolerate accessing an object with one

type through a pointer to an entirely different type. For example, a pointer to `int` 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.

2.5 Template Language Options

The following sections describe template language options. See Chapter 6, page 109 for more information on template instantiation.

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

Default option: `-h autoinstantiate`

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

2.5.2 `-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.

2.5.3 `-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`.

2.5.4 `-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`.

2.5.5 `-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.

2.5.6 `-h remove_instantiation_flags` (CC)

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

2.5.7 `-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.

2.5.8 `-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.

2.6 Virtual Functions Options

The following sections describe virtual function options.

2.6.1 `-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.

2.7 General Language Options

The following sections describe general language options.

2.7.1 `-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.

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

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 on UNICOS systems.

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 `this` parameters can be treated as restricted pointers (Cray Standard 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."

2.7.3 -h [no]calchars (CC, cc)

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 and UNICOS/mk 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.

2.7.4 -h [no]signedshifts (CC, cc)

Default option: -h signedshifts

(UNICOS/mk systems) 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. The -h nosignedshifts option forces the operator to have the same behavior on UNICOS/mk operating systems as on UNICOS operating systems.

2.8 General Optimization Options

The following sections describe general optimization options.

2.8.1 `-O` *level* (CC, cc, c89)

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 the `-h inline`, `-h scalar`, `-h task`, and `-h vector` options.

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

```
-h inline2,scalar2,task2,vector2
```

Optimization features specified by `-O` are equivalent to the following `-h` options (`task`*n* is ignored on UNICOS/mk systems):

Table 1. `-h` Option Descriptions

`-h` option	Description location
`-h inline`*n*	Section 2.12.1, page 22
`-h scalar`*n*	Section 2.13.2, page 23
`-h task`*n*	Section 2.11.1, page 20
`-h vector`*n*	Section 2.10.3, page 19

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

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. On UNICOS systems, this option also disables the limit on the number of vector

updates in a single loop. On UNICOS/mk systems, this option enables the compiler to aggressively assign registers and schedule instructions.

2.8.3 -h display_opt

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

2.8.4 -h [no]intrinsics (CC, cc)

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 D, page 179.

2.8.5 -h [no]pattern (CC, cc)

Default option: -h pattern

The -h [no]pattern option globally enables or disables pattern matching. Pattern matching is on by default, but takes effect only when -h vector2 or -h scalar2 or greater are specified.

2.8.6 -h [no]overindex (CC, cc)

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.

2.9 Multi-streaming Processor Optimization Options (Cray SV1 Series Systems Only)

The following sections describe the multi-streaming processor (MSP) options. For information on MSP #pragma directives, see Section 3.9, page 85.

Note: The MSP is an optional feature. To determine whether the MSP is enabled on your system, enter the `sysconf` command at your system prompt. The `HARDWARE` output field contains the `NMSP=` field that shows the number of MSPs configured. For more information, see the `sysconf`(1) man page.

2.9.1 `-h streamn` (`CC`, `cc`) (Cray SV1 Series Systems Only)

The `-h streamn` option specifies the level of automatic MSP optimizations to be performed. Generally, vectorized applications that execute on a 1–processor system can expect to execute up to 4 times faster on a processor with multi-streaming enabled.

Argument *n* can be one of the following:

n	Description
`0`	No automatic multi-streaming optimizations are performed.
`1`	Conservative automatic multi-streaming optimizations. Automatic multi-streaming optimization is limited to inner vectorized loops. MSP operations performed generate the same results that would be obtained from scalar optimizations; for example, no floating-point reductions are performed.
`2`	Moderate automatic multi-streaming optimizations. Automatic multi-streaming optimization is performed on loop nests.
`3`	Aggressive automatic multi-streaming optimizations. Automatic multi-streaming optimization is performed as with `stream2`.

2.10 Vector Optimization Options

The following sections describe vector optimization options.

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

Default option: `-h noinfinitevl`

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.

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

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 58.

This option can also be used with "vectorization-like" optimizations on UNICOS/mk systems. See Section 3.7, page 57, for more information.

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.

2.10.3 `-h vector`*n* (CC, cc)

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.

On UNICOS/mk systems, the Cray Standard C/C++ compilers can perform optimizations that are similar to vectorization on loops that contain calls to certain functions. These optimizations are enabled or disabled with this option. See Section 3.7, page 57, for more information on these optimizations.

Argument *n* can be one of the following:

n	Description
`0`	No automatic vectorization.
`1`	Conservative automatic vectorization. On UNICOS systems, automatic vectorization is performed. Search loops and reduction loops are not vectorized.
`2`	Moderate automatic vectorization. On UNICOS systems, automatic vectorization is performed as with `vector1`, and vectorization of search loops and reduction loops is added.
`3`	Aggressive automatic vectorization. Automatic vectorization is performed as with `vector2` and restructuring of loops is done to

improve vectorization. Also, the aliasing assumptions specified in the standard are used (for example, it is assumed that no aliasing will occur between two pointers to different structure types). On UNICOS/mk systems, "vectorization-like" optimizations are performed. See Section 3.7, page 57, for more information.

Vectorization directives are described in Section 3.7, page 57.

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

Default option: `-h vsearch`

(UNICOS systems) 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 60. This option is affected by the `-h vector`*n* option (see Section 2.10.3, page 19).

2.11 Tasking Optimization Options

The following sections describe task optimization options.

2.11.1 `-h task`*n* (`CC`, `cc`)

Default option: `-h task0`

The `-h task`*n* option specifies the level of automatic tasking (Autotasking) to be performed. Tasking allows segments of code to execute in parallel on multiple processors. This option has no effect on tasking directives. Tasking and tasking directives are described in Section 3.8, page 62.

Note: The `-h task`*n* option is accepted and ignored on UNICOS/mk systems.

Argument *n* can be one of the following:

n	Description
`0`	No Autotasking.
`1`	Conservative Autotasking. Same as `task0` in this release.
`2`	Moderate Autotasking. Same as `task0` in this release.
`3`	Aggressive Autotasking. This includes loop restructuring for improved tasking performance. Aliasing assumptions specified in the standard are also used; for example, it is assumed that no

aliasing will occur between two pointers to different structure types.

2.11.2 -h taskprivate (cc)

This option gives task private status to all statically-allocated objects in the program.

Unlike -h taskcommon, initialized objects can be made private to each task with the -h taskprivate option. They are initialized at startup for each task prior to the execution of the main entry point.

For information on tasking and tasking directives, see Section 3.8, page 62.

2.11.3 -h taskcommon, -h common (CC, cc)

Default option: -h common

The -h taskcommon option gives task common status to all statically-allocated objects in the program. The -h common option gives common (as opposed to taskcommon) status to all global objects in the program. Tasking and tasking directives are described in Section 3.8, page 62.

Objects that are initialized will not be marked as task common. The -h taskprivate option can be used to make these objects private to each task and be initialized at startup for each task prior to the execution of the main entry point.

2.11.4 -h [no]taskinner (CC, cc)

Default option: -h notaskinner

(UNICOS systems) Autotasking attempts to maximize the amount of parallel work in a taskable loop by interchanging the loop outwards. Sometimes this fails and a taskable loop remains innermost. By default, such a remaining innermost and taskable loop will not task if, at compile time, sufficient parallel work cannot be found. The -h taskinner option enables tasking of the innermost loop with a run-time threshold check to ensure that there is sufficient parallel work in the loop. Aggressive Autotasking (-h task3) must also be specified for this option to take effect.

2.11.5 -h [no]threshold (CC, cc)

Default option: -h threshold

(UNICOS systems) The `-h [no]threshold` option enables or disables generation of run-time threshold testing for autotasked loops. Aggressive Autotasking (`-h task3`) must also be specified for this option to take effect.

2.12 Inlining Optimization Options

The following sections describe inlining options.

2.12.1 `-h inline`*n* (`CC`, `cc`)

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 by this option.

Following are the values for the *n* argument:

n	Description
`0`	No inlining is performed.
`1`	Conservative inlining. Inlining is performed on functions explicitly marked by one of the following: • The `inline` keyword (Cray Standard C++). • A `#pragma _CRI inline` directive (Cray Standard C and C++). Inlining is also performed on functions defined inside a class definition (Cray Standard C++).
`2`	Moderate automatic inlining. Includes level `1` plus some automatic inlining.
`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.

2.13 Scalar Optimization Options

The following sections describe scalar optimization options.

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

Default option: `-h interchange`

The `-h interchange` option instructs 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.

2.13.2 `-h scalar`*n* (`CC`, `cc`)

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 86).

Argument *n* can be one of the following:

n	Description
`0`	No automatic scalar optimization. The `-h nobl`, `-h nofastfpdivide`, `-h nofastmd`, `-h nofastmodulus`, `-h matherror=errno`, and `-h zeroinc` options are implied by `-h scalar0`.
`1`	Conservative automatic scalar optimization. This level implies `-h fastmd`, `-h fastfpdivide`, `-h matherror=abort`, and `-h nozeroinc` and causes automatic loop alignment to be performed.
`2`	Moderate automatic scalar optimization. The scalar optimizations specified by `scalar1` are performed.
`3`	Aggressive automatic scalar optimization. The scalar optimizations specified by `scalar2` are performed and `-h fastmodulus` and `-h bl` are implied.

2.13.3 -h [no]align (CC, cc)

Default option: -h noalign

(UNICOS systems) The -h align option specifies that all functions defined in the file are to be automatically aligned on instruction buffer boundaries. This alignment can significantly improve performance for small, frequently called functions. With -h noalign, automatic function alignment is not done.

To control alignment of functions individually, use the align directive. For more information on the align directive and function alignment, see Section 3.10.1, page 87.

2.13.4 -h [no]bl (CC, cc)

Default option: -h nobl

The -h bl option specifies a faster, but potentially unsafe, form of bottom loading. -h nobl dictates that this technique is not used. This option is affected by the scalar optimization level (see Section 2.13.2, page 23).

2.13.5 -h [no]reduction (CC, cc)

Default option: -h reduction

On UNICOS systems, the -h reduction option instructs the compiler to enable vectorization of all reduction loops. On UNICOS/mk systems, the -h reduction option instructs the compiler to rewrite some multiplication operations to be a series of addition operations. With -h noreduction, these optimizations are not done. This option is affected by the -h scalar*n* option (see Section 2.13.2, page 23). Reduction loops and the noreduction directive are discussed in Section 3.10.6, page 93.

2.13.6 -h [no]zeroinc (CC, cc)

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 2.13.2, page 23).

2.14 UNICOS/mk Specific Optimization Options

The following sections describe UNICOS/mk specific compiler options.

2.14.1 `-h pipeline`*n* (CC, cc commands)

Default option: `-h pipeline0`

(UNICOS/mk systems) The `-h pipeline`*n* option specifies two levels of software pipelining; on and off. The following are the various software pipelining levels and their types of operations:

n	Description
`0`	No pipelining. Default.
`1`	Conservative pipelining. Only safe operator reassociations are performed. Numeric results obtained at this level do not differ from results obtained at level 0.
`2`	Moderate pipelining. Same as `pipeline1`.
`3`	Aggressive pipelining. Same as `pipeline1`.

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

Default option: `-h nounroll`

(UNICOS/mk systems) The `-h unroll` option instructs the compiler to attempt to unroll all loops generated by the compiler. This technique is intended to increase single processor performance at the cost of increasing compile time and executable size. To control unrolling of loops individually, use the `unroll` directive. For more information on this directive and loop unrolling, see Section 3.10.10, page 98.

Note: On UNICOS systems, `-h unroll` is enabled at all times.

2.14.3 `-h` [`no`]`jump` (CC, cc Commands)

Default option: `-h jump`

(UNICOS/mk systems) The `-h jump` option generates jumps instead of branches to external functions. Branches provide slightly better performance. However, branches are limited in the distance to which they can transfer control; jumps have no such limitation. For large programs you may need to use `-h jump` with files that generate calls to functions loaded at too great a distance.

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

Default option: `-h nosplit`

(UNICOS/mk systems) The `-h split` option instructs the compiler to attempt to split all loops generated by the compiler into sets of smaller loops. This technique is intended to increase single processor performance on UNICOS/mk systems by reducing thrashing of Cray T3E system hardware stream buffers. To control splitting of loops individually, use the `split` and `nosplit` directives. For more information on these directives and on loop splitting, see Section 3.10.7, page 94.

2.15 Math Options

The following sections describe compiler options with regard to math functions.

2.15.1 `-h matherror=`*method* (CC, cc)

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. On systems with IEEE floating-point hardware, 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.

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

Default option: `-h fastmd`

(UNICOS systems) The `-h fastmd` option generates shorter code sequences for `int` variables when doing multiply, divide, or comparison operations, or when converting to and from floating-point operations, but allows for only 46 bits of significance. With `-h nofastmd`, this action is disabled. This option is affected by the scalar optimization level (see Section 2.13.2, page 23).

2.15.3 `-h` [`no`]`fastmodulus` (CC, cc)

Default option: `-h nofastmodulus`

(UNICOS systems) The `-h fastmodulus` option generates shorter code sequences for `int` variables used with the modulus operator (%), but allows only 46 significant bits. This option is affected by the scalar optimization level, (see Section 2.13.2, page 23).

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

Default option: `-h noieeeconform`

Floating-point: IEEE only

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.

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.

The `-h ieeeconform` option overrides the `-h fastfpdivide` option.

2.15.5 `-h` [`no`]`fastfpdivide` (CC, cc)

Default option: `-h fastfpdivide`

Floating-point: IEEE only

The `-h fastfpdivide` option decomposes a floating-point divide into a multiply-by-reciprocal in situations where a performance gain can be

realized. For example, this option is useful in loops that contain divides with a loop-invariant divisor or sequences of divides with the same divisor. If the option is enabled, you could see slight numerical differences from compiles for which the option is not enabled. You could also see numerical differences between instances of the same computation for the same compile, depending on the context of the computation.

This option is affected by the scalar optimization level (see Section 2.8.1, page 16).

The `-h ieeeconform` option overrides the `-h fastfpdivide` option.

2.15.6 `-h` [`no`]`rounddiv` (`CC`, `cc`)

Default option: `-h norounddiv`

Floating-point: Cray floating-point

The `-h` [`no`]`rounddiv` option enables or disables strong rounding of all floating-point divide operations to allow the compiler to produce more symmetric results. Strong rounding ensures that all floating-point divide operations whose exact result is an integer will have an absolute value slightly greater than the integer value. This ensures that if these floating-point values are converted back into integer values they will represent the expected results. For example, results such as 2.99999..., when converted, will be treated as 3.

2.15.7 `-h` [`no`]`trunc`[`=`*n*] (`CC`, `cc`)

Default option: `-h notrunc`

Floating-point: Cray floating-point

The `-h trunc`[`=`*n*] option specifies truncation of the last *n* bits (range of *n*: $0 \leq n \leq 47$) of single-precision floating-point arithmetic. This option has no effect on double precision operations, function return values, and compile-time constants.

This option is useful for identifying numerically unstable algorithms. `-h trunc` with no argument is equivalent to `-h trunc=0`; that is, the assembler truncation instructions for floating-point arithmetic are generated, and the results from floating operations are truncated by `0` bits. The `-h notrunc` option generates assembler rounding instructions for floating arithmetic.

2.16 Analysis Tool Options

The following sections describe compiler options that support analysis tools.

2.16.1 `-F` (CC, cc)

(UNICOS systems) The `-F` option enables the generation of additional run-time code that is needed by Perftrace or Flowtrace. Perftrace and Flowtrace are program analysis tools that display the call tree of a program and the amount of time spent in each function. To use `perfview`, you must also specify the `-l perf` option. See the `perftrace` or `flowtrace`(7) man page for more information.

2.16.2 `-h listing` (CC, cc)

Default option: Listing is off

(UNICOS systems) The `-h listing` option generates a pseudo assembly language listing. The listing file name is the same as the source file name, with the suffix replaced by a `.L`.

2.17 Debugging Options

The following sections describe compiler options used for debugging.

2.17.1 `-G` *level* (CC, cc) and `-g` (CC, cc, c89)

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. For the `-G` option, *level* indicates the following:

Table 2. `-G` *level* Definitions

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

More 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.

2.17.2 `-h` [`no`]`bounds` (`cc`)

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.

2.17.3 `-h indef`, `-h zero` (`CC`, `cc`)

The `-h indef` option causes stack-allocated memory to be initialized to undefined values. These values cause run-time errors to occur when an uninitialized stack variable is used, such as in a floating-point operation or in an array subscript. The `-h zero` option causes stack-allocated memory to be initialized to all zeros. These options are especially useful for debugging tasked codes.

2.18 Compiler Message Options

The following sections describe compiler options that affect messages.

2.18.1 `-h msglevel_`*n* (`CC`, `cc`)

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).

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

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 the following:

```
-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.

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

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

args	Description
`i`	Generates inlining optimization messages
`m`	Generates MSP optimization messages (Cray SV1 series systems only)
`s`	Generates scalar optimization messages
`t`	Generates tasking 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
```

2.18.4 `-h` [`no`]`abort` (`CC`, `cc`)

Default option: `-h noabort`

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

2.18.5 `-h errorlimit`[`=`*n*] (`CC`, `cc`)

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`.

2.19 Compilation Phase Options

The following sections describe compiler options that affect compilation phases.

2.19.1 `-E` (`CC`, `cc`, `c89`, `cpp`)

If the `-E` option is specified on the `CC`, `cc`, or `c89` command lines, 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.

2.19.2 `-P` (`CC`, `cc`, `cpp`)

When the `-P` option is specified on the `CC` or `cc` command line, 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.

2.19.3 -h feonly (CC, cc)

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

2.19.4 -S (CC, cc)

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.

2.19.5 -c (CC, cc, c89)

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.

2.19.6 -#, -##, and -### (CC, cc, 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.

2.19.7 -W*phase*["*opt*...] (CC, cc)

The -W *phase* option passes arguments directly to a phase of the compiling system. The -W option appears with argument *phase* to indicate system phases as follows:

Table 3. `-W` *phase* Definitions

phase	System phase	Command
`p`	Preprocessor	
`0`	Compiler	
`a`	Assembler	`as` on Cray PVP systems, `cam` on Cray MPP systems
`l`	Loader	System-specific; `ld` or `cld`.

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.

2.19.8 `-Y`*phase*`,`*dirname* (`CC`, `cc`, `c89`, `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 following values:

Table 4. `-Y` *phase* Definitions

phase	System phase	Command
`p`	Preprocessor	
`0`	Compiler	
`a`	Assembler	`as` on Cray PVP systems, `cam` on Cray MPP systems
`l`	Loader	System-specific; `ld` or `cld`

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.

2.20 Preprocessing Options

The following sections describe compiler options that affect preprocessing.

2.20.1 `-C` (CC, cc, 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` and `cc` commands.

2.20.2 `-D` *macro*[`=`*def*] (CC, cc, c89, 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 7, page 117. Any predefined macro except those required by the standard (see Section 7.1, page 117) 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.

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

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 5, page 36 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 5. `-h pragma` Directive Processing

name	Group	Directives affected
`all`	All	All directives
`allinline`	Inlining	`inline`, `noinline`
`allscalar`	Scalar optimization	`align`, `cache_align`, `cache_bypass`, `concurrent`, `nointerchange`, `noreduction`, `split`, `suppress`, `unroll`
`alltask`	Tasking	`case`, `cncall`, `endcase`, `guard`, `endguard`, `taskloop`, `endloop`, `parallel`, `endparallel`, `prefertask`, `taskcommon`, `common`, `taskprivate`, `taskshared`
`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 example, `parallel` and `endparallel`). 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.

2.20.4 `-I` *incldir* (`CC`, `cc`, `c89`, `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 ().

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 backslash (`/`), 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`.

2.20.5 `-M` (CC, cc, 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`.

2.20.6 `-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.

2.20.7 `-nostdinc` (`CC`, `cc`, `c89`, `cpp`)

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

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

The `-U` option removes any initial definition of *macro*. Any predefined macro except those required by the standard (see Section 7.1, page 117) 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 7, page 117. Macros defined in the system headers are not predefined macros and are not affected by the `-U` option.

2.21 Loader Options

The following sections describe compiler options that affect loader tasks.

2.21.1 `-d` *string* (`CC`, `cc`)

The `-d` *string* option specifies a character string comprised of directive names separated by semicolons that is sent to the loader to be inserted into the loader directives file and processed as though the `-D` *dirstring* option had been specified on the loader command. This allows you to manipulate the loader while using the compiler command.

2.21.2 `-l` *libfile* (`CC`, `cc`, `c89`)

The `-l` *libfile* option identifies library files to be loaded. If *libfile* begins with a period (`.`) or slash (`/`), it is assumed to be a path name and is used without modification. An initial `.` (or `..`) is interpreted as the current working directory (or its parent directory). It is not relative to the input file's directory if that differs from the current working directory.

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

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

`libu.a`

`libm.a`

`libc.a`

`libsma.a` (UNICOS/mk systems only)

`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.

2.21.3 `-L` *libdir* (`CC`, `cc`, `c89`)

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 default directories in the following order:

1. Site-specific and compiler release-specific library directories
2. `/lib`
3. `/usr/lib`

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.

2.21.4 `-o` *outfile* (`CC`, `cc`, `c89`)

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.

2.21.5 `-s` (`CC`, `cc`, `c89`)

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.

2.22 Miscellaneous Options

The following sections describe compiler options that affect general tasks.

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

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.

2.22.2 `-V` (`CC`, `cc`, `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 3.5.0.3 (u144c34129p42138g41109a18)
01/22/02 12:37:25
```

If a file is specified, information about the compilation is displayed in addition to the version information. The additional information includes the compiler generation date, the compilation execution time, the maximum memory used by the compiler (in decimal words), and the resulting number of words used for code and data.

2.22.3 `-X` *npes* (`CC`, `cc`)

(UNICOS/mk systems) The `-X` *npes* option specifies how many processing elements (PEs) are to be used on Cray T3E systems. The *npes* argument specifies the number of PEs and has no default value, it must be explicitly set. For the *npes* argument, specify either an integer from 1 through 2048 or `m`. A value of `m` directs the compiler to generate a malleable `a.out` file. Specifying `-X m` allows you to change the number of PEs used each time the executable `a.out` file is run. If you specify `-X m`, use the `mpprun` command and its `-n` option to specify the number of PEs you want to use. For more information, see the `mpprun`(1) man page. If you do not use `mpprun` on the `a.out` file that is generated when `-X m` is specified, the operating system executes the file on a single processor just as if you had invoked `mpprun` with one processor.

The option is passed from the command to both the compiler and the loader. If the compiler recognizes the option, it becomes a compile-time value and cannot be changed at load time. If the loader recognizes the option, it is a load-time value and cannot be changed at `mppexec` time. For example:

```
cc -X8 -c file.c
cc -X8 file.c
```

In these cases, the value 8 is set at compile time only. In the first line, the loader is not called (specified by the `-c` option) and the option is passed only to the compiler. In the second line, the option is passed to both the compiler and the loader, but since it is first recognized by the compiler it is a compile-time constant, not a load-time constant.

In the following case, the value 8 is used at load time only. The compiler is not called because no source file is specified and the option is passed only to the loader.

```
cc -X8 file.o
```

The `-W` option can also be used to specify which phase of compilation gets the `-X` option. For example:

```
cc -W0,-X8 file.c
cc -Wl,-X8 file.c
```

In the first line, the `-X` option is passed only to the compiler, and the number of PEs is set to 8 at compile time. In the second line, the `-X` option is passed only to the loader and the number of PEs is set to 8 at load time.

If the number of PEs is specified at both compile and load time, the compile-time constant overrides the load-time constant. If the two values disagree, the loader issues a message.

2.23 Command Line Examples

The following examples illustrate a variety of `CC` and `cc` command lines.

- The following example compiles `myprog.C` on UNICOS/mk systems, fixing the number of processing elements (PEs) to 8 and instantiating all template entities that are declared or referenced in the compilation unit.

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

- The following 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
  ```

- The following 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
  ```

- The following 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
  ```

- The following 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
  ```

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

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

- The following 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
  ```

- The following example compiles `myfile.c` and passes an option to the loader (`-Dalign=modules`) that causes blocks of code to be aligned.

  ```
  cc -Wl,"-Dalign=modules" myfile.c
  ```

- The following 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
  ```

2.24 Environment Variables

The environment variables listed below are used during compilation.

Variable	Description
`CRAYOLDCPPLIB`	If set to a nonzero value, this variable instructs the compiler to resolve references using C++ libraries that are compatible with code developed under programming environment 3.4 or 3.5 and then use the C++ library of programming environment 3.6. Some C++ codes compiled with previous compilers may be binary incompatible with the 3.6 C++ compiler. The `CRAYOLDCPPLIB` environment variable suppresses these incompatibilities by using the `libCx.a` library, which is the `libC.a` library of programming environment 3.5, in conjunction with the `libC.a` library of programming environment 3.6. If you do not use the variable, your older code may require modification to compile successfully. Refer to *Programming Environments Release Overview* for more information about modifying your code.

	Note: Setting the CRAYOLDCPPLIB environment variable disables exception handling. Refer to the -h [no]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	Specifies 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. See the target(1) and sh(1) man pages for more information.

#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 Standard C++ only)
- Vectorization
- Scalar
- Tasking
- Inlining

Macro expansion occurs on the directive line after the directive name. That is, macro expansion is applied only to *arguments*. For example, if NUM_CHUNKS is a macro defined with a value 8, the original code is as follows:

```
#pragma _CRI taskloop numchunks(NUM_CHUNKS)
```

The expanded code is equivalent to the following:

```
#pragma _CRI taskloop numchunks(8)
```

At the beginning of each section that describes a directive, information is included about the compilers and systems 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 Standard C and Cray Standard C++
Operating System:	UNICOS and UNICOS/mk
Scope:	Local and global

3.1 Protecting Directives

To ensure that your directives are interpreted only by the Cray Standard C/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 Standard C/C++ compilers diagnose directives that are not recognized only if the `_CRI` specification is used.

3.2 Directives in Cray Standard 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 align`, `#pragma soft`, `#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 6.4, page 114.

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
#pragma _CRI parallel private(i, j, k) \
                      shared(a, b, c) \
                      valude(x, y, z)
```

The following is the same example using the alternative form:

```
_Pragma("_CRI ivdep");
_Pragma("_CRI parallel private(i, j, k) \
                       shared(a, b, c)  \
                       value(x, y, z)");
```

In the following example, the loop automatically vectorizes on UNICOS systems 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;
}
```

It is possible to use this feature in a portable, conformant C program. To do so, you must select a name from the user name space (that is, any valid identifier that does not begin with an underscore, for instance Pragma) and use that name instead of _Pragma. Then insert preprocessor directives at the beginning of the compilation unit, as in the following example:

```
#ifdef _CRAY
#define Pragma(S) _Pragma(S)
#else
#define Pragma(S)
#endif
```

When this definition is used on a UNICOS or UNICOS/mk system, the directives are interpreted as intended; when used on another vendor's system, the directives are removed.

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

```
#define NUM_CHUNKS 8
_Pragma("_CRI parallel numchunks(NUM_CHUNKS)")
```

The following example shows the expanded code, which is the only situation in which macros are expanded inside of string literals:

```
#pragma _CRI taskloop numchunks(8)
```

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 besu Directive

The `besu` directive indicates that *n* BESUs (barrier/eureka synchronization units) should be allocated for use in the compilation unit. The format of this directive is as follows:

```
#pragma _CRI besu n
```

The sum of the BESU counts specified with directives in a program is recorded at link time and placed in the `a.out` header. The operating system allocates the specified number to the application team at program startup. As a special case, the operating system does not allocate a BESU if the BESU count in the `a.out` header is 1 and the program used one PE.

For more information on accessing BESUs, see *Barrier and Eureka Synchronization (Cray T3E Systems)*, publication HMM-141–0. (A nondisclosure agreement must be signed with Cray Inc. before you can obtain this document.) For a convenient source of BESU state codes, see header file `mpp/mpphw_t3e.h`.

This directive is not required when accessing BESUs through the following barrier and eureka event routines: `barrier`(3), `pvm_barrier`(3), `shmem_barrier_all`(3), `set_event`(3), `wait_event`(3), `test_event`(3), `clear_event`(3). However, this directive is required when programming BESUs directly through the techniques described in the *Barrier and Eureka Synchronization (Cray T3E Systems)*, publication HMM-141–0.

3.5.2 [no]bounds Directive (Cray Standard 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];
   .
   .
   .
}
```

3.5.3 `duplicate` Directive (Cray Standard 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
```

3.5.4 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
```

3.5.5 [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 uses_eregs Directive (UNICOS/mk Systems)

Scope: Local

The `uses_eregs` directive reserves all E registers for your use in the function in which the directive appears. It prevents the compiler from generating code that would change E register values. The format of this directive is as follows:

```
#pragma _CRI uses_eregs
```

The `uses_eregs` directive applies only to the function in which it appears. Your code must comply with E register conventions as described in the *Cray Assembler for MPP (CAM) Reference Manual*.

Note: Use of this directive prevents the `cache_bypass` directive from being processed because when `uses_eregs` is in effect, no E registers are available to the compiler.

3.5.7 `soft` Directive

Scope: Global

The `soft` directive specifies external identifiers with references that are to be considered soft. *Soft external* references can be to a function or to a data object. Soft externals do not increase your program's total memory requirements.

The format of this directive is as follows:

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

var List of one or more soft externals, separated by commas (,) and optionally enclosed in parentheses.

Declaring a soft external directs the linker to link the object or function only if it is already linked (that is, if it has been referenced without soft externals in another code file); otherwise, it is left as an unsatisfied external. If you declare a soft external, you also direct the linker to inhibit an unsatisfied external message if it is left unsatisfied.

Note: The loader treats soft externals as unsatisfied externals, so they remain silently unsatisfied if all references are under the influence of a soft directive. Thus, it is your responsibility to ensure that run-time references to soft external names do not occur unless the loader (using some "hard" reference elsewhere) has actually loaded the entry point in question. You can determine whether a soft external has been loaded by calling the `loaded`(3) library function.

The `soft` directive must appear at global scope. Soft externals must have the following attributes:

- They must be declared, but not defined or initialized, in the source file.
- They cannot be declared with a `static` storage class.
- They cannot be declared as task common.

The following example illustrates these restrictions:

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

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

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

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

3.5.8 `vfunction` Directive (UNICOS Systems)

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 either in Cray Assembly Language (CAL) or in Fortran using the Fortran `vfunction` compiler directive. 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]);
    }
}
```

3.5.9 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 Standard 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 6.4, page 114:

- #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 6, page 109 for more information on template instantiation.

3.7 Vectorization Directives

Because vector operations cannot be expressed directly in Cray Standard C/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 Application Code on UNICOS Systems.*

On UNICOS/mk systems, the compiler can perform "vectorization-like" optimizations on certain loops. Vector versions of the following functions are used when the function appears in a vectorizable loop on UNICOS/mk systems: `alog`(3m), `exp`(3m), `sqrt`(3m), `ranf`(3m), `sin`(3m), `cos`(3c), `coss`(3m), `pow`(3c), and `_popcnt`(3i). This “vectorization” is performed using the following process:

1. The loop is stripmined. Stripmining is 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.
2. If necessary, a strip of operands is stored in a temporary array. The vector version of the function is called, which stores the strip of results in a temporary array.
3. The remainder of the loop is computed using the results from step 2.

The subsections that follow describe the compiler directives used to control vectorization on UNICOS systems and "vectorization-like" optimizations on UNICOS/mk systems.

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 SV1 series and Cray T3E systems, the compiler assumes that the safe vector length is the maximum vector length supported by that machine. On Cray SV1 series systems, the `-h [no]infinitevl` compiler option can be used 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 (UNICOS Systems)

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 (UNICOS Systems)

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. See also the prefertask directive in Section 3.8.12, page 78.

3.7.6 shortloop and shortloop128 Directives

Scope: Local

The shortloop (all systems) and shortloop128 (UNICOS systems only) 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 (504 for character arrays) and a minimum iteration count of 1. The shortloop128 compiler directive identifies vector loops that execute with a maximum iteration count of 128 (1016 for character arrays) 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 vector length of UNICOS systems 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 Tasking Directives

The Cray Standard C/C++ compilers support parallel processing using multiple processors on UNICOS systems. Parallel processing is a technique that breaks a computational task into a set of subtasks and then performs each subtask simultaneously. This allows many jobs to run faster by spreading a computational task across multiple processors. The increase in speed of execution depends on the degree of parallelism that is inherent in the program. See *Optimizing Application Code on UNICOS Systems*, for more information.

Tasking can be performed automatically by the compiler (Autotasking) or it can be directed by the user (user-directed tasking). The methods that can be used to accomplish tasking are defined as follows:

- Autotasking is performed automatically by the compiler based on its analysis of the code.

  Autotasking automates loop-level tasking by detecting parallelism in a program and exploiting the parallelism without user intervention. You can add directives to your code that identify loops you know can be tasked and depend upon Autotasking to analyze other loops.

  Generally, Autotasking works best on programs in which most of the work is in nested loops that do not contain function calls.

Autotasking is enabled by specifying the `-h task`*n* option on the command line. For more information on the `-h task`*n* option, see Section 2.11.1, page 20.

- *User-directed tasking*, sometimes called *microtasking* or simply *tasking*, is controlled by the directives you add to your code. This requires that you understand the requirements for tasking and perform your own analysis.

  To direct tasking manually, you must identify the regions of your program that are to run in parallel; then insert tasking directives to specify these regions to the compiler.

  Those jobs that do significant amounts of work and are inherently parallel are candidates for tasking. You must determine where to insert the tasking directives, determine the tasking context of variables, and check whether the results are correct.

The Autotasking directives described in the following sections are not available on UNICOS/mk systems.

3.8.1 Transformations of code for tasking

When analyzing tasking performance, you must understand some of the implementation details of tasking. This section describes the following tasking issues:

- Generation of the master, slave, and unitasked code
- Tasking initialization
- Vectorization with tasking
- Vectorization messages and tasking
- Reserved semaphores and shared registers

3.8.1.1 Master, Slave, and Unitasked Code

When the compiler encounters a parallel region, it creates three distinct sections of code from the region; this is the 3-code model of tasking. These three sections of code are called the master code, the slave code, and the unitasked code. Each of these sections are as large, or larger, than the original code. Therefore, the object file generated for a highly tasked program can be triple the size of the nontasked object file. Compilation time also increases accordingly.

3.8.1.1.1 Master Code

The master code contains the code that sets up the data structures needed for tasking, performs some (or all) of the work in a parallel region, and performs the cleanup operations needed after tasking is complete. The task that executes the master code is known as the*master task*. A copy of the body of the parallel region is placed in the master code. The master code is executed by the same processor that was executing the nontasked code just prior to the parallel region. This processor also resumes execution of the nontasked code following the parallel region. It is the master task that starts all the slave tasks, if necessary.

3.8.1.1.2 Slave Code

The slave code contains a copy of the body of the parallel region and code to determine what part of the parallel region needs to be performed next. The tasks that execute the slave code are known as*slave tasks*. Because the compiler places the slave code in a function, the slave code is sometimes referred to as the *slave function*. The slave code is executed by each of the processors available for tasking, except for the original processor. When processing of the parallel region is complete, all of the processors that were executing the slave function are returned to the operating system.

For debugging purposes, the names of slave functions are in the following form:

`__tsk_`*`name_nnn`*

The *name* is the name of the original function that contained the parallel region, and *nnn* is a 3-digit number indicating how many parallel regions preceded this one in the current function. If the name is longer than 200 characters, only the first 200 characters are used when the slave function's name is created. Numbers are assigned beginning with `000` and are incremented by 1 for each additional parallel region in that function. This counter is reset to `000` for each new function. For example, `__tsk_fctn_002` is the name of the slave function generated for the third parallel region in function `fctn`.

3.8.1.1.3 Unitasked Code

The unitasked code contains only the original parallel region. No tasking code is present in the unitasked code. The unitasked code is executed by the master task if it determines, at run time, that tasking should not be performed. (For example, if the expression for the `if` argument evaluates false or if tasking is already being done at a higher level in the code.) If the unitasked code is executed, no slave processors are used. The term *unitasked code* should not be confused with *nontasked code*. Nontasked code refers to any program that does not have parallel regions, or to the part of the tasked program that is executed outside of any parallel region.

3.8.1.2 Tasking Initialization

A function that potentially executes tasking code requires some initialization. This initialization sets up the arguments for each slave function. These arguments are usually the variables that the slave task shares with the master task and with other slave tasks.

The execution cost for this code is usually small. For functions that contain a large number of distinct parallel regions, however, the cost may become significant.

3.8.1.3 Vectorization with Tasking (Stripmining)

The `chunksize`, `numchunks`, `guided`, and `vector` arguments to the `taskloop` directive can be specified to allow stripmining. *Stripmining* refers to the partitioning of long vector `for` loops into shorter vector chunks and the processing of these chunks through tasked iterations. While stripmining can cause wall-clock time to decrease, it usually increases CPU time because of the reduced vector performance resulting from the smaller vector lengths involved.

Besides executing the vector loop, each processor must execute some loop overhead code for each chunk of the work.

If you specify the `chunksize`, `numchunks`, `guided`, or `vector` arguments, the compiler rewrites the loop as two nested loops. The inner loop becomes a `for` loop that processes the iterations in a vector chunk. This inner loop will be vectorized, subject to the normal rules for vectorization of loops. The outer loop becomes tasking overhead code that processes each of the chunks.

Usually, when you specify the `ivdep` directive prior to the `taskloop` directive the `ivdep` is applied to the original, unitasked version of the loop. When the `ivdep` directive appears prior to a `taskloop` directive containing a `chunksize`, `numchunks`, `guided`, or `vector` argument, the inner `for` loops (as previously described) in both the master and slave code and the original, unitasked version of the loop are treated as though the `ivdep` directive had been specified prior to these loops.

3.8.1.4 Vectorization Messages and Tasking

When vectorization messages are enabled by specification of the `-h report=v` option on the `cc` command line, information is provided regarding all loops in the master, slave, and unitasked versions of the tasked loop. In the master code and slave function, the tasked loop no longer appears as a loop; thus, no vectorization messages are generated. However, the original loop is unaltered in the unitasked code, and a vectorization message will appear for the loop. Any loops inside a tasked loop produce a vectorization message for each occurrence in the master, slave, and unitasked versions of the tasked loop. The loops generated from stripmining (see Section 3.8.1.3, page 65) also cause vectorization messages to be generated.

The line numbers shown in the vectorization messages for loops in the tasked code are the line numbers corresponding to the equivalent loop in the original code, with the `m` suffix for messages from the master code and an `s` suffix for messages from the slave code. Line numbers for messages from the unitasked code have no suffix.

3.8.1.5 Reserved Semaphores and Shared Registers

Because the following semaphores and shared registers are used by tasking on Cray PVP systems, you must not use them in your program:

- Semaphore registers 0-15
- Shared B registers 0-3

- Shared T registers 0–3

If you address these registers in your program (for example, using intrinsic functions), tasking may not work properly.

If you are multi-streaming on a Cray SV1 system, none of the registers will be available.

3.8.2 cncall Directive (UNICOS Systems Only)

Scope: Local

The cncall directive allows a loop to be tasked by asserting that subroutine and function calls within the loop have no loop-related side effects. The cncall directive is an assertion about data dependence. Unlike the taskloop or parallel directives, the compiler can disregard a cncall directive if it detects possible loop-carried dependencies that are not directly related to a call inside the loop. To force tasking, irrespective of dependence issues, use user-directed tasking. For more information on the taskloop directive, see Section 3.8.4, page 70. For more information on the parallel directive, see Section 3.8.3, page 70.

The cncall directive should be inserted immediately preceding the loop to be tasked. The format of this directive is as follows:

```
#pragma _CRI cncall
```

When using this directive, ensure that the following criteria are met for each call within the body of the loop that follows a cncall directive:

- The callee does not modify data in one iteration and reference this same data in a different iteration of the task loop. This rule applies equally to formal parameters and static and global variables.
- The callee does not reference data in one iteration that is defined during another iteration.
- If the callee modifies a formal parameter or a static or global variable, two iterations cannot modify data at the same storage location unless the variables at issue are scoped as private. Following the task loop, the content of the private variables is undefined. The cncall directive does not force the master thread to execute the last iteration of the task loop.

The following examples use the cncall directive.

Example 1:

```
extern float a[], b[], c;
extern void crunch( float *x, float *y, float z);

void
f(int n)
{
   int i;

#pragma _CRI cncall
   for (i = 0; i < n; ++i) {
      crunch( &a[i], &b[i], c );
   }
}
```

In example 1, `a[i]` and `b[i]` denote unique storage locations for each iteration, so `crunch` can read and write `*x` and `*y`. Because `c` is passed by value, `crunch` can read and write `z`.

Example 2 illustrates some of the subtle distinctions involved:

```
extern void tskvalue(int *);

void
s(int ke)
{
   int itask, k, x;

    /* The following loop may not task because   */
    /* the compiler can not be certain that      */
    /* tskvalue defines itask on each iteration. */

#pragma _CRI cncall
   for (k = 0; k < ke; ++k) {
        tskvalue(&itask);   /* itask not necessarily defined. */
        itask = itask + 1;  /* potential recurrence in use of itask.*/
   }

   /* The following loop tasks because in each iteration */
   /* itask is necessarily defined before it is used.    */
#pragma _CRI cncall
    for (k = 0; k < ke; ++k) {
         itask = 0;         /* itask defined before used. */
         tskvalue(&itask);
         itask = itask + 1;
```

```
    }

    /* The following tasks because the cncall directive   */
    /* allows the compiler to assume no loop carried      */
    /* dependence to/from any call-by-reference argument. */
    /* unfortunately, x is socped shared, which is        */
    /* most likely not what the user expects.             */

#pragma _CRI cncall
    for (k = 0; k < ke; ++k) {
        tskvalue(&x);
        itask = x + 1;
    }
}
```

Because the compiler cannot know how a subroutine will use its formal parameters, the compiler must make an educated guess when the `cncall` directive is applied. For some codes, the compiler's guess will not agree with user requirements. For such codes, explicit variable scoping directives are required.

Subroutine `amb`, in example 3, illustrates another subtle pitfall:

```
void
amb(int n, float a[n])
{
   int k;

#pragma _CRI cncall
   for (k = 0; k < n; ++k) {
      s1(a, k);
      s2(a, k);
   }
}
```

In example 3, `a` is a pointer that is scoped `value` by the compiler. So array `a` is effectively `shared`. If `s1` and `s2` only access `a[k]` during iteration `k`, it is correct to scope `a` as `shared`. Likewise, if `s1` and `s2` never modify `a`, `a` must be scoped as `shared`. It would be incorrect to scope `a` as `private`. However, if `s1` were to define all the elements of the array `a`, then array `a` would need to be scoped as `private`, which cannot be done in C or C++ because the name `a` denotes a pointer to the array, not the array itself.

`cncall` directs the compiler to ignore ambiguous dependencies involving function and subroutine calls within the loop that follows. This may lead to

surprising results. In some cases the compiler's scoping choices do not meet requirements or expectations.

The `defaults` clause of the Autotasking `parallel` or `taskloop` directive instructs the compiler to use a simple set of heuristic scoping rules. This contrasts with `cncall`, in which the compiler exploits all available dependence information in its analysis. Given the inherent ambiguities introduced by calls, neither approach can be guaranteed to produce correct results in all circumstances.

When the translation of a `cncall` loop (a loop preceded by a `cncall` directive) is in doubt, you should check the tasking information messages. The `-hreport=t` option on the `CC`(1) and `cc`(1) command lines reports scoping choices for `cncall` loops through messages `6421`, `6422`, and `6423`. For more information on the `-hreport=t` option, see Section 2.18.3, page 31.

3.8.3 `parallel` and `endparallel` Directives (UNICOS Systems)

Scope: Local

The `parallel` directive marks the start of a parallel region. The `endparallel` directive marks the end of a parallel region. Parallel regions are combinations of redundant code blocks (executed by all processors) and partitioned code blocks (portions executed by each processor, such as the iterations of a tasked loop). The `parallel` directive indicates where multiple processors enter execution, which may be different from where they demonstrate a direct benefit (partitioned code block). The format of these directives is as follows:

```
#pragma _CRI parallel [shared(var...)] [private(var...)]
  [value(var...)] [defaults] [if (exp)] [maxcpus (exp)]

#pragma _CRI endparallel
```

Arguments to tasking directives are described in Section 3.8.13, page 79.

3.8.4 `taskloop` Directive (UNICOS Systems)

Scope: Local

The `taskloop` directive indicates that the following `for` loop can be executed in parallel by multiple processors. Although no directive is needed to end a `taskloop` loop, the `endloop` directive (see Section 3.8.5, page 71) can be used to

explicitly do so. Unlike other loop-based directives, the taskloop directive must appear before a for loop.

The taskloop directive can be used either inside or outside of a parallel region. When the directive is used inside a parallel region, the private, shared, value, defaults, if, and maxcpus arguments are not allowed. These arguments, if specified, must be specified on the parallel directive that precedes the taskloop directive. When a taskloop directive is used outside a parallel region, the loop is referred to as a *stand-alone task loop*. The savelast argument can be specified only on stand-alone task loops.

For task loops outside of a parallel region, the format of the taskloop directive is as follows:

```
#pragma _CRI taskloop [shared(var...)] [private(var...)]
   [value(var...)] [defaults] [if (exp)] [maxcpus(exp)]
   [savelast] [dist]
```

For task loops inside a parallel region, the format of the taskloop directive is as follows:

```
#pragma _CRI taskloop [dist]
```

Arguments to tasking directives are described in Section 3.8.13, page 79.

The following example illustrates the use of the taskloop directive:

```
#pragma _CRI taskloop vector
for (i = 0; i < 2000; i++) {
   ...
   xsum = xsum + aa[i]*(bb[i]-cc[aa[i]]);
   xbig = max(abs(aa[i]*bb[i]), xbig);
   ...
}
```

3.8.5 endloop Directive (UNICOS Systems)

Scope: Local

By default, a directive is not needed to end a taskloop loop. The endloop directive is a special terminator for the taskloop directive inside a parallel region. The endloop directive extends the range of the control structure that

contains the taskloop loop. This allows a mechanism to exploit parallelism in loops that contain reduction computations. The endloop directive can appear only in a parallel region. The format of the endloop directive is as follows:

```
#pragma _CRI endloop
```

In the following example, a parallel region is defined that uses a taskloop/endloop pair and a guarded region to implement a reduction computation.

```
      sum = 0;
      big = -1;
#pragma _CRI parallel private(i,xsum,xbig) shared(aa,bb,cc,sum,big)
      xsum = 0;
      xbig = -1;
#pragma _CRI taskloop vector
      for (i = 0; i < 2000; i++) {
        ...
        xsum = xsum + aa[i]*(bb[i]-cc[aa[i]]);
        xbig = max(abs(aa[i]*bb[i]), xbig);
         ...
      }
      #pragma _CRI guard/* protect the update of sum and big */
      sum = sum + xsum;
      big = max(xbig, big);
#pragma _CRI endguard
#pragma _CRI endloop

        ...
    /* ensure that all processors have contributed to */
    /* the sum; all processors are held here until    */
    /* all contributions are in, ensuring that the    */
    /* value of sum and big will be correct for their */
    /* later use within the parallel region.          */
         ...
       if (sum > 1000.0) {
       ...
       }
 #pragma _CRI endparallel
```

In this example, the guarded region protects the update of sum and big, so that each processor does its own update without interference from the others.

(Guarded regions are discussed in Section 3.8.7, page 74.) The endloop directive ensures that no processor can proceed beyond this point until all have contributed to the sum and big values.

3.8.6 case and endcase Directives (UNICOS Systems)

Scope: Local

The case directive serves as a separator between adjacent code blocks that are concurrently executable. The case directive can appear only in a parallel region. The endcase directive serves as the terminator for a group of one or more parallel cases.

The format of the case and endcase directives is as follows:

```
#pragma _CRI case

#pragma _CRI endcase
```

In the following example, the first, second, and third loops execute concurrently:

```
#pragma _CRI parallel private(i) shared(a,b,c)
#pragma _CRI case
        for (i = 0; i < 1000; i++) {
        /* This loop #1 executes in parallel with loops
           #2 and #3. */
           a[i] = 0;
        }
#pragma _CRI case
        /* This loop #2 executes in parallel with loops
           #1 and #3. */
        for (i = 0; i < 1000; i++) {
           b[i] = 1;
        }
#pragma _CRI case
         /* This loop #3 executes in parallel with
            loops #1 and #2. */
        for (i = 0; i < 1000; i++) {
           c[i] = 2;
        }
#pragma _CRI case             /* empty case is allowed */
#pragma _CRI endcase
        .
```

```
        .
        .
#pragma _CRI endparallel
```

3.8.7 guard and endguard Directives (UNICOS Systems)

Scope: Local

The guard and endguard directive pair delimit a guarded region and provide the necessary synchronization to protect (or guard) the code inside the guarded region. A *guarded region* is a code block that is to be executed by only one processor at a time, although all processors in the parallel region execute it.

The format of the guard and endguard directives is as follows:

```
#pragma _CRI guard [exp]

#pragma _CRI endguard [exp]
```

Unnumbered guards do not use the optional parameter *exp* on the guard and endguard directives. Only one processor is allowed to execute in an unnumbered guarded region at a time. If a processor is executing in an unnumbered guarded region, and a second processor wants to enter an unnumbered guarded region, the second processor must wait until the first processor exits the region.

Numbered guards are indicated by the use of the optional parameter *exp*. The expression *exp* must be an integral expression. Only the low-order 6 bits of *exp* are used, thereby allowing up to 64 distinct numbered guards (0 through 63). For optimal performance, *exp* should be an integer constant; the general expression capability is provided only for the unusual case that the guarded region number must be passed to a lower-level function.

The following example illustrates the use of the guard and endguard directives:

```
#pragma _CRI guard     /* protect the update of sum and big */
      sum = sum + xsum;
      big = max(xbig, big);
#pragma _CRI endguard
```

3.8.8 taskprivate Directive (Cray Standard C Compiler)

The taskprivate directive specifies the task private storage class for variables. The format of this directive is as follows (the comma-separated list of variables can be enclosed in parentheses):

```
#pragma _CRI taskprivate variable,...
```

Variables that are given a task private storage class are placed in storage so that each task has a separate copy of the variables; all functions within a task can access the same copy of the task private variable, but no task can access any task private variables belonging to another task.

A primary use for task private variables is efficient porting of macrotasked programs from a shared-memory system (that is, a system, such as VAX, on which independently executing programs can access the other program's memory). On UNICOS and UNICOS/mk systems, independently executing programs cannot access memory belonging to other programs.

This directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with static storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task private, subsequent declarations of that variable in the same source file inherit the task private storage class.

The following example, in which each task gets a private copy of x initialized to 10, illustrates the use of the taskprivate directive

```
main() {
  static int x = 10;
  #pragma _CRI taskprivate x
  ...
}
```

The taskprivate directive takes precedence over the -h common and the -h taskcommon command line options.

The following restrictions apply to the taskprivate variable:

- A `taskprivate` variable cannot also be a soft external.
- The address of a `taskprivate` variable cannot be taken in a constant expression (for example, an initializer).

3.8.9 `taskshared` Directive (Cray Standard C Compiler)

The `taskshared` directive ensures that specified variables are accessible to all tasks (not stored as task private). For example, you can use this directive with the `-h taskprivate` option, to exempt certain variables that would otherwise be task private. The `taskshared` directive overrides the `-h taskcommon` and `-h taskcommon` command line options.

The format of this directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI taskshared variable,...
```

The `taskshared` directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with `static` storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task shared, subsequent declarations of that variable in the same source file inherit the task shared storage class.

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

```
/* The #pragma directive below retains "test" in
   shared storage when the -h taskprivate command line
   option is used. */
int test;
#pragma _CRI taskshared test   /* Keep "test" in taskshared */
```

3.8.10 `taskcommon` Directive

The `taskcommon` directive specifies the task common storage class for variables. The format of this directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI taskcommon variable, ...
```

Variables that are given a task common storage class are placed in storage so that each task has a separate copy of the variables; all functions within a task can access the same copy of the task common variable, but no task can access any task common variables belonging to another task.

A primary use for task common variables is efficient porting of macrotasked programs from a shared-memory system (that is, a system, such as VAX, on which independently executing programs can access the other program's memory). On UNICOS and UNICOS/mk systems, independently executing programs cannot access memory belonging to other programs.

This directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with `static` storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task common, subsequent declarations of that variable in the same source file inherit the task common storage class.

In the following example, the declaration of `x` in `fctn` is the same `x` previously declared to be task common; it, therefore, has task common storage class:

```
int x;
#pragma _CRI taskcommon x
fctn()
{
 extern int x;
}
```

The `taskcommon` directive takes precedence over the `-h common` and `-h taskprivate` command line options.

The following restrictions apply to `taskcommon` variables:

- A `taskcommon` variable cannot be initialized. (A `taskprivate` variable can be initialized, see Section 3.8.8, page 75.) By default, a `taskcommon` variable is initialized to 0.
- A `taskcommon` variable cannot also be a soft external.

- The address of a `taskcommon` variable cannot be taken in a constant expression (for example, an initializer).

3.8.11 `common` Directive

A `common` directive ensures that specified variables are accessible to all tasks (not stored as `taskcommon`). Use this directive, for example, with the `-h taskcommon` option, to exempt certain variables that would otherwise be `taskcommon`. The `common` directive overrides the `-h taskcommon` and `-h taskprivate` command line options. The format of the `common` directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI common variable,...
```

The `common` directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with `static` storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as common, subsequent declarations of that variable in the same source file inherit the common storage class.

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

```
/* The #pragma directive below retains "test" in
   common storage when the -h taskcommon command line
   option is used. */
int test;
#pragma _CRI common test
```

3.8.12 `prefertask` Directive (UNICOS Systems)

Scope: Local

The `prefertask` directive tells the compiler to generate tasked code for the loop that immediately follows it if that loop contains more than one loop in the nest that can be tasked. The directive states a tasking preference but does not guarantee that the loop has no memory dependence hazard. Aggressive tasking (enabled by the `-h task3` command line option) must be enabled for this

directive to take effect. Threshold testing for the loop specified by using the `prefertask` directive is suppressed. The format of the `prefertask` directive is as follows:

```
#pragma _CRI prefertask
```

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

```
for (i = 0; i < n; i++)
{
   #pragma _CRI prefertask
   for (j = 0; j < m; j++)
      e[j][i] = f[j][i] + g[j][i];
}
```

In the preceding example, both loops can be tasked, but the directive directs the compiler to task the inner `for` loop. Without the directive and without any knowledge of `n` and `m`, the compiler tasks the outer `for` loop. Using the directive, the loops are interchanged (to increase parallel granularity) and the resulting outer `for` loop (involving `j`) is tasked. See also the `prefervector` directive in Section 3.7.5, page 60.

3.8.13 Arguments to Tasking Directives

The tasking directive arguments are categorized as context arguments, work distribution arguments, or miscellaneous arguments. Arguments can appear in any order in the directive.

3.8.13.1 Tasking Context

For user-directed tasking, each variable referenced in the parallel region must be assigned a tasking context. *Tasking context* is an attribute that determines how the different processors access a variable in a parallel region. The tasking context (also called *context*) of a variable can be `private`, `shared`, or `value`. The `private`, `shared`, and `value` arguments are called *tasking context arguments*.

It is an error for a variable to be listed more than once in the lists for the tasking context arguments.

Any variables declared inside a parallel region and any variables declared in a function called from inside a parallel region use the default rule (which follows)

to determine their tasking context. The tasking context for these variables cannot be changed.

Note: Variables default to `extern` and `static` default to `shared`. Function parameters and variables declared `taskcommon`, `auto`, and `register` default to `value`.

Unless the `defaults` argument is used, all variables referenced in but declared outside the parallel region must be listed in one of the tasking context argument lists. The tasking context arguments can be used only with the `parallel` directive or the `taskloop` directive for stand-alone taskloops.

In this discussion of tasking context, *global* refers to variables declared at a global level (outside of any function) and *local* refers to variables declared at a local block level. Not all types of identifiers can be placed in the tasking context argument lists. Tasking context lists must not contain tags, constants, function names, labels, expressions of any kind, or members of a `struct`, `union`, or `enum`.

3.8.13.1.1 `private` and `value` Context

Variables with `private` or `value` context have an independent instance for each of the tasks. Generally, scalar variables that are modified inside the parallel region are declared `private` or `value`. In general, loop control variables (LCVs) should be declared `private`; the LCV for tasked loops (both nonordered and semiordered) must be `value` or `private`.

All `private` variables enter the parallel region undefined. All `value` variables enter the parallel region initialized with the value they had just prior to the parallel region. All `private` variables and modified `value` variables are undefined after exiting the parallel region. An exception to this rule occurs when the `savelast` argument is specified with a stand-alone taskloop, and with all `iterate` variables in semiordered tasked loops.

As a debugging aid, try using the `-h msglevel_2` option when compiling your tasked program. If a `private` variable is used before it is initialized, a message will be generated to alert you to this problem.

3.8.13.1.2 `shared` Context

Variables with `shared` context have a single instance that is accessible from all of the tasking processors. Generally, variables with values that are established outside of the parallel region and that are not modified inside of the region are declared `shared`.

For tasked loops, array variables that are sequentially indexed by the loop control variable and modified inside the loop are often shared. Because each processor is working with a different array element, there is no chance for two processors to attempt to modify the same array element at the same time.

Conversely, scalar variables and array variables that are not sequentially indexed by the loop control variable, but which are modified inside the loop, are usually not shared. This is because their value at any given time, as well as their final value, would then be indeterminate.

Caution:

- Modifying a scalar `shared` variable must be done inside a guarded region if there is a possibility that two processors will attempt to update the variable at the same time. This restriction applies to the increment (++) and decrement (--) operators as well as the assignment operators.
- When modifying a `shared` bitfield or character, you must remember that many bitfields and/or characters may be packed into a single word, and that modifying one part of the word must be done with the assurance that no other processor will be modifying any part of that word at the same time.

  Some ways to avoid this problem include using a `chunksize` argument that is a multiple of 8 for tasked loops that sequentially access an array of characters, or placing the offending operation in a guarded region. See Section 3.8.7, page 74, for a description of guarded regions and the `guard` and `endguard` directives.

3.8.13.1.3 Performance Issues

Many times a variable can have `shared` or `value` context without affecting the program results. For example, variables that are referenced but never modified in the parallel region can usually be placed in either the shared or value context lists. In such cases, run-time performance can be improved by adhering to the following rules.

To achieve best performance:

- Place globally declared variables in the `shared` list.
- Place locally declared arrays, structures, and unions in the `shared` list.
- Place all other locally declared variables in the `value` list.

Putting stack variables (`auto`, `register`, and `taskcommon` variables) in `shared` lists causes their addresses to be taken. This involves aliasing problems,

and can cause loops inside and outside parallel regions to not vectorize. Avoid `shared` stack variables, where possible, because of this problem.

3.8.13.2 Context Arguments

The following tasking directive arguments are used to indicate the tasking context for variables referenced in the parallel region. Tasking context is an attribute that determines how the different processors access a variable in a parallel region.

If the `private`, `shared`, or `value` argument is used, at least one variable must be declared in the corresponding list. For these arguments, variable names follow the argument in a comma-separated list enclosed in parentheses.

Argument	Description
`private`	The `private` argument indicates those variables in the parallel region that are to have private context.
`shared`	The `shared` argument indicates those variables in the parallel region that are to have shared context.
`value`	The `value` argument indicates those variables in the parallel region that are to have value context; that is, each task has its own initialized private copy of these variables.
`defaults`	The `defaults` argument indicates that all variables referenced in the parallel region but not listed in the `private`, `shared`, or `value` lists are to have their tasking context determined by using the default rules. The `private`, `shared`, and `value` arguments can be used in conjunction with the `defaults` argument to override the default rules for specified variables.

3.8.13.3 Work Distribution Arguments

The following arguments specify the work distribution policy for the iterations of a tasked loop and can be used only with the `taskloop` directive.

When no explicit scheduling policy is specified, the compiler is free to choose an appropriate work distribution. A loop that is both tasked and vectorized is scheduled with a sufficiently large chunking factor to allow efficient vector execution.

For all work distribution arguments except `single`, each chunk of iterations can be vectorized, subject to the normal rules for vectorization.

Argument	Description
`single`	The `single` argument specifies that the iterations are distributed one at a time to available processors. If no work distribution arguments are specified, the default is `single`.
`chunksize (`*exp*`)`	The `chunksize` argument specifies that the iteration space is broken into chunks of size *exp*, with a possibly smaller residual chunk. *exp* is an integral expression enclosed in parentheses following the `chunksize` argument. This expression is evaluated at run time, just prior to execution of the loop. If `chunksize` is less than or equal to 0, the `chunksize` is assumed to be 1. Otherwise, if the `chunksize` value is greater than the number of iterations in the loop, the value is set to the number of iterations. The `chunksize` expression cannot contain side effects.
`numchunks (`*exp*`)`	The `numchunks` argument specifies that the iteration space is broken into *exp* chunks of equal size, with a possible smaller residual chunk. *exp* is an integral expression enclosed in parentheses following the `numchunks` argument. This expression is evaluated at run time, just prior to execution of the loop. If the `numchunks` expression value is less than or equal to 0, the value is set to 1. If the `numchunks` expression value is greater than or equal to the number of iterations, the value is set to the number of iterations. The `numchunks` expression cannot contain side effects.
`guided`	When `guided` is specified, each of the iterations are handed out in pieces of exponentially decreasing size. Iterations are dispatched in an attempt to balance the load on each processor so that all processors finish their work at approximately the same time. When only 1 iteration remains, it is dispatched to the next processor, and the distribution is complete. The

following is an example of a guided distribution that assumes 8 processors and 1000 iterations:

```
125 110 96 84 74 64 56 49
43 38 33 29 25 22 19 17
15 13 11 10 9 8 7 6
5 4 4 3 3 3 2 2
2 2 1 1 1 1 1 1
1
```

`vector` — The `vector` argument splits the iteration space into chunks of varying sizes down to a minimum size of 64 or 128, depending on the vector length of the target machine. The sizes of the chunks are computed at run time by using an algorithm that attempts to balance the load on each processor so that all processors finish their work at approximately the same time.

3.8.13.4 Miscellaneous Arguments

The following arguments specify miscellaneous arguments that do not fit either of the previous categories.

Argument	Description
`if (`*exp*`)`	The scalar expression (*exp*) enclosed in parentheses that follows the `if` argument is evaluated at run time, just prior to execution of the parallel region. If *exp* is true, tasking is performed. If *exp* is false, tasking is not performed. The `if` expression cannot contain side effects. This argument can be used only with the `parallel` directive or `taskloop` directive for stand-alone task loops (see `taskloop` directive, Section 3.8.4, page 70).
`maxcpus (`*exp*`)`	The `maxcpus` argument specifies the maximum number of CPUs that the parallel region can effectively use. Specifying `maxcpus` does not ensure that *exp* processors will be assigned; it simply specifies the optimal maximum. If *exp* is less than or equal to 0, it is assumed to be 1.
`savelast`	The `savelast` argument indicates that the final iteration of the loop (or loop partition,

when used in conjunction with the vector or chunksize arguments) will be performed by the master processor after all the other iterations (or partitions) have been completed. This argument ensures that all private and value variables with values that are set inside the loop will have the values produced by the final loop iteration when the loop is exited; this is not ensured by default. The savelast argument has no parameters. This argument can be used only with stand-alone task loops (see Section 3.8.4, page 70).

3.9 Multi-streaming Processor (MSP) Directives (Cray SV1 series systems only)

The following subsections describe the multi-streaming processor (MSP) optimization directives.

Note: The MSP is an optional feature. To determine whether MSP is enabled on your system, enter the sysconf(1) command at your system prompt. The HARDWARE output field contains the NMSP= field that shows the number of MSPs configured. For more information, see the sysconf(1) man page.

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 2.9.1, page 18.

The MSP #pragma directives are as follows:

- #pragma nostream (see the following section)
- #pragma preferstream (see Section 3.9.2, page 86)

3.9.1 #pragma nostream Directive (Cray SV1 series Systems Only)

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]
   }
```

3.9.2 `#pragma preferstream` Directive (Cray SV1 series Systems Only)

Scope: Local

The `preferstream` directive tells the compiler to multi-stream 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 multi-streaming 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]
      }
   }
```

3.10 Scalar Directives

The following subsections describe the scalar optimization directives, which control aspects of code generation, register storage, and so on.

3.10.1 align Directive (UNICOS Systems)

The align directive causes functions, loops, or labels to be aligned on instruction buffer boundaries. This increases the size of the compiled program but improves loop performance. When used in global scope, the align directive specifies that functions be aligned on instruction buffer boundaries. When used in local scope, this directive lets you specify that the loop or label following the directive is to be aligned on an instruction buffer boundary.

To determine the number and size of the instruction buffers on your system, use the target(1) command described in the *UNICOS User Commands Reference Manual*.

3.10.1.1 Function Alignment

When the align directive appears in global scope, it must contain a list, optionally parenthesized, of functions that are to be aligned. Each function in the list must be defined somewhere in the file. The code generated for a function that has been specified on a valid, global align directive will be aligned on a machine instruction buffer boundary.

For function alignment, the align directive appears in global scope and has the following format:

```
#pragma _CRI align func...
```

The following example illustrates the use of the align directive when it is used in global scope for function alignment:

```
void mutt(void);
#pragma _CRI align mutt      /* Error: mutt is declared
                                        but not defined in this file */

#pragma _CRI align siamese   /* Error: siamese not defined in the file */
extern long dachshund;
#pragma _CRI align dachshund /* Error: dachshund is not a function */
void dalmatian(void){}
#pragma _CRI align dalmatian /* OK */
```

The align directive is especially useful for small functions that fit entirely into the machine's instruction buffers. By using this directive, you can ensure that such functions are executed entirely within the instruction buffers available on the machine. After the executable code for the function has been read into the

instruction buffers, no additional instruction buffer fetches will be necessary while the function executes.

The code resulting from an inlined function is not aligned, even if the function's name is specified on an `align` directive. The performance improvement obtained by aligning code on an instruction buffer is generally insignificant when compared to that obtained by inlining.

The `-h align` option can be used to specify that all functions defined in the file are to be aligned automatically on instruction buffer boundaries (see Section 2.13.3, page 24).

3.10.1.2 Loop Alignment

The compiler automatically aligns loops when it determines that this will increase performance. It aligns any loops that fit into and require all of the machine's instruction buffers, because the best performance improvement results from executing a loop entirely from within the buffers.

You can specify alignment of a particular loop by using the `align` directive. For loop alignment, the `align` directive must appear in local scope and has the following form:

```
#pragma _CRI align
```

The `align` directive is especially useful for small loops that fit entirely into the machine's instruction buffers. By using this directive, you can ensure that such loops are executed entirely within the instruction buffers available on the machine. After the executable code for the loop has been read into the instruction buffers, no additional instruction buffer fetches are necessary while the loop executes.

The compiler does not automatically align any of the following kinds of loops:

- `if...goto` loops
- Loops containing function calls, unless all function calls have been inlined

In addition, automatic loop aligning is disabled when the `-h scalar0`, `-O0`, `-G`*n*, and `-g` options are specified.

Because the compiler automatically aligns loops, for most programs you will never need to use the `align` directive. However, at times it is desirable to align

loops in addition to those automatically aligned by the compiler. In these cases, you must use the `align` directive.

The `align` directive must appear directly preceding a loop or a label; it must not appear in any other context within local scope. (As described in the previous subsection, the `align` directive may appear in global scope to indicate functions to be aligned.)

The code generated for the statement that follows the `align` directive is aligned on the next instruction buffer boundary. An `align` directive directly preceding a `for` loop causes code starting at the second expression of the `for` statement to be aligned on an instruction buffer boundary. When any code space is left unused, either it is filled with `NOP` instructions or a `jump` instruction is generated, whichever executes faster.

Use of the `align` directive before a label lets you align `if...goto` loops, where the label designates the top of the loop that is the target of the `goto` statement. However, use of the directive for labels is not restricted to this use. Any referenced label that is preceded by an `align` directive will be aligned on an instruction buffer boundary. If the label is never referenced, the `align` directive has no effect.

If a loop following an `align` directive is tasked, the directive applies only to the loop in the unitasked section of code. This is because the original loop does not appear in its original form in the master code and slave function. Any loops that are not tasked and labels following an `align` directive in a parallel region are aligned in all three versions of the generated code (master, slave, and unitasked) (see Section 3.8.1.1, page 63).

Generally, the `align` directive has no effect on automatically aligned loops. However, if a loop is aligned by using the `align` directive, the resulting increase in code size may prevent alignment of an outer loop that would normally have been automatically aligned because the outer loop no longer fits within the instruction buffers. Conversely, an outer loop that would have been too small to be aligned, may now be automatically aligned because of the increased size of an aligned inner loop.

The following example illustrates the use of the `align` directive when it is used in local scope for loop alignment:

```
void fctn(void)
{
   int i;
#pragma _CRI align
   for (i = 0; i < 100; i++) /* This loop will be aligned. */
   {
      .
      .
      .
   }
   i = 0;
#pragma _CRI align
   top_of_loop: ; /* This loop will be aligned. */
      .
      .
      .
      i++;
   if (i < 100) goto top_of_loop;
}
```

3.10.2 cache_align Directive (UNICOS/mk Systems)

The cache_align directive aligns each specified variable on a cache line boundary. This is useful for frequently referenced variables. A *cache* is storage that can be accessed more quickly than conventional memory. A *cache line* is a division within a cache. Properly used, the cache_align directive lets you prevent cache conflicts.

The directive's effect is independent of its position in source. It can appear in global or local scope. The format of the cache_align directive is as follows:

```
#pragma _CRI cache_align var_list
```

In the preceding format, *var_list* represents a list of variable names separated by commas.

3.10.3 cache_bypass Directive (UNICOS/mk Systems)

Scope: Local

The cache_bypass directive specifies that local memory references in a loop should be passed through E registers.

E registers offer fine-grained access to local memory and a higher bandwidth for sparse index array accesses such as gather/scatter operations and large-stride accesses. These operations do not exploit the spatial locality of cache references. Using this directive can greatly decrease run time for gather/scatter operations. The benefits of using this directive are higher with random index streams. Using this directive increases the latency of memory references in return for greater bandwidth, so this directive may increase runtime for loops with a high degree of spatial locality that derive benefit from cache references.

E registers can also be used to initialize large arrays that contain data not immediately needed in cache. This avoids unnecessary reads into cache and improves memory bandwidth efficiency for the initialization.

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

```
#pragma _CRI cache_bypass var_list
```

var_list One or more comma-separated variable names. The variable must have type array of or pointer to (array of or pointer to) a 64–bit scalar type.

This directive precedes the loop that contains data to be accessed through E registers. If both a `cache_bypass` and a `novector` directive are applied to the same loop, the `novector` directive is ignored,

The compiler ignores the `cache_bypass` directive if it determines that it cannot generate code efficiently. To increase the probability of this directive being used, the loop should have the following characteristics:

- The loop must be an inner loop (it must not contain other loops).
- The loop must be vectorizable. You may need to use the `ivdep` directive in conjunction with `cache_bypass` to ensure that the loop is processed.
- The base array or pointer within the loop must be invariant.

Example:

```
/* References of arrays a, b and c bypass cache.
   References to ix and d go through cache. */

void indirect(double a[], double b[], double c[], double d[], int ix[], int n)
{
  int i;

#pragma _CRI cache_bypass a,b,c
#pragma _CRI ivdep
      for (i = 0; i < n; i++) {
        a[ix[i]] = b[ix[i]] + c[ix[i]] * d[i];
      }
}
```

To see the most benefit from the `cache_bypass` directive, you may want to enable loop unrolling. For information on the command line option to control unrolling, see Chapter 2, page 3.

This feature may disable the UNICOS/mk system stream buffer hardware feature for the entire program. This is done on certain Cray T3E platforms because the compiler cannot guarantee correctness in terms of the interaction of the stream buffers and the E register operations generated by this directive. Disabling stream buffers can cause considerable performance degradation for other parts of your program. The stream buffer features can be reenabled by using the `set_d_stream`(3) library function. Consult with your system administrator to determine whether your Cray T3E system falls into this category. If so, see the `streams_guide`(7) man page for details on how and when streams can be safely reenabled in the presence of E register operations.

3.10.4 `concurrent` Directive (Cray SV1 series and Cray T3E Systems Only)

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, multi-streaming, and tasking optimization on Cray SV1 series systems and for pipelining optimization on Cray T3E systems.

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. This clause is only of use for pipelining on Cray T3E systems; any `concurrent` directive with a `safe_distance=` clause will have no effect on other systems.

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]
}
```

3.10.5 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;
          }
       }
    }
```

3.10.6 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 effect of the `noreduction` directive differs depending on your platform.

On UNICOS systems, 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. On UNICOS/mk systems, the `noreduction` directive 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];
}
```

3.10.7 `split` Directive (UNICOS/mk Systems)

Scope: Local

The `split` directive instructs the compiler to attempt to split the following loop into a set of smaller loops.

Such loop splitting attempts to improve single processor performance by making best use of the six stream buffers of the UNICOS/mk system. It achieves this

by splitting an inner loop into a set of smaller loops, each of which allocates no more than six stream buffers, thus avoiding stream buffer thrashing. The stream buffer feature reduces memory latency and increases memory bandwidth by prefetching for long, small-strided sequences of memory references.

The `split` directive has the following format:

```
#pragma _CRI split
```

The `split` directive merely asserts that the loop can profit by splitting. It will not cause incorrect code.

The compiler splits the loop only if it is safe. Generally, a loop is safe to split under the same conditions that a loop is vectorizable. The compiler only splits inner loops. The compiler may not split some loops with conditional code.

The `split` directive also causes the original loop to be stripmined. This is done to increase the potential for cache hits between the resultant smaller loops.

Loop splitting can reduce the execution time of a loop by as much as 40%. Candidates for loop splitting can have trip counts as low as 40. They must also contain more than six different memory references with strides less than 16.

Note that there is a slight potential for increasing the execution time of certain loops. Loop splitting also increases compile time, especially when loop unrolling is also enabled.

For example:

```
#pragma _CRI split
      for (i = 0; i < 1000; i++) {
         a[i] = b[i] * c[i];
         t = d[i] + a[i];
         e[i] = f[i] + t * g[i];
         h[i] += e[i];
      }
```

First, the compiler generates the following loop (notice the expansion of the scalar temporary `t` into the compiler temporary array `ta`):

```
for (i = 0; i < 1000; i++) {
   a[i] = b[i] * c[i];
   ta[i] = d[i] + a[i];
}
for (i=0; i<1000; i++) {
   e[i] = f[i] * ta[i] * g[i];
   h[i] = h[i] + e[i];
}
```

Finally, the compiler stripmines the loops to increase the potential for cache hits and reduce the size of arrays created for scalar expansion:

```
for (i1 = 0; i1 < 1000; i1 += 256) {
   i2 = (i1+256 < 1000) ? i1+256 : 1000;
   for (i = i1; i < i2; i++) {
      a[i] = b[i] * c[i]
      ta[i-i1] = d[i] + a[i]
   }
   for (i = i1; i < i2; i++) {
      e[i] = f[i] * ta[i-i1] * g[i]
      h[i] += e[i]
   }
}
```

If both a `split` and a `novector` directive are applied to the same loop, the `novector` directive is ignored.

3.10.8 `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 and task common 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.

On UNICOS/mk systems, suppress, with no arguments specified, invalidates the entire cache or forces all entities in the cache to be read from memory. This gives suppress a higher performance cost than it has on other architectures, so specifying particular variables can be more efficient.

3.10.9 symmetric Directive (UNICOS/mk Systems)

Scope: Local

The symmetric directive declares that an auto or register variable has the same local address on all processing elements (PEs). This is useful for global addressing using the shmem library functions. For information on the shmem library functions, see the intro_shmem(3) man page. The format for this compiler directive is as follows:

```
#pragma _CRI symmetric var...
```

The symmetric directive must appear in local scope. Each variable listed on the directive must:

- Be declared in the same scope as the directive.
- Have auto or register storage class.
- Not be a function parameter.

Because all PEs must participate in the allocation of symmetric stack variables, there is an implicit barrier before the first executable statement in a block containing symmetric variables.

If a goto statement jumps into a block where a symmetric variable has been declared, the behavior is undefined. If a block is exited by means of a goto, longjmp, and so on, the memory associated with any symmetric variables

declared in that block will be squandered. Neither of these conditions are detected by the compiler.

3.10.10 `unroll` Directive

Scope: Local

The unrolling directive allows the user to control unrolling for individual loops.

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 unroll [n]
```

The *n* argument specifies the total number of loop body copies to be generated. *n* must be in the range of 2 through 63.

If you do not specify a value for *n*, the compiler attempts to 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 can be directed to attempt to unroll all loops generated for the program with the `-h unroll` command line option (see Section 2.14.2, page 25).

On UNICOS/mk systems, the amount of unrolling specified on the `unroll` directive overrides those chosen by the compiler when the `-h unroll` command

line option is specified. On UNICOS systems, 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 two:

```
#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 2.12.1, page 22.
- 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 declaration 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 */
}
```

Cray Standard C++ [4]

The Cray Standard 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 4.1, page 103.

The Dinkum C++ Library documentation, describes the Dinkum C++ Library and is mainly a reference. This documentation is described in Section 4.2, page 103.

4.1 Unsupported Standard C++ Features

The Cray Standard C++ compiler supports the C++ standard except for the features mentioned here. These features are wide characters and multiple locales as the following shows:

- 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 standard 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 following contrasts routines that do not use wide characters and those that do: `fscanf` and `fwscanf`, `sprintf` and `swprintf`)

4.2 Dinkum C++ Libraries

The Cray standard C++ compiler uses the Dinkum C++ libraries which supports standard C++. In order to understand the library you can use the Dinkum C++ Library documentation. This documentation is provided only in HTML form and can be found on your CrayDoc CD, where your administrator installed the documentation, or on the public web site for Cray Inc. documentation. You can also find other references to tutorials and advanced user materials for the standard C++ library in “Related Publications”. Refer to *CrayDoc Installation and*

Administration Guide for more information about the CrayDoc documentation system.

If you have any questions about the contents of the Cray Standard C++ headers, refer to the `/opt/ctl/CC/CC/include` directory.

Cray Standard C Extensions [5]

This chapter describes the Cray Standard C extensions to standard C. A program that uses one or more extensions does not strictly conform to the standard. These extensions are not available in strict conformance mode.

The following are extensions to the C standard:

- Complex data extensions (Section 5.1, page 105)
- `fortran` keyword (Section 5.2, page 106)
- Hexadecimal floating-point constants (Section 5.3, page 106)

5.1 Complex Data Extensions

Cray Standard C extends the complex data facilities defined by standard C. The following are extensions for the complex data facilities:

- Imaginary constants
- Incrementing or decrementing `_Complex` data

The Cray Standard 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 next 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.

5.2 `fortran` Keyword

Note: The `fortran` keyword is not allowed in Cray Standard C++.

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-address; 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.

An example using the `fortran` keyword is shown in Section 9.3.7, page 133.

5.3 Hexadecimal Floating-point Constants

The Cray Standard 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 on UNICOS systems that use Cray floating-point format:

```
float f=0x3ffe800000000000.f;
double g=0xffffffffffffffff.;
main()
{
     printf("f = 0x%16x.f == %g\n", f, f);
     printf("g = 0x%16x. == %g\n", g, g);
}
```

The output from the preceding code is as follows:

```
f = 0x3ffe800000000000.f == 0.125
g = 0xffffffffffffffff. == *.00000
```

Cray Standard C++ Template Instantiation [6]

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*.

Note: The information in this chapter does not pertain to C or the Cray Standard C compiler.

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 Standard 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.

6.1 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 Standard 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 6.5, page 116).
- 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 Standard 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 2.5.8, page 13) 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
bld -q 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
bld -q mylib.a a.o b.o c.o myInstantiationsDir/*.int.o
```

For more information about this alternative see Section 6.3, page 114 and Section 2.5.2, page 12.

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.

6.2 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.

6.3 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 2.5.2, page 12.

6.4 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.

6.5 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 Standard 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 Standard 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.

Predefined Macros [7]

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 the subsections that follow, 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 2.4.2, page 9).

> **Note:** Any of the predefined macros except those required by the standard (see Section 7.1, page 117) 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. These macros are described in the *UNICOS System Libraries Reference Manual*.

7.1 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 Standard C/C++ compilations.
`__cplusplus`	Defined as 1 when compiling Cray Standard C++ code and undefined when compiling Cray Standard C code. The `__cplusplus` macro is

required by the ISO C++ standard, but not the ISO C standard.

7.2 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 operating system is UNIX.
`unix`	Defined as 1 if the operating system is UNIX. This macro is not defined in strict-conformance mode.
`_UNICOS`	Defined as the integer portion of the major release level of the current UNICOS release (for example, 9).

7.3 Macros Based on the Target Machine

The following macros provide information about the characteristics of the target machine:

Macro	Description
`cray`	Defined as 1 on all UNICOS and UNICOS/mk systems. This macro is not defined in strict-conformance mode.
`CRAY`	Defined as 1 on all UNICOS and UNICOS/mk systems. This macro is not defined in strict-conformance mode.
`_CRAY`	Defined as 1 on all UNICOS and UNICOS/mk systems.
`CRAY1`	Defined as 1 on all UNICOS systems; if the hardware is any other machine type, the macro is not defined. This macro is not defined in strict-conformance mode.
`_CRAY1`	Defined as 1 on all UNICOS systems; if the hardware is any other machine type, the macro is not defined.

_CRAYMPP	Defined as 1 on all UNICOS/mk systems. If the hardware is any other machine type, the macro is not defined.
_CRAYSV1	Defined as 1 on all Cray SV1 series systems. If the hardware is any other machine type, the macro is not defined.
_CRAYT3E	Defined as 1 on Cray T3E systems; if the hardware is any other machine type, the macro is not defined.
_CRAYIEEE	Defined as 1 if the targeted CPU type uses IEEE floating-point format; if Cray floating-point format is used, the macro is not defined.
_ADDR32	Defined as 1 if the targeted CPU has 32-bit address registers; if the targeted CPU does not have 32–bit address registers, the macro is not defined.
_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.
_LD64	Defined as 1 if the `long double` basic type has 64 bits of precision; if 128-bit precision is used, the macro is not defined.
_FASTMD	Defined as 1 if the fast multiply/divide sequence is enabled; if the machine type is UNICOS/mk or if fast multiply/divide is not used, the macro is not defined.
_MAXVL	Defined as the maximum hardware vector length (64 or 128); if the machine type is UNICOS/mk, the macro is not defined.

7.4 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.
_CRAYC	Defined as 1 to identify the Cray Standard C/C++ compilers.

Debugging Cray Standard C/C++ Code [8]

The Cray TotalView symbolic debugger is available to help you debug C and C++ codes. In addition, the Cray Standard C/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 Cray TotalView debugger. For more information on these compiler options, see Section 2.17.1, page 29.
- The `-h [no]trunc` option helps identify numerically unstable algorithms. For more information, see Section 2.15.7, page 28.
- 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 2.17.2, page 30. The `#pragma _CRI [no]bounds` directive is described in Section 3.5.2, page 49.
- 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 3.5.4, page 52.
- 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 53.

8.1 Cray TotalView Debugger

The Cray TotalView debugger is designed for use with Cray Standard C, C++, or Fortran source code. The TotalView debugger is documented in *Introducing the Cray TotalView Debugger*.

Some of the functions available in the Cray 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

8.2 Compiler Debugging Options

To use the Cray 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 Cray TotalView debugger allows you to set breakpoints at function entry and exit and at labels.

- `-Gp`

  If you specify the `-Gp` debugging option, the Cray 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 Cray 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 Standard C/C++ compilers do not have to sacrifice runtime 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 [9]

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, which is included in the *Application Programmer's Library Reference Manual*.

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. These calls apply to UNICOS and UNICOS/mk systems unless stated otherwise.

9.1 Calls between C and C++ Functions

The following requirements must be considered when making calls between functions written in C and C++:

- In Cray Standard C++, the `extern "C"` storage class is required when declaring an external function that is written in Cray C or when declaring a Cray Standard 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 Standard C++ objects that are not accessible to Cray Standard 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 Standard C function (`C_add_func`) is called by the Cray Standard C++ `main` function:

```
        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 Standard 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
```

9.2 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 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. UNICOS program linkage macros are described in the *UNICOS Macros and Opdefs Reference Manual*.

In Cray Standard C++, use `extern "C"` to declare the assembly language function.

9.2.1 Cray Assembly Language (CAL) Functions on UNICOS Systems

The use of Cray Assembly Language (CAL) on UNICOS systems is described in the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*.

On UNICOS systems, the `ALLOC`, `DEFARG`, `DEFB`, `DEFT`, `ENTER`, `EXIT`, `MXCALLEN`, and `PROGRAM` macros can be used to define the calling list; B and T register use; temporary storage; and entry and exit points.

9.2.2 Cray Assembler for MPP (CAM) Functions on UNICOS/mk Systems

The use of the Cray Assembler for MPP (CAM) on UNICOS/mk systems is described in the *Cray Assembler for MPP (CAM) Reference Manual*.

On UNICOS/mk systems, the `ALLOC`, `LOAD`, `STORE` and `DEFARG`, `ENTER`, `EXIT`, `ADDRESS`, `VALUE` macros can be used to define local (temporary) storage; entry or exit points; argument processing; and calls to other functions.

9.3 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.

9.3.1 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 9.3.2, page 127.
- 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 9.3.3, page 127.
- Single-dimension arrays of signed 64-bit integers and single dimension arrays of 64-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 Standard C/C++ are incompatible. See Section 9.3.4, page 128.
- Fortran logical values and the Boolean values from C and C++ are not fully compatible. See Section 9.3.4, page 128.
- 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 Standard C, Fortran functions can be declared using the `fortran` keyword (see Section 5.2, page 106). The `fortran` keyword is not available in Cray Standard C++. Instead, Fortran functions must be declared by specifying `extern "C"`.

9.3.2 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 Standard C is strictly by value. To prepare for a function call between two Cray Standard 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 Standard C++ also provides passing by reference.

9.3.3 Array Storage

C and C++ arrays are stored in memory in row-major order; and 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 Standard 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]

9.3.4 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 Standard 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
<fortran.h>	Header file that defines the type _fcd, which maps to the Fortran character descriptor and defines or declares the macros or functions contained in this list.
_cptofcd	Conversion utility that converts a C or C++ character pointer to a Fortran character descriptor.
_fcdtocp	Conversion utility that converts a Fortran character descriptor to a C or C++ character pointer.

`_fcdlen`	Conversion utility that extracts the byte length from the Fortran character descriptor. Because Fortran does not terminate character strings with a null character, `_fcdlen` can be used to determine the last character in the string.
`_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.

For more information on these utilities, see the description of the `_cptofcd`(3) function in the *UNICOS System Libraries Reference Manual*.

9.3.5 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 Standard 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 Standard 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 Standard C and Fortran examples are executed by the following commands, and they produce the output shown:

```
$ cc -c c.c
$ ftn -c f.f
$ segldr 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
$
```

9.3.6 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 a Fortran `COMMON` or `TASKCOMMON` statement, along with the 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/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();
}
```

The following 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
```

The output of the previous C or C++ source code is as follows:

```
In C
  a =  1.0
  b = 2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0
```

The output of the previous Fortran source code is as follows:

```
IN FORTRAN
  STA = 1.
  STB = 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
```

9.3.7 Cray Standard C and Fortran Example

The following example illustrates a Cray Standard C function that calls a Fortran subprogram. The Fortran subprogram follows the Cray Standard C function, and the input and output from this sequence follows the Fortran subprogram.

```
/*                C program (main.c):                   */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

fortran double FTNFCTN (_fcd, int *);

double FLOAT1 = 1.6;
double FLOAT2;  /* Initialized in FTNFCTN */

main()
{
     int clogical, ftnlogical, cstringlen;
     double rtnval;
     char *cstring = "C Character String";
      _fcd ftnstring;

/* Convert cstring and clogical to their Fortran equivalents */
     ftnstring = _cptofcd(cstring, strlen(cstring));
     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);

     rtnval = FTNFCTN(ftnstring, &ftnlogical);

/* Convert ftnstring and ftnlogical to their C equivalents */
     cstring = _fcdtocp(ftnstring);
     cstringlen = _fcdlen(ftnstring);
     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 Standard C function and Fortran subprogram are executed by the following commands and produce the following output:

```
$ cc -c main.c
$ ftn -c ftnfctn.f
$ segldr 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
$
```

9.3.8 Calling a Fortran Program from a Cray Standard C++ Program

The following example illustrates how a Fortran program can be called from a Cray Standard 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 Standard 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
```

9.4 Calling a C and C++ Function from an Assembly Language or Fortran Program

A C or C++ function can be called from Fortran or assembly language. 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 interoperabillity. If you are using the method documented in this section to call C functions from Fortran, keep in mind the information in Section 9.3, page 126.

When calling a Cray Standard C++ function from Fortran or assembly language, observe the following rules:

- The Cray Standard C++ function must be declared with `extern "C"` storage class.
- The program must be linked with the CC(1) command.

The example that follows illustrates a Fortran program that calls a Cray Standard C function. The Cray Standard C function being called, the commands required, and the associated input and output are also included.

```
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 Standard 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;

double CFCTN(_fcd str, int *log)
{
     int slen;
     int clog;
     float returnval;
     char *cstring;
     char newstr[25];

/* Convert str and log passed from Fortran MAIN */
/* into C equivalents */
    slen = _fcdlen(str);
    cstring = malloc(slen+1);
    strncpy(cstring, _fcdtocp(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(_fcdtocp(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 Standard 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 [10]

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.

10.1 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 Standard C/C++ compilers for these cases is summarized in this section.

10.1.1 Messages

All diagnostic messages issued by the compilers are reported through the UNICOS message system. For information on messages issued by the compilers and for information about the UNICOS message system, see Appendix C, page 173.

10.1.2 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 shells, see the `sh`(1), `csh`(1), or `ksh`(1) man pages.

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 Standard C++) refer to interactive devices and buffer them accordingly. For further information, see the description of I/O in the *UNICOS System Libraries Reference Manual*.

10.1.2.1 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 Standard C++, all characters of a name are significant.

10.1.2.2 Types

Table 6, page 142 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 6, page 142. Thus, the `sizeof` operator will return a size that is equal to the value in the *Memory* column of Table 6, page 142 divided by 8 (the number of bits in a byte).

Table 6. Data Type Mapping

	UNICOS		UNICOS/mk	
Type	Representation (bits)	Memory (bits)	Representation (bits)	Memory (bits)
`bool`	8	8	8	8
(Cray Standard C++ only)				
`char`	8	8	8	8
`wchar_t`	64	64	64	64
(Cray Standard C++ only)				
`short`	32	64	32	32
`int`	46/64[1]	64	64	64

[1] Depends on use of the `-h [no]fastmd` option. This option is described in Section 2.15.2, page 26

	UNICOS		UNICOS/mk	
Type	Representation (bits)	Memory (bits)	Representation (bits)	Memory (bits)
`long`	64	64	64	64
`long long`[2]	64	64	64	64
`float`	64	64	32	32
`double`	64	64	64	64
`long double`	128	128	64	64
`float complex`[3]	128 (64 each part)	128	64 (32 each part)	64
`double complex`[3]	128 (64 each part)	128	128 (64 each part)	128
`long double complex`[3]	256 (128 each part)	256	128 (64 each part)	128
`void` and `char` pointers	64	64	64	64
Other pointers	32	64	64	64

10.1.2.3 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 2.7.3, page 15.

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

2 Available in extended mode only.

3 Extension to the Cray Standard C compiler.

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 7. 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 8. 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

10.1.2.4 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.

10.1.2.5 Integers

All integral values are represented in a twos complement format. For representation and memory storage requirements for integral types, see Table 6, page 142.

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 0s on UNICOS systems and 1s on UNICOS/mk systems. On UNICOS/mk systems, this behavior can be modified by using the `-h nosignedshifts` option (see Section 2.7.4, page 15).

On UNICOS/mk systems, the shift operators (>> and <<) use only the rightmost six bits of the second operand. For example, shifting by 65 is the same as shifting by 1. On UNICOS systems, bits higher than the sixth bit are not ignored. Values higher than 63 cause the result to be 0.

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 on UNICOS systems, 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 runtime abort.

10.1.2.6 Floating-point Arithmetic

Cray systems use either Cray floating-point arithmetic or IEEE floating-point arithmetic. These types of floating-point representation are described in the sections that follow.

10.1.2.6.1 Cray floating-point Representation

Types `float` and `double` represent Cray single-precision (64-bit) floating-point values; `long double` represents Cray double-precision (128-bit) floating-point values.

An integral number that is converted to a floating-point number that cannot exactly represent the original value is truncated toward 0. A floating-point number that is converted to a narrower floating-point number is also truncated toward 0.

Floating-point arithmetic depends on implementation-defined ranges for types of data. The values of the minimums and maximums for these ranges are defined by macros in the standard header file `float.h`. All floating-point operations on operands that are within the defined range yield results that are also in this range if the true mathematical result is in the range. The results are accurate to within the ability of the hardware to represent the true value.

The maximum positive value for types `float`, `double`, and `long double` is approximately as follows:

2.7×10^{2456}

Several math functions return this upper limit if the true value equals or exceeds it.

The minimum positive value for types `float`, `double`, and `long double` is approximately as follows:

3.67×10^{-2466}

These numbers define a range that is slightly smaller than the value that can be represented on a UNICOS or UNICOS/mk system, but use of numbers outside this range may not yield predictable results. For exact values, use the values defined in the header file, `float.h`.

A floating-point value, when rounded off, can be accurately represented to approximately 14 decimal places for types `float` and `double`, and to approximately 28 decimal places for type `long double` as determined by the following equation:

$$number\ of\ decimal\ digits = \frac{number\ of\ bits}{\log_2 10.0} \tag{10.1}$$

Digits beyond these precisions may not be accurate. It is safest to assume only 14 or 28 decimal places of accuracy.

Epsilon, the difference between 1.0 and the smallest value greater than 1.0 that is representable in the given floating-point type, is approximately 7.1×10^{-15} for types `float` and `double`, and approximately 2.5×10^{-29} for type `long double`.

10.1.2.6.2 IEEE Floating-point Representation

On UNICOS/mk systems, `float` represents IEEE single-precision (32-bit) floating-point values; `double` and `long double` represent double-precision (64-bit) floating-point values. IEEE extended double precision (128–bit) is not available on UNICOS/mk systems.

On UNICOS systems with IEEE floating-point hardware, `float` and `double` represent IEEE double-precision (64-bit) floating-point values. The `long double` represents IEEE extended double-precision (128-bit) floating-point values. IEEE single-precision (32-bit) is not available on UNICOS systems.

An integral number that is converted to a floating-point number that cannot exactly represent the original value is rounded according to the current rounding mode. A floating-point number that is converted to a floating-point number with fewer significant digits also is rounded according to the current rounding mode on UNICOS/mk systems; on UNICOS systems, the number is rounded to closest, but not in an IEEE round-to-nearest fashion.

Floating-point arithmetic depends on implementation-defined ranges for types of data. The values of the minimums and maximums for these ranges are defined by macros in the standard header file, `float.h`. All floating-point operations on operands that are within the defined range yield results that are also in this range if the true mathematical result is in the range. The results are accurate to within the ability of the hardware to represent the true value.

The maximum positive values are approximately as follows:

3.4×10^{38}	Single (32 bits)
1.8×10^{308}	Double (64 bits)
1.2×10^{4932}	Extended double (128 bits)

The minimum positive values are approximately as follows:

1.8×10^{-38}	Single (32 bits)
2.2×10^{-308}	Double (64 bits)
3.4×10^{-4932}	Extended double (128 bits)

For exact values, use the macros defined in the header file, `float.h`(3c).

Rounding of 32 and 64 bit floating-point arithmetic is determined by the current rounding mode. The 128 bit floating-point arithmetic is rounded to the closest, without regard to the rounding mode. A floating-point value, when rounded off, can be accurately represented to approximately 7 decimal places for single-precision types, approximately 16 decimal places for double-precision types, and approximately 34 decimal places for extended double-precision types as determined by the following equation:

$$number\ of\ decimal\ digits = \frac{number\ of\ bits}{\log_2 10.0} \tag{10.2}$$

Digits beyond these precisions may not be accurate.

Epsilon, the difference between 1.0 and the smallest value greater than 1.0 that is representable in the given floating-point type, is approximately as follows:

1.2×10^{-7}	Single (32 bits)
2.2×10^{-16}	Double (64 bits)
1.9×10^{-34}	Extended double (128 bits)

Upon entering the `main` function at the beginning of the program execution, the rounding mode is set to round to nearest, all floating-point exception status flags are cleared, and traps are enabled for overflow, invalid operation, and division-by-zero exceptions. Traps are disabled for all other exceptions.

10.1.2.7 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 int` 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 `int` in the header `stddef.h`.

If a pointer type's value is cast to a signed or unsigned `int` or `long int`, and then cast back to the original type's value, the two pointer values will compare equal.

Pointers on UNICOS systems differ from pointers on UNICOS/mk systems. The sections that follow describe pointer implementation on each type of system.

10.1.2.7.1 Pointers on UNICOS Systems

Although a pointer value can be stored in an object of integer type, an operation may give different results when performed on the same value treated as an integer or as a pointer. An integer result should not be used as a pointer. For example, do not assume that adding 5 to an integer is the same as adding 5 to a pointer, because the result is affected by the kind of pointer used in the operation. In particular, results may differ from those on a system using a simpler representation of pointers, such as UNICOS/mk systems.

Pointers other than character pointers are internally represented just like integers: as a single 64-bit field. Character pointers use one of the formats shown in Figure 1, page 149, depending on the size of A registers.

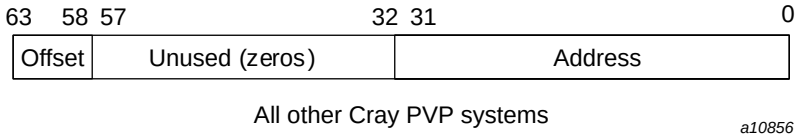

Figure 1. Character Pointer Format

Converting a 64-bit integer to a character pointer type results in a pointer to the byte specified by the value in the offset field of the word specified in the address field.

10.1.2.7.2 Pointers on UNICOS/mk Systems

Pointers on UNICOS/mk 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.

10.1.2.8 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.

10.1.2.9 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 UNICOS or UNICOS/mk 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` or `short` on UNICOS/mk systems 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 `unsigned 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++.

10.1.2.10 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.

10.1.2.11 Declarators

A maximum of 12 pointer, array, and/or function declarators are allowed to modify an arithmetic, structure, or union type.

10.1.2.12 Statements

The compiler has no fixed limit on the maximum number of case values allowed in a `switch` statement.

The Cray Standard C++ compiler parses `asm` statements for correct syntax, but otherwise ignores them.

10.1.2.13 Exceptions

In Cray Standard 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.

10.1.2.14 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).

10.1.3 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 2.20.4, page 36.

The source file character sequence in a `#include` directive must be a valid UNICOS 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 7, page 117.

The #pragma directives are described in section Chapter 3, page 45.

Libraries and Loaders [A]

This appendix describes the libraries that are available with the Cray Standard C/C++ Programming Environment and the loaders, `ld`(1) and `cld`(1).

A.1 Cray Standard C/C++ Libraries Current Programming Environments

Libraries that support Cray Standard C/C++ are automatically available on all systems when you use the `CC`(1), `cc`(1), or `c89`(1) commands 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 loaders 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.

A.2 Loaders

When you issue the `cc`(1), `CC`(1), or `c89`(1) commands 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(1) command. To link C++ object files using one of the loader commands (ld(1) or cld(1)), the -h keep=*files* option (see Section 2.7.1, page 14) must be specified on the command line when compiling source files.

The loaders, ld(1) and cld(1), can be accessed by using one of the following methods:

- You can access the loader directly by using the ld(1) or cld(1) command. You can also use the segldr(1) command to access ld(1).
- You can let the cc(1), CC(1), or c89(1) command choose the loader. This method may cause slower loading and use more memory, but it also has the following advantages:
  - You do not need to know the loader command line interface.
  - You do need to know which loader to call for the targeted machine.
  - 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(1), you need not worry about template instantiation requirements or about loading the compiler-generated static constructors and destructors.

A.2.1 Loader for UNICOS Systems (SEGLDR)

The default loader on all UNICOS systems is SEGLDR. The CC(1), cc(1), and c89(1) commands call SEGLDR by using the ld(1) command. Because SEGLDR was designed specifically for use with UNICOS systems, it offers several advantages. Despite its name, SEGLDR produces both segmented and nonsegmented executable programs and is an efficient and full-featured loader for all types of programs. You can control SEGLDR operations with options on the segldr(1) command line or directives in a directives file. For more information, see the segldr(1) man page and the *Segment Loader (SEGLDR) and ld Reference Manual*.

A.2.2 Loader for UNICOS/mk Systems (cld(1))

The default loader on UNICOS/mk systems is cld(1). The CC(1), cc(1), and c89(1) commands call cld(1) by using the cld(1) command. Because cld(1)

was designed specifically for use with UNICOS/mk systems, it offers several advantages. You can control `cld`(1) operations with options on the `cld`(1) command line or directives in a directives file. For more information, see the `cld`(1) man page.

Cray Standard C/C++ Dialects [B]

This appendix details the features of the C and C++ languages that are accepted by the Cray Standard C / C++ compilers, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other environments.

B.1 C++ Conformance

The Cray Standard C++ compiler accepts the C++ language as defined by the *ISO/IEC 14882:1998* standard, with the exceptions listed in Section B.1.2, page 160.

The Cray Standard 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 48). Command line options are also available to enable and disable anachronisms (see Section B.2, page 161) and strict standard-conformance checking (see Section B.3, page 162, and Section B.4, page 163). The command line options are described in Chapter 2, page 3.

B.1.1 Supported Features

The following features, which are in the *ISO/IEC 14882:1998* standard but not in traditional C++:[1]

- 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.
- A global-scope qualifier is allowed in member references of the form `x.::A::B and p->::A::B`.

1 As defined in *The Annotated C++ Reference Manual* (ARM), by Ellis and Stroustrup, Addison Wesley, 1990.

- 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 (runtime 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.

B.1.2 Unsupported Features

The following features, which are defined in the *ISO/IEC 14882:1998* standard but are not in traditional C++ are not supported:

- `reinterpret_cast` does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- Two-phase name binding in templates as described in the *Working Paper*, is not implemented.
- Putting a `try/catch` around the initializers and body of a constructor is not implemented.
- Template `template` parameters are not implemented.
- Universal character set escapes (e.g., `\uabcd`) are not implemented.
- The `export` keyword for templates is not implemented.

- `extern inline` functions are not supported.
- Covariant return types on overriding virtual functions are not supported.

B.2 C++ Anachronisms Accepted

C++ anachronisms are enabled by using the `-h anachronisms` command line option (see Section 2.4.7, page 11). 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.

B.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. The following 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";
```

B.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 5.2, page 106.
- Cray hexadecimal floating point constants. For more information, see Section 5.3, page 106.

B.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 unsigned.
- 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 [C]

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`(1) command, described in the following section.

For further information about the message system, see the *Cray Message System Programmer's Guide*, or the message system section of the *UNICOS System Libraries Reference Manual*. The introduction to that section can be viewed online as the `message`(3) man page.

C.1 Expanding Messages with the `explain`(1) Command

You can use the `explain`(1) 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 Standard C/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`(1) 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.
```

C.2 Controlling the Use of Messages

The following sections summarize the command line options that affect the issuing of messages from the compiler.

C.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.

C.2.2 Environment Options for Messages

The following environment variables are used by the message system. For more information, see the *Cray Message System Programmer's Guide*.

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.

C.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 Standard C/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.

C.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.
TASKING	Information about tasking optimizations performed on the compiled code.
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 multi-streaming processor (MSP) optimizations performed on the compiled code.
OPTIMIZATION	Information about general optimizations.

C.4 Common System Messages

The four 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).

- `Floating-point Exception`

  A floating-point exception occurs when a program attempts to perform a floating-point operation that is not valid. On UNICOS systems, this error can occur in integer arithmetic because some integer operations are performed with floating-point arithmetic.

Intrinsic Functions [D]

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 Standard 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. When using the Cray Standard C compiler, it is not necessary to declare intrinsic functions. In either case, 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 Standard C intrinsic functions are in the name space reserved for the implementation.

For detailed descriptions of appropriate hardware instructions, see the *Symbolic Machine Instructions Reference Manual*.

> **Note:** Several of these intrinsic functions have both a vector and a scalar version on UNICOS systems. 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 table provides a summary of all C and C++ intrinsic functions and indicates their availability on various platforms. See the appropriate man page for more information.

Table 9. Summary of C and C++ Intrinsic Functions

Function	UNICOS systems	UNICOS/mk systems
_argcount[1]	X	X
_cmr	X	
_dshiftl	X	X
_dshiftr	X	X
_EX	X	
_gbit	X	X
_gbits	X	X
_getvm	X	
_int_mult_upper[2]	X	X
_leadz	X	X
_mask	X	X
_maskl	X	X
_maskr	X	X
_mclr	X	X
_mld	X	X
_memory_barrier		X
_mldmx	X	X
_mmx	X	X
_mul	X	X
_my_pe		X
_numargs	X	X
_num_pes		X
_pbit	X	X
_pbits	X	X
_popcnt	X	X

1 Available only on Cray T90 systems.
2 Available only on systems with IEEE floating-point hardware.

Function	UNICOS systems	UNICOS/mk systems
_poppar	X	X
_ranf	X	X
_readSB	X	
_readSR	X	
_readST	X	
_remote_write_barrier		X
_rtc	X	X
_semclr	X	
_semget	X	
_semput	X	
_semset	X	
_semts	X	
_setvm	X	
_write_memory_barrier		X
_writeSB	X	
_writeST	X	
bte_move[3]	X	

3 Cray SV1 series only

Index

M