Cray Fortran Reference Manual

(8.7)

S-3901

Contents

1 About the Cray Fortran Reference Manual

The *Cray® Fortran Reference Manual* includes reference information for the Cray Fortran compiler.

Cray Fortran Reference Manual (S-3901) 8.7

This version includes updated Fortran reference information to support CCE software release 8.7, released April 19, 2018.

Scope and Audience

This guide is intended as a general overview of the Cray Fortran compiler for users and application programmers.

Typographic Conventions

`Monospace`	Indicates program code, reserved words, library functions, command-line prompts, screen output, file/path names, and other software constructs.
`Monospaced Bold`	Indicates commands that must be entered on a command line or in response to an interactive prompt.
`Oblique` or *`Italics`*	Indicates user-supplied values in commands or syntax definitions.
Proportional Bold	Indicates a **GUI Window**, **GUI element**, cascading menu (**Ctrl**→**Alt**→**Delete**), or key strokes (press **Enter**).
\ (backslash)	At the end of a command line, indicates the Linux® shell line continuation character (lines joined by a backslash are parsed as a single line).

Trademarks

2 Fortran Compiler Introduction

The Cray Compiling Environment 8.7 Fortran compiler supports Cray XC, and Cray CS series systems. The Cray Fortran compiler supports the Fortran 2008 standard (ISO/IEC 1539-1:2010). The Cray Fortran compiler is also documented in man pages, beginning with the `crayftn(1)` man page. Where the information in this manual differs from the man page, the information in the man page supersedes this manual.

The Cray Fortran Programming Environment

The Cray Fortran Programming Environment consists of the tools and libraries used to develop Fortran applications. These are:

- The `ftn` command, which invokes the Cray Fortran compiler. The `ftn` command is properly termed a *compiler driver*, as it is used both to compile source code into object code and to link object code files and libraries to create executable files. This compiling and linking can be performed either as separate processes or as one contiguous process, which has significant implications for file handling considerations. These implications are described later in this section. See the `crayftn(1)` man page for more information.
- CrayLibs libraries, which provides library routines, intrinsic procedures, I/O routines, and data conversion routines.
- The `ftnlx` command, which generates listings and checks for possible errors in Fortran programs. See the `ftnlx(1)` man page for more information.
- The `ftnsplit` command, which splits named Fortran files into separate files with one program unit per file. See the `ftnsplit(1)` man page for more information.
- The `ftnmgen` command, which invokes the Fortran makefile generator. See the `ftnmgen(1)` man page for more information.

Cray Fortran Compiler Messages

The Cray Fortran compiler can produce many messages during compilation and linking. To expand on these messages, use the `explain` command. For more information, see the `explain(1)` man page.

Document-specific Conventions

Cray pointer The term Cray pointer refers to the Cray pointer data type extension.

Fortran Standard Compatibility

In the Fortran standard, the term processor means the combination of a Fortran compiler and the computing system that executes the code. A processor conforms to the standard if it compiles and executes programs that conform to the standard, provided that the Fortran program is not too large or complex for the computer system in question.

The compiler can be directed to flag and generate messages when nonstandard usage of Fortran is encountered. For more information about this command line option (`ftn -en`), see *-d disable_opt and -e enable_opt* or the `crayftn(1)` man page. When the option is in effect, the compiler prints messages for extensions to the standard that are used in the program. As required by the standard, the compiler also flags the following items and provides the reason that the item is being flagged:

- Obsolescent features
- Deleted features
- Kind type parameters not supported
- Violations of any syntax rules and the accompanying constraints
- Characters not permitted by the processor
- Illegal source form
- Violations of the scope rules for names, labels, operators, and assignment symbols

  The Cray Fortran compiler includes extensions to the Fortran standard. Because the compiler processes programs according to the standard, it is considered to be a standard-conforming processor. When the option to note deviations from the Fortran standard is in effect (`-en`), extensions to the standard are flagged with ANSI messages when detected at compile time.

Fortran 2008 Compatibility

No known issues.

Fortran Extensions

The Cray Fortran Compiler supports extended features beyond those specified by the current standard. For more information, see *Cray Fortran Language Extensions*.

Related Fortran Publications

For more information about the Fortran language and its history, consult the following commercially available reference books:

- Fortran 2003 and 2008 standards can be downloaded from *https://wg5-fortran.org/*. The Fortran 2008 standard is also available directly from the ISO.
- Chapman, S. *Fortran 95/2003 for Scientists & Engineers*. McGraw Hill, 2007. ISBN 0073191574.
- Metcalf, M., J. Reid, and M. Cohen. *Modern Fortran*. Oxford University Press, 2011. ISBN-13 978-0 199601424.
- Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, and Brian T. Smith, *The Fortran 2003 Handbook: The Complete Syntax, Features, and Procedures*. Springer, 2009. ISBN 978-1-84628-378-9.

3 Invoke the Cray Fortran Compiler

The `ftn(1)` command invokes the Cray Fortran compiler when the Cray Compiling Environment is loaded.

Cray Fortran Command Syntax

The `ftn` command is a driver that invokes the Cray Fortran Compiler when the Cray Compiling Environment is loaded, and links in the libraries required in order to produce code that can be executed on the Cray compute nodes. Valid `ftn` options include those of the `ftn(1)` driver, as well as those specfic to the Cray Fortran Compiler:

`crayftn` [`-A` *module_name*[, *module_name*] ...] [`-b` *bin_obj_file*] [`-c`] [`-d` *disable_opt*] [`-D` *identifier* [= *value*]] [`-e` *enable_opt*] [`-f` *source_form*] [`-F`] [`-g`] [`-G` *debug_lvl*] [`-h` *arg*] [`-I` *include_dir*] [`-J` *dir_name*] [`-K trap=`*opt* [,*opt*] . . .] [`-l` *lib_file*] [`-L` *ldir*] [`-m` *msg_lvl*] [`-M` *msgs*] [`-N` *col*] [`-o` *out_file*] [`-O` *opt*[,*opt*] ...] [`-p` *module_site*] [`-Q` *path*] [`-r` *list_opt*] [`-R` *runchk*] [`-s` *size*] [`-S`] [`-T`] [`-U` *identifier*[, *identifier*] ...] [`-v`] [`-V`] [`-W`*phase*,"*opt*..."] [`-x` *dirlist*] [`-X` *npes*] [`-Y`*phase*,*dirname*] [`--`] *sourcefile* [*sourcefile* ...]

sourcefile Suffix

The *sourcefile*.*suffix* names the file or files to be processed. The file suffixes indicate the content of each file and determine whether the preprocessor, compiler, assembler, or linker will be invoked.

Table 1. `sourcefile` Suffixes

`.f, .for`	Fixed-format source, compile
`.F, .FOR`	Fixed-format source, preprocess, compile
`.f90`, `.f95`, `.f03`, `.f08`, `.ftn`	Free-format source, compile
`.F90`, `.F95`, `.F03`, `.F08`, `.FTN`	Free-format source, preprocess, compile
`.o`	object file, link
`.a`	assembler source, assemble

The source form specified on the -f*source_form* option overrides the source form implied by the file suffixes.

If only one source file is specified on the command line, the `.o` file is created and deleted. To retain the `.o` file, use the `-c` option to disable the linker. Object files produced by the Cray Fortran, C, C++, or assembler compilers, can be specified. Object files are passed to the linker in the order in which they appear on the `ftn` command line. If the linker is disabled by the `-b` or `-c` option, no files are passed to the linker.

File Types Used or Created by the Compiler

The compiler uses and creates several types of files during processing:

`a.out` Default name of the executable output file. Use the compiler driver command line option `-o` to specify an executable name other than `a.out`.

`.i` Files containing output from the source preprocessor

`.o` Relocatable object code. During compilation, these relocatable object files are saved in the current directory automatically. If CrayPat is used to conduct performance analysis experiments, the object files created during compilation must be kept in order to preserve source-to-executable function mapping. To do so, use the `-h keepfiles` option.

`.a` Library files containing external references

`.s` Assembly language files. Files with `.s` extensions are assembled and written to the corresponding `.o` file.

`.mod` If the `-em` option is specified, the compiler writes a *`modulename`*`.mod` file for each module; modulename is created by taking the name of the module and, if necessary, converting it to uppercase. This file contains module information, including any contained module procedures.

4 Compiler Command Line Options

With the use of appropriate options, it is possible to direct the compiler to generate intermediate translations, including relocatable object files (`-c` option), assembly source expansions (`-S` option). In general, it is possible to save the intermediate files and reference them later on another invocation of the compiler, with other files or libraries included as necessary.

Options that are not recognized by the compiler driver are passed to the linking phase.

If the specified options conflict, 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.

Options with that start with `-h` are the same as options starting with `-O`.

There are many options that start with `-h`. Specify multiple `-h` options using commas to separate the arguments.

Conflicting options

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

```
ftn -h fp1,fp0 myfile.f
```

4.1 Cache Management Options

`-h cache`*`level`*`, -O cache`*`level`*

Default: `cache2`

Specifiy the levels of automatic cache management to perform. Automatic cache management can be overridden by the use of the cache directives (`cache` , `cache_nt` , and `loop_info`).

The values for *`level`*:

0 Cache blocking (including directive-based blocking) is turned off. This level is compatible with all scalar and vector optimization levels.

1 Conservative automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted cache footprint of the symbol in isolation is small enough to experience the reuse.

2 Moderately aggressive automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted state of the cache model is such that the symbol will experience the reuse.

3 Aggressive automatic cache management. Characteristics include potentially high compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the allocation of the symbol to the cache is predicted to increase the number of cache hits.

4.2 Debug Options

`-G` *`debug_lvl`*, `-g`

Controls the tradeoffs between ease of debugging and compiler optimizations. The compiler produces some level of internal debugger information (DWARF) at all times. This DWARF data provides function and source line information to debuggers for tracebacks and breakpoints, as well as type and location information about data variables.

The `-g` or `-G` options can be specified on a per-file basis, so that only part of an application pays the price for improved debugging.

`debug_lvl`

`0` Full DWARF information is available for debugging, but at the cost of a slower and larger executable. Breakpoints can be set at each line. Most optimizations are disabled including floating point optimizations. This level of debugging is supported when `-O ipa0`, `-O scalar0`, `-O thread1`, and `-O vector0` are in effect.

Implies `-h fp0`.

`1` Most DWARF information is available with partial optimization. Some optimizations make tracebacks and limited breakpoints are available in the debugger. Some scalar optimizations and all loop nest restructuring is disabled, but the source code will be visible and most symbols will be available. This allows block-by-block debugging, with the exception of innermost loops. The executable will be faster than with `-g` or `-G0`.

`2` Partial DWARF information. Most optimizations, tracebacks and very limited breakpoints are available in the debugger. The source code will be visible and some symbols will be available. This level allows post-mortem debugging, but local information such as the value of a loop index variable is not necessarily reliable at this level because such information often is carried in registers in optimized code. The executable will be faster and smaller than with `-G1`.

`fast` Compile code for use with Cray fast-track debugging. With the exception of inlining, all code optimizations are enabled. This option is useful only if used in conjunction with a debugger that supports fast-track debugging.

`-h develop`

Default: off

Reduce compile time at the expense of optimization. This option is intended to be used when a program is under development and compiled frequently. This option is different from and independent of the `-O` option. For example, `-O0` disables all optimizations, but sometimes can increase compile time because certain optimizations reduce code size, which allow other phases of the compiler to deal with less code.

-h dir_check

Enables a run time check for the `!dir$ collapse` directive and checks the validity of the `loop_info` count information. Equivalent to the `-Rd` option and added to improve C/Fortran command line compatibility.

-h gasp[=*opt*[:*opt*]]

Default: `disabled`

Request GASP (Global Address Space Performance Analysis) instrumentation. Requests instrumentation of events generated by shared local accesses. Instrumenting these events can add runtime overhead to the application.

When *opt* is specified, the compiler provides additional instrumentation as follows:

local	Requests instrumentation of events generated by shared local accesses. Instrumenting these events can add runtime overhead to the application
functions	Enables function instrumentation. Sets `-hipa0`

-h nodwarf

Disables DWARF generation during compilation. By default, DWARF source line information is generated to support traceback analysis. `-hdwarf` is deprecated. This option has no affect if `-g` or `-G` is specified.

-h zero

Default: disabled

Initializes all undefined local stack, static, and heap variables to 0 (zero). If a user variable is of type character, it is initialized to `NUL`. If logical, initialized to false. The stack variables are initialized upon each execution of the procedure. When used in combination with `-ei`, Real and Complex variables are initialized to signaling `NaNs`, while all other typed objects are initialized to `0`. Objects in common blocks will be initialized if the common block is declared within a `BLOCKDATA` program unit compiled with this option.

4.3 List and Compiler Info Options

-h list=*list_opt*, -r *list_opt*

`-h list=`*list_opt* option is identical to `-r` *list_opt*

The values for *list_opt* (note that all options are off by default):

a Include all reports in the listing (including source, cross references, options, lint, loopmarks, common block, and options used during compilation).

c Listing includes a COMMON block report (lists all common blocks and members of each block).

d Decompiles (translates) the intermediate representation of the compiler into listings that resemble the format of the source code. Use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes that can be made to the Fortran source to improve its performance.

The compiler produces two decompilation listing files, with these extensions, per source file specified on the command line: `.opt` and `.cg`.

e Expand included files in the source listing. This option is off by default.

E Same as `-h list=e`.

i Used with the `-h list=m` option to intersperse loop optimization messages within the loopmark listing. By default, the messages are placed at the bottom of the program unit.

l Lists source code and includes lint style checking. The listing includes the `COMMON` block report (`c` option).

m Produces a source listing with loopmark information. To provide a more complete report, this option automatically enables the `-O negmsgs` option to show why loops were not optimized. If this information is not required, use the `-O nonegmsgs` option on the same command line. Loopmark information will not be displayed if the `-d B` option has been specified.

o Show all options used by the compiler during compilation.

s Lists source code.

T Retain *`file`*`.T` after processing rather than deleting it. The *`file`*`.T` can be used to call `ftnlx` directly. For more information, see the `ftnlx(1)` man page.

x Generate a cross-reference listing.

-v

The `-v` option sends compilation information to the standard error file (stderr). The information generated indicates the compilation phases as they occur and all options and arguments being passed to each processing phase.

-V

Displays compiler version information. Version information consists of the product name, the version number, and the current date and time.

Unlike all other command-line options, this option can be specified without specifying an input file name. If the command line specifies no source file, no compilation occurs.

4.4 General Optimization Options

-h [no]add_paren

Default: `-h noadd_paren`

The `-h add_paren` option automatically adds parenthesis to select associative operations (+,-,*) to encourage left to right evaluation of floating point and complex expressions. For more information, see the crayftn(1) man page. Left to right evaluation is not required by the language standards, but some applications may expect it.

-h [no]aggress, -O [no]aggress

Default: `noaggress`

Provides greater opportunity to optimize loops that would otherwise by inhibited from optimization due to an internal compiler size limitation. `noaggress` leaves this size limitation in effect. With `aggress`, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size.

alias=mode

Default: `-halias=default`

The values for mode are:

none The compiler assumes that the program contains no aliases; i.e. any given object is accessed through exactly one mechanism. For example, an object may be consistently accessed directly or via a unique pointer, but the object may not be accessed via multiple pointers. This mode may improve optimization, but use it with caution; if aliases are present, incorrect code may be generated.

default The compiler infers either the c or fortran alias analysis mode from the source language. C++ uses the same mode as C, with the pointer aliasing rules additionally applying to references.

c The compiler assumes that aliases may exist as allowed by the C language. Briefly, two pointers of similar types (e.g., int and unsigned int) may be used to access the same object, but pointers of different types may not be used to access the same object. There are two exceptions to this rule. First, a pointer to char (char*) may be used to access an object of any type. Second, a structure field may be accessed via a pointer to its type. Accesses to a union are permitted to employ type-punning (i.e., reading from a different field than the one most recently written), provided that such accesses are made only through the union itself and not through pointers to its members.

fortran The compiler assumes that aliases may exist as allowed by the Fortran language. In particular, the compiler assumes that arguments never alias. This mode may be specified for C programs which behave like Fortran to mimic the effect of applying the restrict qualifier to all pointer parameter types.

tolerant The compiler assumes that the program is not well-behaved with respect to pointers, such that it may use a pointer to access an object of a completely different type (e.g., using a pointer to long to access a double). This mode activates an extremely conservative alias analysis that may reduce optimization.

-h [no]autoprefetch, -O [no]autoprefetch

Default: `autoprefetch`

This option controls autoprefetch optimization. This option does not affect the `loop_info noprefetch` or `prefetch` directives.

-h [no]autothread, -O [no]autothread

Default: `noautothread`

The `-h autothread` option enables autothreading.

This is identical to the `-O autothread` option and is provided for command-line compatibility between the Cray C and Fortran compilers.

-h display_opt

The `-h display_opt` option displays the compiler optimization settings currently in force.

This option is identical to the `-eo` option and is provided for command-line compatibility between the Cray C compiler and Fortran compilers.

`-h flex_mp=`*`level`*

Default: `-h flex_mp=default`

The `-h flex_mp=`*`level`* option controls the aggressiveness of optimizations which may affect floating point and complex repeatability when application requirements require identical results when varying the number of ranks or threads.

The values for *`level`* are:

intolerant	Has the highest probability of repeatable results, but also the highest performance penalty.
strict	Uses some safe optimizations and yields higher performance than intolerant, with a high probability of repeatable results.
conservative	Uses more aggressive optimization and yields higher performance than intolerant, but results may not be sufficiently repeatable for some applications.
default	Uses more aggressive optimization and yields higher performance than conservative, but results may not be sufficiently repeatable for some applications.
tolerant	Uses most aggressive optimization and yields highest performance, but results may not be sufficiently repeatable for some applications.

`-h [no]fma`

The `-hnofma` option can be used to disable the generation of fused multiply add (FMA) instructions, if supported on the target hardware. FMA instructions are generated automatically at default, and at `-hfp` levels of 1 or higher. This option could be used for debugging a numerically sensitive application. The use of FMAs are generally better for performance, but introduce different, but not necessarily incorrect, rounding. This will only affect compiler generated FMA opportunities and will not affect pre-built libraries.

`-h fusion`*`n`*`, -O fusion`*`n`*

Default: `fusion2`

Loop fusion can improve the performance of loops, although in rare cases it may degrade performance. The *n* argument allows loop fusion to be turned on or off and determine where fusion should occur.

Loop fusion is disabled when *n* is set to 0.

The values for *n* are:

0 No fusion. Ignore all `fusion` directives and do not attempt to fuse other loops.

1 Attempt to fuse loops that are marked by the `fusion` directive.

2 Attempt to fuse all loops (includes array syntax implied loops), except those marked with the `nofusion` directive.

-h loop_trips=[tiny | small | medium | large | huge], -O loop_trips=[tiny | small | medium | large | huge]

Specifies runtime loop trip counts for all loops in a compiled source file. This information is used to optimize the runtime characteristics of the application.

-h [no]msgs, -O [no]msgs

Default: `nomsgs`

The `-h msgs` option causes the compiler to write optimization messages to `stderr`.

This option is identical to the `-O msgs` option and is provided for command-line compatibility with the Cray C compiler.

-h [no]negmsgs, -O [no]negmsgs

Default: `nonegmsgs`

The `-h negmsgs` option causes the compiler to write messages to `stderr` that indicate why optimizations such as vectorization, inlining, or cloning did not occur. The `-h negmsgs` option enables the `-h msgs` option. The `-h list=a` option enables the `-h negmsgs` option.

This option is identical to the `-O negmsgs` option and is provided for command-line compatibility with the Cray C compiler.

-h [no]omp_trace

Default: `-h noomp_trace`

The `-h omp_trace` option turns the insertion of the CrayPat OpenMP tracing calls on.

-h [no]overindex, -O [no]overindex

Default: `nooverindex`

The `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 `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.

-h [no]pattern, -O [no]pattern

Default: `pattern`

Globally enables pattern matching. When the compiler recognizes certain patterns in the source code, it replaces the construct with a call to an optimized library routine. A loop or statement that has been pattern matched and replaced with a call to a library routine is indicated with an A in the loopmark listing. The `nopattern` option globally disables pattern matching and causes the compiler to ignore the PATTERN and NOPATTERN directives.

Pattern matching is not always worthwhile. If there is a small amount of work in the pattern-matched construct, the call overhead may outweigh the time saved by using the optimized library routine. When compiling using the default optimization settings, the compiler attempts to determine whether each given candidate for pattern matching will in fact yield improved performance.

-h pl=*program_library*

Create and use a persistent repository of compiler information specified by *program_library*. When used with -hwp, this option provides application-wide, cross-file, automatic inlining. The *program_library* repository is implemented as a directory and the information contained in program library is built up with each compiler invocation. Any compilation that does not have the -hpl option will not add information to this repository. Because of the persistence of *program_library*, it is the user's responsibility to manage it. For example, rm -r *program_library* might be added to the make clean target in an application makefile. Because *program_library* is a directory, use rm -r to remove it. If an application makefile works by creating files in multiple directories during a single build, the program_library should be an absolute path, otherwise multiple and incomplete program library repositories will be created. For example, avoid -hpl=./PL.1 and use -hpl=/fullpath/builddir/PL.1 instead.

-h shortcircuit*n*, -O shortcircuit*n*

Default: shortcircuit2

Specify various levels of short circuit evaluation. Short circuit evaluation is an optimization in which the compiler analyzes all or part of a logical expression based on the results of a preliminary analysis. When short circuiting is enabled, the compiler attempts short circuit evaluation of logical expressions that are used in IF statement scalar logical expressions. This evaluation is performed on the .AND. operator and the .OR. operator.

Assume *operand1* .OR. *operand2*. The *operand2* need not be evaluated if *operand1* is true because in that case, the entire expression evaluates to true. Likewise, if *operand2* is true, *operand1* need not be evaluated.

The compiler performs short circuit evaluation in a variety of ways, based on the following command line options

- -O shortcircuit0 disables short circuiting of IF and ELSEIF statement logical conditions.
- -O shortcircuit1 specifies short circuiting of IF and ELSEIF logical conditions only when a PRESENT, ALLOCATED, or ASSOCIATED intrinsic procedure is in the condition.
- The short circuiting is performed left to right. In other words, the left operand is evaluated first, and if it determines the value of the operation, the right operand is not evaluated. The following code segment shows how this option could be used:

```
SUBROUTINE SUB(A)
INTEGER,OPTIONAL::A
IF (PRESENT(A) .AND. A==0) THEN
...
```

- The expression A==0 must not be evaluated if A is not PRESENT. The short circuiting performed when -O shortcircuit1 is in effect causes the evaluation of PRESENT(A) first. If that is false, A==0 is not evaluated. If -O shortcircuit1 is in effect, the preceding example is equivalent to the following example:

```
SUBROUTINE SUB(A)
INTEGER,OPTIONAL::A
IF (PRESENT(A)) THEN
   IF (A==0) THEN
   ...
```

- -O shortcircuit2 specifies short circuiting of IF and ELSEIF logical conditions, and it is done left to right. This is the default for architectures without predicated vector support.

- -O shortcircuit3 specifies short circuiting of IF and ELSEIF logical conditions. It is an attempt to avoid making function calls. If either the left or right operand to .AND. and .OR. operators contain function calls, short circuit evaluation is performed. This is the default for architectures with predicated vector support.

-h profile_generate

The `-h profile_generate` option directs that the source code be instrumented for gathering profile information. The compiler inserts calls and data-gathering instructions to allow CrayPat to gather information about the loops in a compilation unit. If using this option, CrayPat must be run on the resulting executable so the CrayPat data-gathering routines are linked in. For information about CrayPat and profile information, see the *Cray Performance Measurement and Analysis Tools User Guide*.

Do not combine the `-g` and `-h profile_generate` compiler command-line options. Doing so produces reports in CrayPat that contain blank tables and spurious warning messages.

-h [no]safe_addr

Default: `-h safe_addr`

Provides assurance that most conditionally executed memory references are thread safe, which in turn supports a more aggressive use of speculative writes, thereby improving application performance. If `-h nosafe_addr` is specified, the optimizer performs speculative stores only when it can prove absolute thread safety using the information available within the application code.

-h thread*n*, -O thread*n*

Default: `-h thread2`

The `-h thread`*n* option controls the optimization of both OpenMP and automatic threading.

The values for *n*:

0 No autothreading or OMP threading. The `thread0` option is similar to `-h noomp` , but `-h noomp` disables OpenMP only and does not affect autothreading.

1 Specifies strict compliance with the OpenMP standard for directive compilation. Strict compliance is defined as no extra optimizations in or around OpenMP constructs. In other words, the compiler performs only the requested optimizations. If `-h thread1` is specified, it is equivalent to specifying `-h nosafe_addr`.

2 OpenMP parallel regions are subjected to some optimizations; that is, some parallel region expansion. Parallel region expansion is an optimization that merges two adjacent parallel regions in a compilation unit into a single parallel region.

3 Full optimization: loop restructuring, including modifying iteration space for static schedules (breaking standard compliance). Reduction results may not be repeatable.

-h [no]thread_do_concurrent

The `-h thread_do_concurrent` option permits `DO CONCURRENT` nests to be threaded unless prohibited by the `loop_info prefer_nothread` directive. The `-h nothread_do_concurrent` option disallows `DO CONCURRENT` nests to be threaded unless forced by the `loop_info prefer_thread` directive.

Default: `thread_do_concurrent`

-h unroll*n*, -O unroll*n*

Default: `unroll2`

The `-h unroll`*n* option globally controls loop unrolling and changes the assertiveness of the `UNROLL` directive. By default, the compiler attempts to unroll loops, unless the `NOUNROLL` directive is specified of a loop. Generally, unrolling loops increases single processor performance at the cost of increased compile time and code size.

Loop unrolling is disabled when the `-O` or `-h` scalar level is set to 0.

The values for *n*:

0 No unrolling (ignore all unroll directives and do not attempt to unroll other loops).

1 Attempt to unroll loops that are marked by the `unroll` directive.

2 Unroll loops when performance is expected to improve. Loops marked with the `unroll` or `nounroll` directive override automatic unrolling.

-h wp

Enables the whole program mode.

This option causes the compiler backend (IPA, optimizer, codegenerator) to be invoked at application link time, enabling whole program automatic inlining/cloning and future whole program interprocedural analysis (IPA) optimizations. Since the `-hwp` option provides automatic application-wide inlining, the `-Oipafrom` option is no longer needed for cross-file inlining and using these two options together is not permitted. Requires that `-h pl=`*`program_library`* is also specified.

The options `-hpl=` and `-hwp` should be specified on all compiler invocations and on the compiler link invocation. Since `-hwp` delays the compiler optimization step until link time, `-c` compiles will take less time and the link step will take longer. Normally, this is just a time shift from one build phase to another with roughly the same overall compile time. In some cases increased inlining may cause an increase in overall compile time. Using `-hwp` allows the compiler backend to be invoked in parallel during a build. Setting the environment variable `NPROC` controls the number of concurrent compiler backend invocations and this parallelism may reduce overall compile time.

-O opt,*opt*…

The `-O opt` option specifies optimization features. More than one `-O` option can be specified, with accompanying arguments, on the command line. If specifying more than one argument to `-O`, separate the individual arguments with commas and do not include intervening spaces.The `-eo` option or the `ftnlx` command displays the optimization options the compiler uses at compile time. The `-eo` option is identical to the `-h display_opt` option which is provided for command-line compatibility with the Cray C compiler.

The `-O 0`, `-O 1`, `-O 2`, and `-O 3` options allow a general level of optimization to be specified that includes vectorization, scalar optimization, and inlining. Generally, as the optimization level increases, compilation time increases and execution time decreases.

The `-O 1`, `-O 2`, and `-O 3` specifications do not directly correspond to the numeric optimization levels for scalar optimization, vectorization, and inlining. For example, specifying `-O 3` does not necessarily enable vector3. Cray reserves the right to alter the specific optimizations performed at these levels from release to release.

The other optimization options, such as `-O aggress` and `-O cache`*n*, control pattern matching, cache management, zero incrementing, and several other optimization features. Some of these features can also be controlled through compiler directives.

Table 2. Optimization values

	scalar0	scalar1	scalar2	scalar3	vector0	vector1	vector2	vector3	thread0	thread1	thread2	thread3
Low compile cost	X				X				X			
Moderate compile cost		X	X			X	X					
Potentially high compile cost				X				X		X	X	X
Potential numerical differences from unoptimized execution(operator reassociation)		X	X	X	X	X	X	X		X	X	X
Implies at least scalar1						X				X	X	X
Implies at least scalar2							X	X			X	X
Loop nest restructuring		X	X	X		X	X	X	X	X	X	X
Vectorize array syntax statements					X	X	X	X				
OpenMP disabled									X			

All `-O` options, except `-O0`, 1, 2 and 3, are also available with the `-h` option for command-line compatibility with the Cray C compiler.

`-Olevel`

Default: -O2

Specify a general level of optimization that includes vectorization, scalar optimization, cache management, and inlining. Generally, as the optimization level increases, compilation time increases and execution time decreases.

The `-O`*`level`* specifications do not directly correspond to the numeric optimization levels for scalar optimization, vectorization, and inlining. For example, specifying `-O 3` does not necessarily enable `-h vector3`. Cray reserves the right to alter the specific optimizations performed at these levels from release to release. Use the `-h display_opt` option to display the optimization options used during compilation.

The values for *`level`*:

0 Disable optimization, including floating point optimizations. Low compile time, small compile size, no global scalar optimization. Vectorize most array syntax statements, but disable all other vectorizations. Implies `-h fp0`. Some informational messages may not be issued.

1 Conservative optimization: moderate compile time and size, global scalar optimizations, and loop nest restructuring. Results may differ from the results obtained when `-O 0` is specified because of operator

reassociation. No optimizations will be performed that might create false exceptions. Only array syntax statements and inner loops are vectorized and the system does not perform some vector reductions. User tasking is enabled, so OpenMP directives are recognized.

2 Moderate optimization: moderate compile time and size, global scalar optimizations, pattern matching, and loop nest restructuring. Results may differ from results obtained when `-O 1` is specified because of vector reductions. The `-O 2` option enables automatic vectorization of array syntax and entire loop nests. This is the default level of optimization.

3 Aggressive optimization: potentially larger compile time and size, global scalar optimizations, possible loop nest restructuring, and pattern matching. The optimizations performed might create false exceptions in rare instances. Results may differ from results obtained when `-O 1` is specified because of vector reductions.

`-O [no]zeroinc`, `-h [no]zeroinc`

Default: `-O nozeroinc`

The `-O zeroinc` option causes the compiler to assume that a constant increment variable (CIV) can be incremented by zero. A CIV is a variable that is incremented only by a loop invariant value. For example, in a loop with variable J, the statement J = J + K, where K can be equal to zero, J is a CIV. `-O zeroinc` can cause less strength reduction to occur in loops that have variable increments. The `-O [no]zeroinc` option is identical to the `-h [no]zeroinc` option.

4.5 Interprocedural Analysis (IPA) Optimization Options

Interprocedural analysis includes a set of compiler techniques that examine an entire program, as opposed to a single function, to increase the opportunity for optimization, particularly when a program uses many, frequently called functions.

Inlining/cloning transforms code in ways that increase the opportunity for optimization. The user controls inlining/cloning through the use of command line options. If the user desires more fine grained control, directives may be placed within the code. By default, the compiler will attempt inlining/cloning where it deems beneficial. Inlining/cloning may increase object code size.

`-h ipa`*`level`*`, -O ipa`*`level`*

Default: `-h ipa3`

Specifies the level of interprocedural optimization (IPA). `level` may be one of the following values, and includes functionality of previous non-zero levels:

0 All inlining and cloning is disabled. All inlining and cloning compiler directives are ignored.

1 Directive IPA. Inlining/cloning is attempted for call sites and routines that are under the control of a compiler directive. See *Inline and Clone Directives*.

2 Inlining. Inline a call site to an arbitrary depth as long as the expansion does not exceed some compiler-determined threshold. The call site must flatten for any expansion to occur. The call site is said to "flatten" when there are no calls present in the expanded code. The call site must reside within the body of a loop and the entire loop body must flatten. A loop body is said to "flatten" when all call sites within the body of the loop are flattened. Includes level 1.

3 (Default) Inlining and cloning. Includes level 1 and 2. Plus, constant actual argument inlining and tiny routine inlining.

4 Cloning and aggressive inlining. This includes levels 1, 2, and 3. Additionally, a call site does not have to reside in a loop body to inline; nor does the call site have to necessarily flatten to inline.

5 Aggressive inlining and aggressive cloning. Includes levels 1, 2, 3, and 4.

-O ipafrom=source[:source]...

The `-O ipafrom=source[:source]` option explicitly indicates the functions to consider for inlining/cloning. The source arguments identify each file or directory that contains the functions to consider for inlining/cloning. Spaces are not allowed on either side of the equal sign.

All inlining directives are recognized at -Oipa levels > 1.

For information about inlining directives, see *Cray Fortran Directive Use*.

Note that the routines in *source* are not actually linked with the final program. They are simply templates for the inliner. To have a routine contained in *source* linked with the program, include it in an input file to the compilation.

Use one or more of the objects described in the *File Types* table in the *source* argument.

Table 3. File Types

Fortran source files	The routines in Fortran source files are candidates for inline expansion and must contain error-free code. Source files that are acceptable for inlining are files that have one of the following extensions: `.f`, `.F`, `.f90`, `.F90`, `.f95`, `.F95`, `.f03`, `.F03`, `.f08`, `.F08`, `.ftn`, `.FTN`, `.for`, or `.FOR`.
Module files	When compiling with `-em` and `-Omodinline` is in effect, the precompiled module information is written to *modulename*`.mod`. The compiler writes a *modulename*`.mod` file for each module; *modulename* is created by taking the name of the module and, if necessary, converting it to uppercase.
dir	A directory that contains any of the file types described in this table.

Combined inlining is invoked by specifying the `-O ipan` and `-O ipafrom=` options on the command line. This inlining mode will look only in *source* for potential targets for expansion, while applying the selected level of inlining heuristics specified by the `-O ipan` option.

-O [no]modinline, -h [no]modinline

Default: `-O modinline`

Same as the `-h modinline` option. The `-O modinline` option prepares module procedures so they can be inlined by directing the compiler to create templates for module procedures encountered in a module. These templates are attached to *file*`.o` or *modulename*`.mod`. The files that contain these inlinable templates can be saved and used later to inline call sites within a program being compiled.

When `-e m` is in effect, module information is stored in *`modname`*`.mod`. The compiler writes a *`modulename`*`.mod` file for each module; *`modulename`* is created by taking the name of the module and, if necessary, converting it to uppercase.

The process of inlining module procedures requires only that *`file`*`.o` or *`modulename`*`.mod` be available during compilation through the typical module processing mechanism. The USE statement makes the templates available to the inliner. There is no need to specify the *`file`*`.o` or *`modulename`*`.mod` with the `-O ipafrom` option.

When `-O modinline` is specified, the `MODINLINE` and `NOMODINLINE` directives are recognized. Using the `-O modinline` option increases the size of *`file`*`.o`.

To ensure that *`file`*`.o` is not removed, specify this option in conjunction with the `-c` option. For information about the `-c` option, see *-c*.

4.6 Linker Options

-h [system|default]_alloc

Default: `-h default_alloc`

By default, the compiler uses a modified malloc implementation that offers better support for memory needs. The `-h system_alloc` option directs the compiler to link in the native malloc provided by the OS instead of the modified implementation.

-h dynamic

The `-h dynamic` option directs the compiler driver to link dynamic libraries at run time. This option is used to create dynamically linked executable files and may not be used with the `-h static` or `-h shared` options. Note that the preferred invocation is to call the generic `ftn` command with the `-dynamic` option, rather than using this compiler specific option. See the `ftn(1)` man page.

-h [no]pgas_runtime

Default: `pgas_runtime`

The `-h pgas_runtime` option directs the compiler driver to link with the runtime libraries required when linking programs that use UPC, or coarrays. In general, a resource manager job launcher such as `aprun` or `srun` must be used to launch the resulting executable.

The `-hnopgas_runtime` option prevents this runtime library environment from being added to the link line.

Use the `-hnopgas_runtime` option with a program that does not use UPC or coarrays, when it needs to be executed outside of the `aprun`/`srun` job launch context. For example, to test a serial program which does not contain any UPC or coarray code on a login or service node, or fork/exec an executable on a compute node. Also, compile non-coarray Fortran using the `-hnocaf` option.

-h shared

The `-h shared` option creates a library which may be dynamically linked at run time. Note that the preferred invocation is to call the generic `ftn` command with the `-shared` option, rather than using this compiler specific option. See the `ftn(1)` man page.

-h static

The `-h static` option directs the linker to use the static, rather than the dynamic, version of the libraries to create an executable file. Note that the preferred invocation is to call the generic `ftn` command with the `-static` option. See the `ftn(1)` man page.

-l *libname*

The `-l libname` option directs the compiler driver to search for the specified object library file when linking an executable file. To request more than one library file, specify multiple `-l` options.

When statically linking, the compiler driver searches for libraries by prepending *ldir*`/lib` to *libname* and appending `.a`, for each *ldir* that has been specified by using the `-L` option. It uses the first file it finds.

When dynamically linking, the library search process is similar to the static case, with a few differences. The compiler driver searches for libraries by prepending *ldir*`/lib` on the front of *libname* and appending `.so` on the end of it, for each *ldir* that has been specified by using the `-L` option. If a matching `.so` is not found, the compiler driver replaces `.so` with `.a` and repeats the process from the beginning. It uses the first file it finds.

There is no search order dependency for libraries.

If personal libraries are specified by using the `-l` command line option, those libraries are added before the default CCE library list. The `-l` option is passed to the linker. For example, when the following command line is issued, the linker looks for a library named `libmylib.a` and adds it to the top of the list of default libraries.

```
ftn -l mylib target.f
```

-L *ldir*

Changes the `-l` option search algorithm to look for library files in directory `ldir` during link time. To request more than one library directory, specify multiple `-L` options.

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

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

-h byteswapio

Forces byte-swapping of all input and output files for direct and sequential unformatted I/O. Byteswapio is implemented during the linker phase so that it can be uniformly applied across the entire executable.

4.7 Math Options

-h fp*level*, -O fp*level*

Default: `-h fp2`

The `-h fp` option controls the level of floating-point optimizations. The *`level`* argument controls the level of allowable optimization; 0 gives the compiler minimum freedom to optimize floating-point operations, while 4 gives it maximum freedom. The higher the level, the less the floating-point operations conform to the IEEE standard.

Each *`level`*, 1-4, includes the optimizations at the previous *`level`*.

The values for *`level`*:

0 Use this level only when the code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance. The resulting executable code conforms more closely to the IEEE floating-point standard than the default mode (`-h fp2`). Many identity optimizations are disabled. Vectorization of floating-point and complex reductions are disabled. Executable code is slower than higher floating-point optimization levels.

1 Use this option only when the code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance. This option performs generally safe, non-conforming IEEE optimizations, such as folding `a == a` to true, where a is a floating point object. At this level, a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow. Rewrite of division into multiplication by reciprocal is inhibited.

2 Default.

3 Use when performance is more critical than the level of IEEE standard conformance provided by `fp2`. The `-h fp3` option is an acceptable level of optimization for many applications.

4 Use if the application uses algorithms which are tolerant of reduced precision. Do not use `-h fp4` for codes that use Boost I/O or for any codes that do "roll your own" I/O.

The `approx` option allows (`approx`) or prevents (`noapprox`) rewrites of square root and divide expressions using hardware reciprocal approximations. The defaults are `noapprox` for `-hfp0` or `-hfp1` and `approx` for all other optimization levels.

Default: `fp2`

Target	ARM	MIC	Other X86
32-bit `fp2`	-	D, S, E	-
64-bit `fp2`	-	D, S, E	-
32-bit `fp3+`	-	D, S, E	D, S
64-bit `fp3+`	-	D, S, E	-

D	Division and reciprocal
S	Square root and reciprocal square root
E	Exponentiation
MIC	Intel MIC processors (Knights Landing and later)

Table 4. Floating-point Optimization Levels

Optimization Type	`fp0`	`fp1`	`fp2` (default)	`fp3`	`fp4`
Safety	Maximum	High	High	Moderate	Low
Complex divisions	Accurate and slower	Accurate and slower	Fast[1]	Fast[1]	Fast[1]
Exponentiation rewrite	None	None	When optimization benefit is very high[2]	Always[2,3]	Always[2,3]
Strength reduction	None	None	Fast	Fast	Fast
Rewrite division as reciprocal equivalent[4]	None	None	Yes	Aggressive	Aggressive
Floating point reductions	Slow	Fast	Fast	Fast	Fast
Expression factoring	None	Yes	Yes	Yes	Yes
Expression tree balancing	None	None	Yes	Yes	Yes
Inline 32-bit operations[5]	No	No	No	Yes	Yes
Fused multiply-add[6]	No	Yes	Yes	Yes	Yes

4.8 Message Options

`-m msg_lvl`

The `-m msg_lvl` option specifies the minimum compiler message levels to enable. The following list shows the integers to specify in order to enable each type of message and which messages are generated by default.

1 Algebraically correct but may lack precision in boundary cases.
2 Rewriting values raised to a constant power into an algebraically equivalent series of multiplications and/or square roots.
3 Rewriting exponentiations (ab) not previously optimized into the algebraically equivalent form exp(b * ln(a)).
4 For example, x/y is transformed to x * 1.0/y.
5 32-bit division, square root, and reciprocal square root use very fast but less precise code sequences.
6 Uses fused multiply-add instructions on architectures that support it.

msg_lvl	Message types enabled
0	Error, warning, caution, note, and comment
1	Error, warning, caution, and note
2	Error, warning, and caution
3	Error and warning (default)
4	Error

Caution and warning messages denote, respectively, possible and probable user errors.

By default, messages are sent to the standard error file, `stderr`, and are displayed on the terminal. If the `-r` option is specified, messages are also sent to the listing file.

To see more detailed explanations of messages, use the `explain` command. This command retrieves message explanations and displays them online. For example, to obtain documentation on message 500, enter the following command:

```
% explain ftn-500
```

The default *msg_lvl* is 3, which suppresses messages at the comment, note, and caution level. It is not possible to suppress messages at the error level. To suppress specific comment, note, caution, and warning messages, see *-M msgs* below.

To obtain messages regarding nonstandard Fortran usage, specify -e n. For more information about this option, see *-d disable_opt and -e enable_opt* .

-M msgs

The `-M msgs` option suppresses specific messages at the warning, caution, note, and comment levels and can change the default message severity to an error or a warning level. The severity of error-level messages with this option cannot be suppressed or altered.

To suppress messages, specify one or more integer numbers that correspond to the Cray Fortran compiler messages to suppress. To specify more than one message number, specify a comma (but no spaces) between the message numbers. For example, -M 110,300 suppresses messages 110 and 300.

To change a message's severity to an error level or a warning level, specify an E (for error) or a W (for warning) and then the number of the message. For example, consider the following option: -M 300,E600,W400. This specification results in the following messages:

- Message 300 is disabled and is not issued, provided that it is not an error-level message by default. Error-level messages cannot be suppressed and cannot have their severity downgraded.
- Message 600 is issued as an error-level message, regardless of its default severity.
- Message 400 is issued as a warning-level message, provided that it is not an error-level message by default.

-O [no]msgs

Default: `-O nomsgs`

The `-O msgs` option causes the compiler to write optimization messages to stderr.

Similar information in a more-readable format can be obtained by using the `-rm` option instead. Specifying the `-rm` option enables `-O msgs`. For more information, see *-h list=list_opt, -r list_opt*.

-O [no]negmsgs

Default: -O nonegmsgs

The -O negmsgs option causes the compiler to generate messages to stderr that indicate why optimizations such as vectorization or inlining did not occur in a given instance.

The -O negmsgs option enables the -O msgs option. The -rm option enables the -O negmsgs option.

-h ignore_unknown_dirs

Suppresses generation of warning messages when the compiler encounters an unknown directive.

-h error_on_warning

Default: off

The -h error_on_warning option changes the message level of all warning messages to error. This option is off by default.

4.9 Miscellaneous Options

-c

Default: off

The -c option disables the link step and saves the binary object file version of the program in *file*.o, where *file* is the name of the source file and .o is the suffix used. If there is more than one input file, a *file*.o is created for each input file specified.

If only one input file is processed and neither the -b bin_obj_file nor the -c options are specified, the binary version of the program is not saved after the link is completed.

If both the -b bin_obj_file and -c options are specified on the ftn command line, the link step is disabled and the binary object file is written to the name specified as the argument to the -b bin_obj_file option. For more information, see *-b bin_obj_file*.

If both the -o out_file and the -c option are specified on the ftn command line, the link step is disabled and the binary file is written to the *out_file* specified as an argument to -o. For more information, see *-o out_file*.

-o *out_file*

The -o *out_file* option overrides the default executable file name, a.out, with the name specified by the *out_file* argument.

If the -o *out_file* option is specified on the command line along with the -c option, the link step is disabled and the binary file is written to the *out_file* specified as an argument to -o.

-h craylibs_arch_override

Forces Craymath to honor the processor architecture specified by the -h cpu option. Processor architecture is typically specified by loading one of the targeting modules, e.g., craype-sandybridge, but can be overriden at

link time by using the `-h cpu` option. If the `CRAYLIBS_ARCH_OVERRIDE` environment variable is specified, it takes precedence over this option.

-h [no]fp_trap

Default: `fp_trap` if traps are enabled using the `-K trap` option, or if `-Ofp[0,1]` is in effect. Otherwise, the default is `nofp_trap`.

Controls whether the compiler generates code that is compatible with floating-point traps.

-h keepfiles

Default: off

Prevents the removal of the object (`.o`) and temporary assembly (`.s`) files after an executable is created. Normally, the compiler automatically removes these files after linking them to create an executable. Since the original object files are required to instrument a program for performance analysis, if CrayPat is to be used to conduct performance analysis experiments, use this option to preserve the object files.

-h keep_frame_pointer*opts*

Default: `off`

Retain call stack information back to main entry point for CrayPat performance sampling. Prevents call stack frame from being optimized out of a function so CrayPat performance sampling is able to trace call stack back to entry point.

-h loop_trips=[tiny | small | medium | large | huge], -O loop_trips=[tiny | small | medium | large | huge]

Specifies runtime loop trip counts for all loops in a compiled source file. This information is used to optimize the runtime characteristics of the application.

-h pic, -h PIC

Generate position independent code (PIC), which allows a virtual address change from one process to another, as is necessary in the case of shared, dynamically linked objects. The virtual addresses of the instructions and data in PIC code are not known until dynamic link time. For the Cray implementation, the `pic` and `PIC` options have the same effect and should be used to compile codes using more than 2GB of static memory, or for creating dynamically linked libraries.

-K trap=*exceptions*

Default: off

Enable traps for the specified *`exceptions`*. By default, no exceptions are trapped. Enabling traps by using this option also has the effect of setting `-h [no]fp_trap`.

If the specified options contradict each other, the last option predominates. For example, `-K trap=none,fp` is equivalent to `-Ktrap=fp`.

This option does not affect compile time optimizations; it detects run time exceptions.

This option is processed only at link time and affects the entire program; it is not processed when compiling subprograms. Use this command line option to set traps beginning with execution of the main program. The

program may subsequently change these settings by calling intrinsic or library procedures. If this option is used, `-hfp_trap` may need to be specified when other files of the application are compiled.

denorm	Trap on denormalized operands.
divz	Trap on divide-by-zero.
fp	Trap on divz, inv, or ovf exceptions.
inexact	Trap on inexact result (i.e., rounded result). Enabling traps for inexact results is not recommended.
inv	Trap on invalid operation.
none	Disables all traps (default).
ovf	Trap on overflow (i.e., the result of an operation is too large to be represented).
unf	Trap on underflow (i.e., the result of an operation is too small to be represented).

`-o` *`outfile`*

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 source file, a relocatable object file named outfile is produced.

`-R runchk`

The `-R runchk` option allows any of a group of run time checks for the program to be specified. To specify more than one type of checking, specify consecutive *`runchk`* arguments, such as: `-R bs`.

Performance is degraded when run time checking is enabled. This capability, though useful for debugging, is not recommended for production runs.

The run time checks available are as follows:

runchk	Checking performed
b	Enables checking of array bounds. If a problem is detected at run time, a message is issued but execution continues. The NOBOUNDS directive overrides this option. For more information, see *NOBOUNDS*. Bounds checking behavior differs with the optimization level. At the default optimization level, `-O 2`, some run time checking is inhibited. Complete checking is guaranteed only when optimization is turned off by specifying `-O 0` on the `ftn` command line.
c	Enables run time conformance checking of array or coarray operands in array expressions. Even without the `-Rc` option, such checking is performed during compilation when the dimensions of array operands can be determined.
d	Enables directive checking at run time. Errors detected at compile time are reported during compilation and so are not reported at run time. The `collapse` directive is checked, as are the `loop_info` clauses *`min_trips`* and *`max_trips`*. Violation of a run time check results in an immediate fatal error diagnostic.
p	Generates run time code to check the association or allocation status of referenced POINTER variables, ALLOCATABLE arrays, or assumed-shape arrays. A warning message is issued at run time for references to disassociated pointers, unallocated allocatable arrays, or assumed shape dummy arguments that are associated with a pointer or allocatable actual argument when the actual argument is not associated or allocated.

s Enables checking of character substring bounds. This option behaves similarly to option `-R b`.

Bounds checking behavior differs with the optimization level. At the default optimization level, `-O 2`, some run time checking is inhibited. Complete checking is guaranteed only when optimization is turned off by specifying `-O 0` on the `ftn` command line.

-S

Compiles the named 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`.

-V

Displays compiler version information. Version information consists of the product name, the version number, and the current date and time.

Unlike all other command-line options, this option can be specified without specifying an input file name. If the command line specifies no source file, no compilation occurs.

-X *npes*

Specifies the number of Fortran images in the program. The *npes* value must match the number of images that will be specified through aprun or srun at job launch. Ensure that all object files are compiled using the same *npes* value. When using mixed *npes* values or if the number of PEs provided at run time differs from the `-X` *npes* value, a run time error will be received.

-W *phase*, "*args*"

Passes *args* directly to a *phase* of the compiling system. *phase* indicates a compiling system phase as shown:

Phase	Compiling System Phase	Command
0 (zero)	Compiler	`ftn`
a	Assember	`as`
c	CUDA linker	nvlink
l	linker	`ld`
r	Lister	`ftnlx`
x	PTX assember	`ptxas`

args to be passed to compiler system phases can be entered in either of two styles:

- *args* enclosed in double quotes. Spaces appear within *args*.
- *args* not enclosed in double quotes. No spaces appear within *args*. 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` to the linker.

```
%ftn -Wl,"-e name" file.F08
```

```
%ftn -Wl,-e,name file.F08
```

```
% ftn -Wl,-ename file.F08
```

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

The `-Wa"`*`assembler_opt`*`"` option passes assembler_opt directly to the assembler. For example, `-Wa"-h"` passes the `-h` option directly to the as command, directing it to enable all pseudos, regardless of location field name. This option is meaningful to the system only when file.s is specified as an input file on the command line. For more information about assembler options, see the as(1) man page.

The `-Wl,-rpath` *`ldir`* option changes the runtime library search algorithm to look for files in directory *`ldir`* at link time. To request more than one library directory, specify multiple `-rpath` options. At link time, all *`ldir`* arguments are added to the executable.

A library may be found at link time with a `-L` option, but may not be found at run time if a corresponding `-Wl,-rpath` option was not supplied. Also, note that the compiler driver does not pass the `-rpath` option to the linker. Explicitly specify `-Wl,-rpath` when using `-L`.

The dynamic linker will search all *`ldir`* paths first for shared dynamic libraries at runtime, with one exception. The Linux environment variable `LD_LIBRARY_PATH` precedes all other search paths for shared dynamically linked libraries. The use of `LD_LIBRARY_PATH` is discouraged! Setting `LD_LIBRARY_PATH` changes the shared dynamically linked library search paths for all executable files in the environment.

The `-Wr"`*`lister_opt`*`"` option passes *`lister_opt`* directly to the `ftnlx` command. For example, specifying `-Wr"-o file.o"` passes the argument cfile.o directly to the `ftnlx` command's `-o` option; this directs `ftnlx` to override the default output listing and put the output file in cfile.o. If specifying the `-Wr"`lister_opt`"` option, specify the `-r` *`list_opt`* option. For more information about options, see the `ftnlx` man page.

The `-Wx,` *`args`* option can be used to pass command line arguments to the PTX assembler for OpenACC applications.

The `-Wc,` *`args`* option can be used to pass command line arguments to the CUDA linker for OpenACC applications.

`-Y` *`phase`*`,` *`dirname`*

Specifies a new directory, *`dirname`*, from which the designated *`phase`* should be executed. *`phase`* indicates a compiling system phase as shown:

Phase	Compiling System Phase	Command
0 (zero)	Compiler	`ftn`
a	Assembler	`as`
l	Linker	`ld`

Because there is no separate preprocessor, `-Yp` and `-Y0` are equivalent.

4.10 Miscellaneous Fortran Specific Options

--

The -- symbol signifies the end of options. After this symbol, specify files to be processed. This symbol is optional. It may be useful if the input file names begin with one or more dash (-) characters.

-A *module_name*[, *module_name*] ...

The `-A module_name` , *`module_name`* ... option directs the compiler to behave as if a USE *`module_name`* statement was entered for each *`module_name`* in the Fortran source code. The USE statements are entered in every program unit and interface body in the source file being compiled.

-b *bin_obj_file*

The `-b bin_obj_file` option disables the link step and saves the binary object file version of the program in *`bin_obj_file`*.

Only one input file is allowed when the `-b bin_obj_file` option is specified. If there is more than one input file, use the `-c` option to disable the link step and save the binary files to their default file names. If only one input file is processed and neither the `-b` nor the `-c` option is specified, the binary version of the program is not saved after the link is completed.

If both the `-b bin_obj_file` and `-c` options are specified on the `ftn` command line, the link step is disabled and the binary object file is written to the name specified as the argument to the `-b bin_obj_file` option. For more information about the `-c` option, see *-c*.

By default, the binary file is saved in *`file`*`.o`, where *`file`* is the name of the source file and .o is the suffix used.

-d *disable_opt* and -e *enable_opt*

The `-d disable` and `-e enable` options disable or enable compiling options. To specify more than one compiling option, enter the options without separators between them; for example, -eaf. The *Compiling Options* table below shows the arguments to use for *`disable`* or *`enable`*.

Table 5. Compiling Options

args	**Action, if enabled**
0	Initializes all undefined local stack, static, and heap variables to 0. If a user variable is of type character, it is initialized to NUL. If a user variable is type logical, it is initialized to false. The stack variables are initialized upon each execution of the procedure. Enabling this option can help identify problems caused by using uninitialized numeric and logical variables. When used in combination with `-ei`, Real and Complex variables are initialized to signaling NaNs, while all other typed objects are initialized to 0. Objects in common blocks

args	**Action, if enabled**
	will be initialized if the common block is declared within a BLOCKDATA program unit compiled with this option. Default: disabled
a	Aborts compilation after encountering the first error. Default: disabled
A	Treat all module variables as PUBLIC. Does not override explicit PRIVATE statements or attributes. Disabling this option has the effect of including a PRIVATE statement in the specification part of the module. Default: enabled
b	If enabled, issue a warning message rather than an error message when the compiler detects a call to a procedure with one or more dummy arguments having the TARGET, VOLATILE or ASYNCHRONOUS attribute and there is not an explicit interface definition. Default: disabled
B	Generates binary output. If disabled, inhibits all optimization and allows only syntactic and semantic checking. Default: enabled
c	Interface checking: use Cray's system modules to check library calls in a compilation. If there is a procedure with the same name as one in the library, errors will be received because the compiler does not skip user-specified procedures when it performs the checks. Default: disabled
C	Enable/disable some types of standard call site checking. The current Fortran standard requires that the number and types of arguments must agree between the caller and callee. These constraints are enforced in cases where the compiler can detect them, however, specifying `-dC` disables some of this error checking, which may be necessary in order to get some older Fortran codes to compile. If error checking is disabled, unexpected compile-time or run time results may occur.

args	**Action, if enabled**
	The compiler by default attempts to detect situations in which an interface block should be specified but is not. Specifying `-dC` disables this type of checking as well. Default: enabled
d	Controls a column-oriented debugging feature when using fixed source form. When the option is enabled, the compiler replaces the D or d characters in column 1 of the source with a blank and treats the entire line as a valid source line. This feature can be useful, for example, during debugging to insert PRINT statements. When disabled, a D or d character in column 1 is treated as a comment character. Default: disabled
D	The -eD option enables all debugging options. This option is equivalent to specifying the `-G0` with the `-m2`, `-r1`, and `-R bcdsp` options. See also `-ed`. Default: disabled
e	Enable/disable masking expression support for non-integer type operands. This allows masking expressions to be evaluated without type conversion. For example, with this enabled, `real_variable = real_variable .or. another_real_variable` will be evaluated as a bitwise `or` and the assignment will occur without type conversion. By default, the compiler will turn `.or.` into the `IOR` intrinsic and type conversion will occur when the value is assigned to the `real_variable`. This option may not be supported in all situations because of new compiler optimization requirements. Default: disabled
E	The -eE option enables existing declarations to duplicate the declarations contained in a used module. Therefore, there is no need to modify the older code by removing the existing declarations. Because the declarations are not removed, the associated objects will duplicate declarations already in the code, which is not standard.

args	**Action, if enabled**

	Existing declarations of a procedure must match the interface definitions in the module; otherwise an error is generated. Only existing declarations that declare the function name or generic name in an EXTERNAL or type statement are allowable under this option.
	
	This example illustrates some of the acceptable types of existing declarations. Program older contains the older code, while module m contains the interfaces to check.
	
	`module m`
	`interface`
	`  subroutine one(r)`
	`  real :: r`
	`F  end subroutine`
	
	`  function two()`
	`  integer :: two`
	`  end function`
	`end interface`
	`end module`
	
	`program older`
	`use m  !Or use -Am on the`
	`compiler command line`
	`external one  !Use associated`
	`objects`
	`integer :: two  !in declarative`
	`statements`
	
	`call one(r)`
	`j = two()`
	`end program`
	
	Default: disabled

f	Turns .mod files into lowercase names for makefile flexibility. Default: disabled
F	Controls preprocessor expansion of macros in Fortran source lines. Default: enabled
g	Allows branching into the code block for a DO or DO WHILE construct. Historically, codes used branches out of and into DO constructs. Fortran standards prohibit branching into a DO construct from outside of that construct. By default, the Cray Fortran compiler will issue an error for this situation. Cray does not

args	**Action, if enabled**
	recommend branching into a DO construct, but if `-eg` is specified, the code will compile. Default: disabled
h	Enables support for 8-bit and 16-bit INTEGER and LOGICAL types that use explicit kind or star values. By default (`-eh`), data objects declared as INTEGER(kind=1) or LOGICAL(kind=1) are 8 bits long, and objects declared as INTEGER(kind=2) or LOGICAL(kind=2) are 16 bits long. When this option is disabled (`-dh`), data objects declared as INTEGER(kind=1), INTEGER(kind=2), LOGICAL(kind=1), or LOGICAL(kind=2) are 32 bits long. Vectorization of 8- and 16-bit objects is deferred. Default: enabled
I	Treats all variables as if an IMPLICIT NONE statement had been specified. Does not override any IMPLICIT statements or explicit type statements. All variables must be typed. Default: disabled
i	Initializes all undefined local stack, static, and heap variables of type `REAL` or `COMPLEX` to an invalid value (signaling NaN). Stack variables are initialized upon each execution of the procedure. Objects in common blocks will be initialized if the common block is declared within a `BLOCKDATA` program unit compiled with this option. Default: disabled
j	Executes DO loops at least once. Default: disabled
m	When this option is enabled, the compiler creates `.mod` files to hold module information for future compiles. When it is disabled, and a module is compiled, the compiler deletes any existing `MODULENAME.mod` files it finds in the output directory before creating new module information in the `.o` file. By default, module files are written to the current working directory. Use the `-J dir_name` option to specify an alternate output directory for `.mod` files only.

args	Action, if enabled
	For more information about the `-J dir_name` option, *`-J dir_name`*, below. Whether this option is enabled or disabled, the search order for satisfying modules references in USE statements is as follows: The current working directory. Any directories or files specified with the `-p` option. Any directories specified with the `-I` option. Any directories or files specified with the *`FTN_MODULE_PATH`* environment variable. When searching within a directory, the compiler first checks all `.mod` files, then the `.o` files, and then the `.a` files. The compiler creates modules through the MODULE statement. A module is referenced with the USE statement. All `.mod` files are named *`modulename`*`.mod`, where *`modulename`* is the name of the module specified in the MODULE or USE statement. Default: disabled
n	Generates messages to note all nonstandard Fortran usage. Default: disabled
N	`-eN` is the same as `-en`, except that the messages are ERROR level. If `-dN` is specified, ANSI messages are not generated. This is the same as `-dn`. If multiple `-en`, `-dn`, `-eN`, or `-dN` options are specified, the last one encountered takes precedence. Default: disabled
o	Display to `stderr` the optimization options used by the compiler for this compilation. Default: disabled
p	Controls double precision type. This option can only be enabled when the default data size is 64 bits (`-s default64` or `-s real64`).

args	Action, if enabled
	When `-s default64` or `-s real64` is specified, and double precision arithmetic is disabled, DOUBLE PRECISION variables and constants specified with the `D` exponent are converted to default real type (64-bit). If double precision is enabled (`-ep`), they are handled as a double precision type (128-bit). Similarly when the `-s default64` or `-s real64` option is used, variables declared on a DOUBLE COMPLEX statement and complex constants specified with the `D` exponent are mapped to the complex type in which each part has a default real type, so the complex variable is 128-bit. If double precision is enabled (`-ep`), each part has double precision type, so the double complex variable is 256-bit.
P	Performs source preprocessing on Fortran source files, but does not compile (see *sourcefile Suffix* under for valid file extensions). When specified, source code is included by #include directives but not by Fortran INCLUDE lines. Generates *`file`*`.i`, which contains the source code after the preprocessing has been performed and the effects have been applied to the source program. For more information about source preprocessing, see *Source Preprocessing*. Default: disabled
q	Aborts compilation if 100 or more errors are generated. Default: enabled
Q	Controls whether the compiler accepts variable names that begin with a leading underscore (_) character. For example, when Q is enabled, the compiler accepts _ANT as a variable name. Enabling this option can cause collisions with system name space, such library entry point names. Default: disabled
R	Compiles all functions and subroutines as if they had been defined with the RECURSIVE attribute. Default: disabled
s	Scales the values of the *`count`* and *`count_rate`* arguments for the SYSTEM_CLOCK intrinsic function down by a factor of 2 **14 (16384) if the storage size of the value of each of the count and count-rate argumens is 32 bits. Otherwise, no scaling occurs.

args	Action, if enabled
	Default: enabled
S	Generates assembly language output and saves it in *file*.s. Default: disabled
T	Controls preprocessing of Fortran source files. When enabled, source preprocessing is performed. Macro expansion within Fortran source lines is enabled but can be controlled by the -e/d F command line option. When disabled, -dT, preprocessing of the Fortran source file is not performed, even for files with upper case suffixes, like file.F90. Default: When not specified, default is to honor the case of the file name suffix, and other preprocessing options such as -e/d Z -e/d P and -e/d F
v	Allocates variables to static storage. These variables are treated as if they had appeared in a SAVE statement. The following types of variables are not allocated to static storage: automatic variables (explicitly or implicitly stated), variables declared with the AUTOMATIC attribute, variables allocated in an ALLOCATE statement, and local variables in explicit recursive procedures. Variables with the ALLOCATABLE attribute remain allocated on procedure exit, unless explicitly deallocated, but they are not allocated in static memory. Variables in explicit recursive procedures consist of those in functions, in subroutines, and in internal procedures within functions and subroutines that have been defined with the RECURSIVE attribute. The STACK compiler directive overrides -ev; for more information about this compiler directive, see *STACK*. Default: disabled
w	Enables support for automatic memory allocation for allocatable variables and arrays that are on the left hand side of intrinsic assignment statements. Using this option may degrade run time performance, even when automatic memory allocation is not needed. It can affect optimizations for a code region containing an assignment to allocatable variables or arrays; for example, by preventing loop fusion for multiple array syntax assignment statements with the same shape.

args	Action, if enabled
	Default: enabled
X	If a module variable has initializers, implicit or explicit, and the variable has greater than 10,000 elements to be initialized, optionally create a new module procedure to do the initialization at run time before MAIN is called. Enabling this option may significantly reduce compile time and reduce the size of the executable for some code, while increasing execution time. If performance is the only issue, disable this option. Default : enabled
z	Initialize all memory allocated by Fortran ALLOCATE statements to zero. This option applies only for the current source file and should be specified for each source file compilation where this behavior is desired. Default: disabled
Z	Performs source preprocessing and compilation on Fortran source files (see *sourcefile Suffix* for valid file extensions). When specified, source code is included by #include directives and by Fortran INCLUDE lines. Generates *file*`.i`, which contains the source code after the preprocessing has been performed and the effects applied to the source program. For more information about source preprocessing, see *Source Preprocessing*. Default: disabled

-F

The `-F` option is obsolete and is supported for compatibility with legacy make files. Macro expansion is now enabled by default. To disable macro expansion, use the `-dF` option.

For more information about source preprocessing, see *Source Preprocessing*.

-f source_form

The `-f source_form` option specifies whether the Fortran source file is written in fixed source form or free source form. For *source_form*, enter free or fixed. The *source_form* specified here overrides any source form implied by the source file suffix. A FIXED or FREE directive specified in the source code overrides this option (see *FREE and FIXED*).

The default source form is fixed for input files that have the `.f`, `.F`, `.for`, or `.FOR` suffix. The default source form is free for input files that have the `.f90`, `.F90`, `.f95`, `.F95`, `.f03`, `.F03`, `.f08`, `.F08`, `.ftn`, or `.FTN` suffix. Note that the Fortran standard has declared fixed source form to be obsolescent.

If the file has a .F, `.FOR`, .F90, .F95, .F03, .F08, or `.FTN` suffix, the source preprocessor is invoked. See *Source Preprocessing* about preprocessing.

-h byteswapio

The `-h byteswapio` option forces byte-swapping from big-endian to little-endian (or visa versa) of all input and output files for direct and sequential unformatted I/O.

REAL(KIND=16) and COMPLEX(KIND=16) are not supported.

-h [no]contiguous

Default: `-h nocontiguous`

The `-hcontiguous` option declares that every assumed-shape array and array pointer target is contiguous, whether or not they have a CONTIGOUS keyword, potentially increasing the range of permitted compiler optimizations. By default, the compiler does not assume that all array pointers are pointers associated with contiguous targets or that all assumed shape arrays are contiguous and there is no way to verify this at compile time.

Use with caution. This additional level of compiler optimization is safe when the memory objects occupy contiguous blocks of memory. Also, if there is potential for hidden dependencies between the memory locations which the pointers are referring to, then do not use this option.

-h [no]contiguous_assumed_shape

Default: `nocontiguous_assumed_shape`

If `contiguous_assumed_shape` is specified, all assumed-shape dummy arguments are implicitly marked with the `CONTIGUOUS` attribute.

-h [no]fp_trap

Default: `fp_trap`, if traps are enabled using the `-K trap` option, or if `-Ofp[0,1]` is in effect. Otherwise, the default is `nofp_trap`.

Controls whether the compiler generates code that is compatible with floating-point traps.

-h [no]func_trace

Default: nofunc_trace

The `-h func_trace` option is for use only with CrayPat (Cray performance analysis tool). If this option is specified, the compiler inserts CrayPat entry points into each function in the compiled source file. The names of the entry points are:

```
__pat_tp_func_entry
__pat_tp_func_return
```

These are resolved by CrayPat when the program is instrumented using the `pat_build` command. When the instrumented program is executed and it encounters either of these entry points, CrayPat captures the address of the current function and its return address.

-h [no]second_underscore

Default: -h nosecond_underscore

The -h [no]second_underscore option controls the way in which external names are generated. By default, the compiler generates external names in lower case and will add one trailing underscore (_). This behavior matches the PGI Fortran compiler's external behavior. If -h second_underscore is specified, the compiler adds a second trailing underscore if the original external name has any underscores in it. This behavior matches the GNU compiler's external naming behavior.

-h page_align_allocate

The -h page_align_allocate option directs the compiler to force allocations of arrays larger than the memory page size to be aligned on a page boundary. This option affects only the ALLOCATE statements of the current source file; therefore it must be specified for each source file where this behavior is desired. Using this option can improve DIRECTIO performance.

-J *dir_name*

The -J dir_name option specifies the directory to which *file*.mod files are written when the -e m option is specified on the command line. By default, module files are written to the current working directory.

The compiler will automatically search the *dir_name* directory for modules to satisfy USE statements. An error is issued if the -em option is not specified when the -J dir_name is used.

-N *col*

The -N col option specifies the line width for fixed- and free-format source lines. The value used for *col* specifies the maximum number of columns per line.

For free form sources, *col* can be set to 132, 255, or 1023.

For fixed form sources, *col* can be set to 72, 80, 132, 255, or 1023.

Characters in columns beyond the *col* specification are ignored.

By default, lines are 72 characters wide for fixed-format sources and 255 characters wide for free-form sources.

-p module_site[,*module_site*]

The -p module_site option tells the compiler where to look for Fortran modules to satisfy USE statements. The *module_site* argument specifies the name of a file or directory to search for modules. The *module_site* specified can be a .mod file, .o (object) file, .a (archive) file, or a directory.

By default, module files are written to the current working directory. Alternatively, use the -J dir_name option during compilation to specify an alternate output directory for .mod files only. The compiler will search for modules stored in the directories specified using the -J dir_name option for the current compilation automatically; there is no need to use the -p option explicitly to make the compiler do this. For more information, see *-J dir_name*.

The search order for satisfying module references in USE statements is as follows:

1. The current working directory (or -J dir_name directory, if specified).
2. Any directories or files specified with the -p option.
3. Any directories with the -I option.
4. Any directories or files specified with the *FTN_MODULE_PATH* environment variable.

When searching within a directory, the compiler first searches the `.mod` files, then the `.o` files, then the `.a` files, and then the directories, in the order specified.

File name substitution (such as *.o) is not allowed. If the path name begins with a slash (/), the name is assumed to be an absolute path name. Otherwise, it is assumed to be a path name relative to the working directory. Specify multiple *`module_site`* locations with the `-p` option either by separating them with commas or by using a separate `-p` argument for each *`module_site`*. There is no limit on the number of `-p` options to be specified.

Cray provides some modules as part of the Cray Fortran Compiler Programming Environment. These are referred to as system modules. The system files containing these modules are searched last.

Consider the following command line:

```
% ftn -p steve.o -p mike.o joe.f
```

Assume that `steve.o` contains a module called Rock and `mike.o` contains a module called Stone. A reference to use Rock in `joe.f` causes the compiler to use Rock from `steve.o`. A reference to Stone in `joe.f` causes the compiler to use Stone from `mike.o`.

The following example specifies binary file `murphy.o` and library file `molly.a`:

```
% ftn -p murphy.o -p molly.a prog.f
```

In this example, assume that the following directory structure exists in the home directory:

```
       programs
      /    |    \
 tests   one.f   two.f
   |
use_it.f
```

The following module is in file `programs/one.f`, and the compiled version of it is in `programs/one.o`:

```
MODULE one
INTEGER i
END MODULE
```

The next module is in file programs/two.f, and the compiled version of it is in `programs/two.o`:

```
MODULE two
INTEGER j
END MODULE
```

The following program is in file `programs/tests/use_it.f`:

```
PROGRAM demo
USE one
USE two
. . .
END PROGRAM
```

To compile `use_it.f`, enter the following command from the home directory, which contians the subdirectory programs:

```
% ftn -p programs programs/tests/use_it.f
```

In the next set of program units, a module is contained within the first program unit and accessed by more than one program unit. The first file, `progone.f`, contains the following code:

```
MODULE split
INTEGER k
REAL a
END MODULE

PROGRAM demopr
USE split
INTEGER j
j = 3
k = 1
a = 2.0
CALL suba(j)
PRINT *, 'j=', j
PRINT *, 'k=', k
PRINT *, 'a=', a
END
```

The second file, `progtwo.f`, contains the following code:

```
SUBROUTINE suba(1)
USE split
INTEGER 1
1 = 4
k = 5
CALL subb (1)
RETURN
END

SUBROUTINE subb(m)
USE split
INTEGER m
m = 6
a = 7.0
RETURN
END
```

Use the following command line to compile the two files with one `ftn` command and a relative pathname:

```
% ftn -p progone.o progone.f progtwo.f
```

When the `-e m` option is in effect, use the `-p module_site` option to specify one or more directories that contain module files rather than specifying every individual module file name.

-Q *path*

The `-Q path` option specifies the directory that will contain all saved nontemporary files from this compilation (for example, all .o and .mod files). Specific file types (like .o files) are saved to a different directory if the -b, -J, -o, or -eS option is specified.

The following examples use this directory structure:

```
          current_dir
    ----------------------
    |         |          |
    |         |          |
    |         |          |
bin_out   mod_out    all_out
```

The following example saves all nontemporary files (x.o and any .mod files) in the current directory:

```
% ftn -b x.o -em x.f90
```

The following example saves all nontemporary files in the all_out directory and `x.o` in the current directory:

```
% ftn -Q all_out -em -b x.o x.f90
```

The following example saves the x.o file to the bin_out and all .mod files to the all_out directory:

```
% ftn -Q all_out -b bin_out/x.o -em x.f90
```

The following example saves the a.out file to the all_out and all .mod files to the mod_out directory:

```
% ftn -Q all_out -J mod_out x.f90
```

-s *size*

The `-s size` option allows the modification of the sizes of variables, literal constants, and intrinsic function results declared as type REAL, INTEGER, LOGICAL, COMPLEX, DOUBLE COMPLEX, or DOUBLE PRECISION. Use one of the following values for *size*:

size	Action
`byte_pointer`	(Default) Applies a byte scaling factor to integers used in pointer arithmetic involving Cray pointers. That is, Cray pointers are moved on byte instead of word boundaries. Pointer arithmetic scaling is explained in *Pointer Scaling Factor*.
`default32`	(Default) Adjusts the data size of default types as follows: • 32 bits: REAL, INTEGER, LOGICAL • 64 bits: COMPLEX, DOUBLE PRECISION • 128 bits: DOUBLE COMPLEX

size	**Action**
	The data sizes of integers and logicals that use explicit kind and star values are not affected by this option. However, they are affected by the `-e h` option. See *-d disable_opt and -e enable_opt*
`default64`	Adjust the data size of default types as follows: ● 64 bits: REAL, INTEGER, LOGICAL ● 64 bits: DOUBLE PRECISION (implied -dp) ● 128 bits: COMPLEX ● 128 bits: DOUBLE COMPLEX (implied -dp) If the -s default64 is used at compile time, specify this option when invoking the `ftn` command. The data sizes of integers and logicals that use explicit kind and star values are not affected by this option. However, they are affected by the `-dh` option. See *-d disable_opt and -e enable_opt*
`integer32`	(Default) Adjusts the default data size of default integers and logicals to 32 bits.
`integer64`	Adjusts the default data size of default integers and logicals to 64 bits.
`real32`	(Default) Adjusts the default data size of default real types as follows: ● 32 bits: REAL ● 64 bits: COMPLEX and DOUBLE PRECISION ● 128 bits: DOUBLE COMPLEX
`real64`	Adjusts the default data size of default real types as follows: ● 64 bits: REAL ● 64 bits: DOUBLE PRECISION (implied -dp) ● 128 bits: COMPLEX ● 128 bits: DOUBLE COMPLEX (implied -dp)
`word_pointer`	Applies a word scaling factor to integers used in pointer arithmetic involving Cray pointers. That is, Cray pointers are moved on word instead of byte boundaries. Pointer arithmetic scaling is explained later in *Pointer Scaling Factor*.

The default data size options (for example, `-s default64`) option do not affect the size of explicitly declared data (for example, `REAL(KIND=4) X`).

`REAL(KIND=16)` and `COMPLEX(KIND=16)` support 128-bit floating point and 256-bit complex types, sometimes referred to as quad-precision. See *128-Bit Precision*.

Different Default Data Size Options on the Command Line

Be careful when mixing different default data size options on the same command line because equivalencing data of one default size with data of another default size can cause unexpected results. For example, assume that the following command line is used for a program:

```
% ftn -s default64 -s integer32 ...
```

The mixture of these default size options causes the program below to equivalence 32-bit integer data with 64-bit real data and to incompletely clear the real array.

```
Program test
   IMPLICIT NONE

   real r
   integer i
   common /blk/ r(10), i(10)
   integer overlay(10)

   equivalence (overlay, r)

   call clear(overlay)
   call clear(i)

contains
subroutine clear(i)
   integer, dimension (10) :: i

   i = 0
end subroutine

end program test
```

The above program sets only the first 10 32-bit words of array `r` to zero. It should instead set 10 64-bit words to zero.

Pointer Scaling Factor

```
Cray_ptr = Cray_ptr + integer_value
```

as

```
Cray_ptr = Cray_ptr + (integer_value * scaling_factor)
```

The scaling factor is dependent on the size of the default integer and which scaling option (`-s byte_pointer` or `-s word_pointer`) is enabled.

Table 6. Scaling Factor in Pointer Arithmetic

Scaling Options	Default Integer Size	Scaling Factor
`-s byte_pointer`	32 or 64 bits	1
`-s word_pointer` and `-s default32` enabled	32 bits	4
`-s word_pointer` and `-s default64` enabled	64 bits	8

Therefore, when the `-s byte_pointer` option is enabled, this example increments ptr by i bytes:

```
pointer (ptr, ptee)   !Cray pointer

ptr = ptr + i
```

When the `-s word_pointer` and `-s default32` options are enabled, the same example is viewed by the compiler as:

```
ptr = ptr + (4*i)
```

When the `-s word_pointer` and `-s default64` options are enabled, the same example is viewed by the compiler as:

```
ptr = ptr + (8*i)
```

`-T`

The `-T` option disables the compiler but displays all options currently in effect. The Cray Fortran compiler generates information identical to that generated when the `-v` option is specified on the command line; when `-T` is specified, however, no processing is performed. When this option is specified, output is written to the standard error file (`stderr`).

4.11 Performance Tool Options

`-h [no]func_trace`

Default: `nofunc_trace`

The `-h func_trace` option is for use only with CrayPat (Cray performance analysis tool). If this option is specified, the compiler inserts CrayPat entry points into each function in the compiled source file. The names of the entry points are:

```
__pat_tp_func_entry
__pat_tp_func_return
```

These are resolved by CrayPat when the program is instrumented using the `pat_build` command. When the instrumented program is executed and it encounters either of these entry points, CrayPat captures the address of the current function and its return address.

-h keepfiles

The `-h keepfiles` option prevents the removal of the object (`.o`) and temporary assembly (`.s`) files after an executable is created. Normally, the compiler automatically removes these files after linking them to create an executable. Use this option to preserve the object files required by CrayPat to instrument a program for performance analysis experiments.

-h keep_frame_pointer

Default: off

Retain call stack information back to main entry point for CrayPat performance sampling. Prevents call stack frame from being optimized out of a function so CrayPat performance sampling is able to trace call stack back to entry point.

-h [no]omp_trace

Default: `-h noomp_trace`

The `-h omp_trace` option turns the insertion of the CrayPat OpenMP tracing calls on or off.

4.12 Preprocess Options

-D *identifier=value*

The `-D` *`identifier=value`* option defines variables used for source preprocessing as if they had been defined by a #define source preprocessing directive. If a *`value`* is specified, there can be no spaces on either side of the equal sign (+). If no *`value`* is specified, the default value of 1 is used.

The `-U` option undefines variables used for source preprocessing. If both `-D` and `-U` are used for the same *`identifier`*, in any order, the *`identifier`* is undefined.

This option is ignored unless one of the following conditions is true:

- The Fortran input source file is specified as either *`file`*`.F`, *`file`*`.F90`, *`file`*`.F95`, *`file`*`.F03`, *`file`*`.F08`, or *`file`*`.FTN`.
- The -eP or -eZ options have been specified.

  For more information about source preprocessing, see *Source Preprocessing*.

-I *incldir*

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

1. Directory of the input file. Note that the input file is not always in the current working directory. The input file is searched, not the current working directory, if they differ. If the user wants the current working directory searched, the `-I` option should be added to the command line.

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

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.

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

The `-I` *`incldir`* option specifies a directory to be searched for files named in `INCLUDE` lines in the Fortran source file and for files named in `#include` source preprocessing directives. Additionally, all user-specified `-I` *`incldir`* directories are searched for `MODULE USE` resolution after all user-specified `-p` *`paths`* are searched.

An `-I` option must be specified for each directory to be searched. Directories can be specified in incldir as full path names or as path names relative to the working directory. By default, only the directory of the file referencing the included file and system directories are searched. None of the system-specified `-I` *`incldir`* directories are searched during `MODULE USE` resolution.

The following example causes the compiler to search for files included within `earth.f` in the directories `/usr/local/dir and ../moon`:

```
% ftn -I /usr/local/dir -I ../moon earth.f
```

If the `INCLUDE` line or `#include` directive in the source file specifies an absolute name (that is, one that begins with a slash (/)), that name is used, and no other directory is searched. If a relative name is used (that is, one that does not begin with a slash (/)), the compiler searches for the file in the directory of the source file containing the `INCLUDE` line or `#include` directive. If this directory contains no file of that name, the compiler then searches the directories named by the `-I` options, as specified on the command line, from left to right.

`-U identifier`,*`identifier`*…

The `-U identifier` ,*`identifier`* ... option undefines variables used for source preprocessing. This option removes the initial definition of a predefined macro or sets a user predefined macro to an undefined state.

The `-D identifier` =*`value`* option defines variables used for source preprocessing. If both `-D` and `-U` are used for the same *`identifier`*, in any order, the *`identifier`* is undefined.

This option is ignored unless one of the following conditions is true:

- The Fortran input source file is specified as either *`file`*`.F`, *`file`*`.F90`, *`file`*`.F95`, *`file`*`.F03`, *`file`*`.F08`, or *`file`*`.FTN`.
- The `-e P` or `-e Z` options have been specified.

For more information about source preprocessing, see *Source Preprocessing*.

`-x dirlist`

The `-x dirlist` option disables specified directives or specified classes of directives. If specifying a multiword directive, either enclose the directive name in quotation marks or remove the spaces between the words in the directive's name.

For *`dirlist`*, enter one of the following arguments:

dirlist	Item disabled
all	All compiler directives and OpenMP Fortran directives. For information about the OpenMP, see *OpenMP Overview*.
dir	All compiler directives.
directive	One or more compiler directives or OpenMP Fortran directives. If specifying more than one, separate them with commas; for example: -x INLINEALWAYS,"NO SIDE EFFECTS",BOUNDS.
omp	All OpenMP Fortran directives
acc	All OpenACC Fortran directives.
conditional_omp	All C$ and !$ conditional compilation lines.

4.13 Program Model Specific Options

-h [no]acc

Default: `-h acc`

Enables or disables compiler recognition of OpenACC directives (`!$acc` sentinel).

-h [no]caf

Default: `-h caf`

Enable the compiler to recognize coarray syntax. The macro `_CRAY_COARRAY` will be defined as 1 if `-hcaf` is specified on the command line.

Note that `-hnocaf` is required for Fortran code when it will be linked with C++ code.

Note that On Cray Cluster Systems (CCS), the default is `nocaf`. If using the `-h caf` option, the `-L <path_to_pmi>` option must also be used. The `path_to_pmi` argument should be the location where `libpmi.so` is located on the system. For example, if a site is using slurm and the slurm install base is located at `/global/opt/slurm/default`, the the following link option should be provided: `-L /global/opt/slurm/default/lib64`.

-h [no]omp, -O [no]omp

Default: `-h omp` if `-O1` or higher is implied or specified.

Enables or disables compiler recognition of OpenMP directives. Using the `-h [no]omp` option is similar to the `-h thread0` option, in that it disables OpenMP, but unlike `-h thread0`, it does not affect autothreading. The `-h [no]omp` option is identical to the `-O [no]omp` option and is provided for command-line compatibility with the Cray C compiler. If `-O0` is implied or specified then `-h noomp` is implied.

4.14 Scalar Optimization Options

-h [no]interchange, -O [no]interchange

Default: `interchange`

This option allows the compiler to attempt to interchange all loops - a technique that is used to improve 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.

To disable interchange of loops individually, use the `nointerchange` directive prior to the loop.

`nointerchange` inhibits the compiler's attempts to interchange loops.

Specifying the `nointerchange` option is equivalent to specifying a `NOINTERCHANGE` directive prior to every loop. To disable loop interchange on individual loops, use the `NOINTERCHANGE` directive.

-h scalar*n*, -O scalar*n*

Default: `scalar2`

Specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option. The values for n are:

0 Minimal automatic scalar optimization. Implies `-h zeroinc`.

1 Conservative automatic scalar optimization. Implies `-h nozeroinc`.

2 Aggressive automatic scalar optimization. The scalar optimizations that provide the best application performance are used, with some limitations imposed to allow for faster compilation times.

3 Very aggressive optimization; compilation times may increase significantly.

-h [no]zeroinc , -O [no]zeroinc

Default: `-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.

It is identical to the `-O [no]zeroinc` option.

The `-h zeroinc` option causes the compiler to assume that some constant increment variables (CIVs) in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code.

`-h zeroinc` can cause less strength reduction to occur in loops that have variable increments.

A CIV is a variable that is incremented only by a loop invariant value. For example, in a loop with variable *J*, the statement `J = J + K`, where *K* can be equal to zero, *J* is a CIV.

4.15 Target Options

-h cpu=*target_system*

Specifies the Cray system on which the absolute binary file is to be executed, where *target_system* can be either x86-64, sandybridge, ivybridge, haswell, broadwell, mic-knl, x86-skylake, and arm-thunderx2.

Rather than setting this option directly, users should load one of the targeting modules (craype-x86-skylake, for example). The targeting modules set CRAY_CPU_TARGET and define paths to the corresponding libraries. The compiler driver script translates CRAY_CPU_TARGET to the corresponding cpu= option when calling the compiler.

If the target_system is set during compilation of any source file, it must also be set to that same target during linking and loading. If a user wishes to override the current target_system value set by the module environment (via the CRAY_CPU_TARGET definition), they should do so by specifying -hcpu=*target_system* on the compiler command line.

-h network=*nic*

Specifies the target machine's system interconnection network. Currently, supported values for *nic* are aries, infiniband, and opa.

Rather than setting this option directly, users should load one of the network modules (craype-network-aries for example). The network modules cause the corresponding compiler option to be set and define paths to the corresponding libraries.

4.16 Vector Optimization Options

-h concurrent

The -h concurrent option indicates that no data dependence exists between array references of the same loop, for every loop in the file. This can be useful for vectorization optimizations. Equivalent to adding a CONCURRENT directives before every loop in the file, including loops created from array syntax.

-h vector*n*, -O vector*n*

Default: vector2

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

0 No automatic vectorization. Characteristics include low compile time and small compile size. This option is compatible with all scalar optimization levels.

1 Specifies conservative vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when -h vector0 is specified because of vector reductions. The -h vector1 option is compatible with -h scalar1, -h scalar2, and -h scalar3.

2 Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. Compatible with `-h scalar2` and `-h scalar3`.

3 Specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed.

-h [no]preferred_vector_width[=[64 | 128 | 256 | 512]]

Specifies the preferred vector width to use when vectorizing loops. This option does not guarantee that the specified vector width will be used, only that it is preferred. The optimizer is still free to choose a smaller width if it is expected to perform better. As the set of acceptable width is target-sensitive and fairly complicated, the optimizer diagnoses any illegal values.

A value is not required when specifying `nopreferred_vector_width`.

This argument can also be passed in to the `OPTIMIZE` directive, as described in *OPTIMIZE*.

5 Set Environment Variables to the Cray Fortran Compiler

Environment variables are predefined shell variables, taken from the execution environment, that determine some of the shell characteristics. Several environment variables pertain to the Cray Fortran compiler. The Cray Fortran compiler recognizes general and multiprocessing environment variables.

The multiprocessing variables in the following sections affect the way the program will perform on multiple processors. Use environment variables to tune the system for parallel processing without rebuilding libraries or other system software.

The variables allow for controlling parallel processing at compile time and at run time. Compile time environment variables apply to all compilations in a session.

The following examples show how to set an environment variable:

- With the standard shell, enter:

```
CRAY_FTN_OPTIONS=options
export CRAY_FTN_OPTIONS
```

- With the C shell, enter:

```
setenv CRAY_FTN_OPTIONS options
```

The following sections describe the environment variables recognized by the Cray Fortran compiler.

Many of the environment variables described in this chapter refer to the default system locations of Programming Environment components. If the Cray Fortran Compiler Programming Environment has been installed in a non-default location, see the system support staff for path information.

CRAY_FTN_OPTIONS

The *CRAY_FTN_OPTIONS* environment variable specifies additional options to attach to the command line. This option follows the options specified directly on the command line. File names cannot appear. These options are inserted at the rightmost portion of the command line before the input files and binary files are listed. This allows the environment variable to be set once and have the specified set of options used in all compilations. This is especially useful for adding options to compilations done with build tools.

For example, assume that this environment variable was set as follows:

```
setenv CRAY_FTN_OPTIONS -G0
```

With the variable set, the following two command line specifications are equivalent:

```
% ftn  -c t.f
% ftn -c -G0 t.f
```

FORTRAN_MODULE_PATH

Like the Cray Fortran compiler -p module_site command line option, this environment variable allows for the specification of the files or the directory to search for the modules to use. The files can be archive files, build files (bld file), or binary files.

The compiler appends the paths specified by the *FORTRAN_MODULE_PATH* environment variable to the path specified by the -p module_site command line option.

Since the *FORTRAN_MODULE_PATH* environment variable can specify multiple files and directories, a colon separates each path as shown in the following example:

```
% set FORTRAN_MODULE_PATH='path1 : path2 : path3'
```

LISTIO_PRECISION

The *LISTIO_PRECISION* environment variable controls the number of digits of precision printed by list-directed output. The *LISTIO_PRECISION* environment variable can be set to FULL or PRECISION.

- FULL prints full precision (default).
- PRECISION prints *x* or *x* + 1 decimal digits, where *x* is the value of the PRECISION intrinsic function for a given real value. This is a smaller number of digits, which usually ensures that the last decimal digit is accurate to within 1 unit. This number of digits is usually insufficient to assure that subsequent input will restore a bit-identical floating-point value.

NLSPATH

The NLSPATH environment variable specifies the message system library catalog path. This environment variable affects compiler interactions with the message system. For more information about this environment variable, see the `catopen(3)` man page.

NPROC

The NPROC environment variable specifies the maximum number of processes to be run. Setting NPROC to a number other than 1 can speed up a compilation if machine resources permit.

The effect of NPROC is seen at compilation time, not at execution time. NPROC requests a number of compilations to be done in parallel. It affects all the compilers and also the `make` command.

For example, assume that NPROC is set as follows:

```
setenv NPROC 2
```

The following command is entered:

```
ftn -o t main.f sub.f
```

In this example, the compilations from .f files to .o files for `main.f` and `sub.f` happen in parallel, and when both are done, the link step is performed. If NPROC is unset, or set to 1, main.f is compiled to main.o; sub.f is compiled to sub.o, and then the link step is performed.

The NPROC can be set to any value, but large values can overload the system. For debugging purposes, NPROC should be set to 1. By default, NPROC is 1.

ZERO_WIDTH_PRECISION

The *ZERO_WIDTH_PRECISION* environment variable controls the field width when field width *w* of F*w*.d is zero on output. The *ZERO_WIDTH_PRECISION* environment variable can be set to PRECISION or HALF.

- PRECISION specifies that full precision will be written. This is the default.
- HALF specifies that half of the full precision will be written.

Run Time Environment Variables

Run time environment variables allow for adjusting the following elements of the run time environment:

- CRAY_ACC_DEBUG
  - Write accelerator-related activity to stdout for debugging purposes. Valid output levels range from 0, which indicates no output, through 3, which indicates verbose.
  - Default: 0
- CRAY_AUTO_APRUN_OPTIONS
  - Default options for automatic `aprun`. See the `aprun(1)` man page.'
- CRAY_RANK_THREAD_PREFIX
  - Prepend a string identifying mpi rank and omp thread id to each line written to `stdout` and `stderr`.
- CRAY_MALLOPT_OFF (Only relevant if -hsystem_alloc is specified)
  - If set, then the system default mallopt parameters are used, instead of the compiler default parameters. For most programs, run time performance is improved by using the compiler defaults, but more memory may be used.
- FORMAT_TYPE_CHECKING
  - The *FORMAT_TYPE_CHECKING* environment variable specifies various levels of conformance between the data type of each I/O list item and the formatted data edit descriptor.
  - When set to RELAXED, the run time I/O library enforces limited conformance between the data type of each I/O list item and the formatted data edit descriptor.
  - When set to STRICT77, the run time I/O library enforces strict FORTRAN 77 conformance between the data type of each I/O list item and the formatted data edit descriptor.
  - When set to STRICT90 or STRICT95, the run time I/O library enforces strict Fortran 90/95 conformance between the data type of each I/O list item and the formatted data edit descriptor.
  - See the following tables:
    - *Default Compatibility Between I/O List Data Types and Data Edit Descriptors*
    - *RELAXED Compatibility Between Data Types and Data Edit Descriptors*
    - *STRICT77 Compatibility Between Data Types and Data Edit Descriptors*
    - *STRICT90 or STRICT95 Compatibility Between Data Types and Data Edit Descriptors*
- MALLOC_MMAP_MAX_ (Only relevant if -hsystem_alloc is specified)
  - Specifies the maximum number of memory chunks to allocate with mmap. The compiler default value is 0. For most programs, run time performance is improved by using the compiler default, but more memory may be used.

- MALLOC_TRIM_THRESHOLD_ (Only relevant if -hsystem_alloc is specified)
  - Specifies the minimum size of the unused memory region at the top of the heap before the region is returned to the operating system. The compiler default value is 536870912 bytes. For most programs, run time performance is improved by using the compiler default, but more memory may be used.
- NO_STOP_MESSAGE
  - If set, and if the STOP *stop_code* statement in the Fortran code does not specify the optional *stop_code*, then STOP messages are not produced when this statement is executed.
- PGAS_ERROR_FILE
  - Specifies the location to which `libpgas` (the library which provides an interface to the internal system network) error messages are written. The default is `stderr`. If `stdout` is specified, errors will be written to standard output.
- TMPDIR
  - Compiler temporary files and user scratch files are placed in the directory specified by the *TMPDIR* environment variable.
- CRAYLIBS_ARCH_OVERRIDE
  - Override the default Cray math library run time selection and specify the library to use by CPU architecture. The valid options are: `haswell`, `ivybridge`, `sandybridge`, `x86-64`, or `mic-knl`.

    Can be used to specify that a lowest-common-denominator math library be used instead of the default selection, thus ensuring that identical computations produce identical results regardless of the type of compute node CPU actually used. The trade-off is that specifying an older library may affect performance on a newer CPU. For example, if `ivybridge` is specified, the code will run and produce identical results on a `haswell` compute node, but performance may be reduced.

    Default: If not set, the library specific to the type of CPU selected at run time is used.

`aprun` Resource Limits

The `aprun` command always forwards its own core and cpu resource limits (`RLIMIT_CPU` and `RLIMIT_CORE`) to the compute nodes where those limits are set for the application. If a `-m` value is specified, `RLIMIT_RSS` is also forwarded.

If the `APRUN_XFER_LIMITS` run time environment variable is set to a non-zero value, the following resource limits are also forwarded:

- `RLIMIT_FSIZE`
- `RLIMIT_DATA`
- `RLIMIT_STACK`
- `RLIMIT_RSS`
- `RLIMIT_NPROC`
- `RLIMIT_NOFILE`
- `RLIMIT_MEMLOCK`
- `RLIMIT_AS`
- `RLIMIT_LOCKS`

- `RLIMIT_SIGPENDING`
- `RLIMIT_MSGQUEUE`
- `RLIMIT_NICE`
- `RLIMIT_RTPRIO`

This forwarding is diasbled by default.

This forwarding of user resource limits can cause problems on systems where the login node's limits are more restrictive than the default compute node limits.

6 Cray Fortran Directive Use

Directives are lines inserted into source code that specify actions to be performed by the compiler. They are not Fortran statements.

If a directive is specified while running on a system that does not support that particular directive, the compiler generates a message and continues with the compilation.

Directive Lines

A directive line begins with the characters CDIR$ or !DIR$, unless otherwise noted. These leading characters are sometimes referred to as a sentinel.

OpenMP directives are indicated using the !OMP$ sentinel. OpenACC directives are indicated with the !ACC$ sentinel.

How to specify directives depends on the source form being used, as follows:

- If using fixed source form, indicate a directive line by placing the characters CDIR$ or !DIR$ in columns 1 through 5. If the compiler encounters a nonblank character in column 6, the line is assumed to be a directive continuation line. Columns 7 and beyond can contain one or more directives. Characters in directives entered in columns beyond the default column width are ignored.
- If using free source form, indicate a directive by the characters !DIR$, followed by a space, and then one or more directives. If the position following the !DIR$ contains a character other than a blank, tab, or newline character, the line is assumed to be a continuation line. The !DIR$ need not start in column 1, but it must be the first text on a line.

  In the following example, an asterisk (*) appears in column 6 to indicate that the second line is a continuation of the preceding line:

  ```
  !DIR$ NOSIDEEFFECTS
  !DIR$*ab
  ```

  The `FIXED` and `FREE` directives must appear alone on a directive line and cannot be continued.

  To specify more than one directive on a line, separate each directive with a comma. Some directives require specification of one or more arguments; when specifying a directive of this type, no other directive can appear on the line.

  Spaces can precede, follow, or be embedded within a directive, regardless of source form.

  Code portability is maintained despite the use of directives. In the following example, the ! symbol in column 1 causes other compilers to treat the Cray Fortran compiler directive as a comment:

  ```
        A=10.
  !DIR$ NOVECTOR
        DO 10,I=1,10...
  ```

Do not use source preprocessor (#) directives within multiline compiler directives (CDIR$ or !DIR$).

6.1 Inline and Clone Directives

inline

```
!DIR$ INLINE
!DIR$ NOINLINE
!DIR$ RESETINLINE
!DIR$ INLINEALWAYS [name [,name] ...]
!DIR$ INLINENEVER [name [,name] ...]
```

Inlining replaces calls to user-defined functions with the code that represents the function. This can improve performance by saving the expense of the function call overhead. It also increases the possibility of additional code optimization. Inlining may increase object code size.

Enabling inlining instructs the compiler to attempt to inline functions at call sites. Resetting inlining returns the inlining state to the state specified on the command line (`-h ipa`*n* or `-O ipa`*n*).

The enabling/disabling directives remain in effect until the opposite directive is encountered, or until a reset directive is encountered, or until the end of the program unit.

The `always` directive specifies functions that the compiler should always attempt to inline. If the directive is placed in the definition of the function, inlining is attempted at every call site to that function in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is attempted at every call site to *name* within the specific function containing the directive. The `never` directive specifies functions that should not be inlined. If the directive is placed in the definition of the function, inlining is never attempted at any call site to that function in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is never attempted at any call site to *name* within the specific function containing the directive. An error message is issued if both never and always are specified for the same procedure in the same program unit.

INLINEALWAYS, INLINENEVER Directive

```
SUBROUTINE S()
!DIR$ INLINEALWAYS S ! THIS SAYS ATTEMPT
! INLINING OF S AT ALL CALLS.
...
END SUBROUTINE
SUBROUTINE T
!DIR$ INLINENEVER S ! DO NOT INLINE ANY CALLS TO S
! IN SUBROUTINE T.
CALL S()
...
END SUBROUTINE
SUBROUTINE V
!DIR$ NOINLINE ! HAS HIGHER PRECEDENCE THAN INLINEALWAYS.
CALL S() ! DO NOT INLINE THIS CALL TO S.
!DIR$ INLINE
CALL S() ! ATTEMPT INLINING OF THIS CALL TO S.
...
END SUBROUTINE
```

```
SUBROUTINE W
CALL S() ! ATTEMPT INLINING OF THIS CALL TO S.
...
END SUBROUTINE
```

CLONE

```
!DIR$ CLONE
!DIR$ NOCLONE
!DIR$ RESETCLONE
!DIR$ CLONEALWAYS [name [,name] ...]
!DIR$ CLONENEVER [name [,name] ...]
```

Cloning is the creation of a duplicate routine. The compiler creates a clone when it detects characteristics of the call that will likely lead to a better optimized routine. The original call site is replace with a call to the newly cloned routine. This includes things like constant actual arguments, the passing of contiguous arrays when the compiler cannot so determine from the callee, non-present OPTIONAL dummy arguments, etc.

`NOCLONE` disables all cloning. `CLONE` forces cloning if a condition exists to clone.

`RESETCLONE` returns the cloning to the state specified on the compiler command line. The cloning directives remain in effect until a different cloning directive is encountered or until the end of the program unit. Use `-hnegmsgs` to see messages that highlight where cloning did occur and conditions that may have inhibited cloning. Use the `CLONEALWAYS` and `CLONENEVER` directives to control cloning of a specific procedure. If the directive is placed in the definition of the function, cloning is always/never attempted at every call site to name. If the directive is placed in a function other than the definition, cloning is always/never attempted at every call to name within the specific function containing the directive.

CLONENEVER and CLONEALWAYS

```
SUBROUTINE S()
!DIR$ CLONEALWAYS S ! ATTEMPT CLONING OF S AT ALL CALLS.
...
END SUBROUTINE

SUBROUTINE T
!DIR$ CLONENEVER S ! DO NOT CLONE ANY CALLS TO S IN SUBROUTINE T.
CALL S()
...
END SUBROUTINE

SUBROUTINE V
!DIR$ NOCLONE ! HIGHER PRECEDENCE THAN CLONEALWAYS.
CALL S()      ! DO NOT CLONE THIS CALL TO S.
!DIR$ CLONE
CALL S()      ! ATTEMPT CLONING OF THIS CALL TO S.
...
END SUBROUTINE
```

[NO]MODINLINE

```
!DIR$ MODINLINE
!DIR$ NOMODINLINE
```

The MODINLINE and NOMODINLINE directives enable and disable the creation of inlinable templates for specific module procedures.

The MODINLINE and NOMODINLINE directives are ignored if -O nomodinline is specified on the ftn command line.

These directives are in effect for the scope of the program unit in which they are specified, including all contained procedures. If one of these directives is specified in a contained procedure, the contained procedure's directive overrides the containing procedure's directive.

The compiler generates a message if these directives are specified outside of a module and ignores the directive.

```
      MODULE BEGIN
      ...
      CONTAINS
        SUBROUTINE S()           ! Uses SUBROUTINE S's !DIR$
!DIR$   NOMODINLINE
        ...
        CONTAINS
          SUBROUTINE INSIDE_S()  ! Uses SUBROUTINE S's !DIR$
          ...
          END SUBROUTINE INSIDE_S
        END SUBROUTINE S
        SUBROUTINE T()           ! Uses MODULE BEGIN's !DIR$
        ...
        CONTAINS
          SUBROUTINE INSIDE_T()  ! Uses MODULE BEGIN's !DIR$
          ...
          END SUBROUTINE INSIDE_T
          SUBROUTINE MORE_INSIDE_T
!DIR$     NOMODINLINE
          ...
          END SUBROUTINE MORE_INSIDE_T
        END SUBROUTINE T
      END MODULE BEGIN
```

In the preceding example, the subroutines are affected as follows:

- Inlining templates are not produced for S, INSIDE_S, or MORE_INSIDE_T.
- Inlining templates are produced for T and INSIDE_T.

6.2 Local Control Directives

The following directives provide local control over specific compiler features. The -f and -R command line options apply to an entire compilation, but these directives override any command line specifications for source form or bounds checking.

[NO]BOUNDS

```
!DIR$ BOUNDS [ array [, array ] ... ]
!DIR$ NOBOUNDS [ array [, array ] ... ]
```

array The name of an array. The name cannot be a subobject of a derived type. When no array name is specified, the directive applies to all arrays.

Array bounds checking provides a check of most array references at both compile time and run time to ensure that each subscript is within the array's declared size. Bounds checking behavior differs with the optimization level. Bounds checking is not performed on arrays dimensioned as 1. Enables `-Ooverindex`. Complete checking is guaranteed only when optimization is turned off by specifying `-O 0` on the `ftn` command line.

The `-R` command line option controls bounds checking for a whole compilation. The `BOUNDS` and `NOBOUNDS` directives toggle the feature on and off within a program unit. Either directive can specify particular arrays or can apply to all arrays.

Equivalent to the `-Rb` option and added to improve C/Fortran command line compatibility.

`BOUNDS` remains in effect for a given array until the appearance of a `NOBOUNDS` directive that applies to that array, or until the end of the program unit. Bounds checking can be enabled and disabled many times in a single program unit.

To be effective, these directives must follow the declarations for all affected arrays. It is suggested that they be placed at the end of a program unit's specification statements unless they are meant to control particular ranges of code.

The bounds checking feature detects any reference to an array element whose subscript exceeds the array's declared size.

```
    REAL A(10)
C DETECTED AT COMPILE TIME:
     A(11) = X
C DETECTED AT RUN TIME IF IFUN(M) EXCEEDS 10:
     A(IFUN(M)) = W
```

The compiler generates an error message if it detects that an array element section reference with an out-of-bounds subscript attempts to reference memory. If the compiler cannot detect the out-of-bounds subscript (for example, if the subscript includes a function reference), a message is issued for out-of-bound subscripts when the program runs, but the program is allowed to complete execution.

Bounds checking does not inhibit vectorization but typically increases program run time. If an array's last dimension declarator is *, checking is not performed on the last dimension's upper bound. Arrays in formatted `WRITE` and `READ` statements are not checked.

Array bounds checking does not prevent operand range errors that result when operand prefetching attempts to access an invalid address outside an array. Bounds checking is needed when very large values are used to calculate addresses for memory references.

If bounds checking detects an out-of-bounds array reference, a message is issued for only the first out-of-bounds array reference in the loop.

```
DIMENSION A(10)
      MAX = 20
      A(MAX) = 2
      DO 10 I = 1, MAX
         A(I) = I
10    CONTINUE
      CALL TWO(MAX,A)
```

```
      END
      SUBROUTINE TWO(MAX,A)
      REAL A(*)  ! NO UPPER BOUNDS CHECKING DONE
      END
```

The following messages are issued for the preceding program:

```
lib-1961 a.out: WARNING
  Subscript 20 is out of range for dimension 1 for array
  'A' at line 3 in file 't.f' with bounds 1:10.

lib-1962 a.out: WARNING
  Subscript 1:20:1 is out of range for dimension 1 for array
  'A' at line 5 in file 't.f' with bounds 1:10.
```

FREE and FIXED

```
!DIR$ FREE
!DIR$ FIXED
```

The FREE and FIXED directives specify whether the source code in the program unit is written in free source form or fixed source form. The FREE and FIXED directives override the -f option, if specified, on the command line.

These directives apply to the source file in which they appear, and they allow for switching source forms within a source file.

Source form can be changed from within an INCLUDE file. After the INCLUDE file has been processed, the source form reverts back to the source form that was being used prior to processing of the INCLUDE file.

OPTIMIZE

```
!DIR$ OPTIMIZE (option option)
```

The OPTIMIZE directive enables/disables optimization in the program unit in which it appears, overriding the optimization level set via the compiler command line. This directive may only appear in the declarative section of a program unit. A program unit may be a program, subroutine, function, module, or submodule, but not block data program unit.

OPTIMIZE does not affect any modules invoked with the USE statement in the program unit that contains them. They do affect CONTAINed procedures that do not include an explicit OPTIMIZE directive.

The OPTIMIZE directive with no *option* specified is equivalent to OPTIMIZE -O2.

The OPTIMIZE directive accepts the following subset of the command line options which control optimization. Refer to crayftn(1) for a description of these options.

The OPTIMIZE directive accepts the following subset of the command line options which control optimization. Refer to crayftn(1) for a description of these options.

- -O level
- -h acc
- -h acc_model=
- -h add_paren
- -h [no]aggress
- -h align_arrays

- `-h [no]autothread`
- `-h [no]autoprefetch`
- `-h cache` *n*
- `-h concurrent`
- `-h contiguous`
- `-h contiguous_assumed_shape`
- `-h flex_mp=`*level*
- `-h fp` *n*
- `-h fp_trap`
- `-h fusion` *n*
- `-h infinitevl`
- `-h loop_trips`
- `-h msgs`
- `-h negmsgs`
- `-h nointerchange`
- `-h omp`
- `-h overindex`
- `-h page_align_allocate`
- `-h [no]pattern`
- `-h scalar` *n*
- `-h shortcircuit` *level*
- `-h thread` *n*
- `-h unroll` *n*
- `-h vector` *n*
- `-h zero`

See the `OPTIMIZE(7)` man page.

6.3 Miscellaneous Directives

[NO]AUTOTHREAD

```
!DIR$ AUTOTHREAD
!DIR$ NOAUTOTHREAD
```

The `[no]autothread` directive turns autothreading on or off for selected blocks of code.

These directives are ignored if the `-h thread0` or `-O thread0` options are used.

CACHE

```
!DIR$ CACHE base_name[, base_name]
```

base_name The base name of the object that should be placed into the cache. If a pointer in the list is specified, only the references and not the pointer itself are cached.

The `cache` directive asserts that all memory operations with the specified symbols as the base are to be allocated in cache. This is an advisory directive. The cache directive is meaningful for stores in that it allows the user to override a decision made by the automatic cache management. This directive may be locally overridden by the use of a `LOOP_INFO` directive. This directive overrides automatic cache management decisions. To use the directive, place it only in the specification part, before any executable statement.

CACHE_NT

```
!DIR$ CACHE_NT base_name [,base_name ...]
```

base_name The base name of the object that should use non-temporal reads and writes. This can be the base name of any object such as an array, scalar structure, and so on, without member references. If a pointer in the list is specified, only the references, not the pointer itself, have the cache non-temporal property.

Use this directive to identify objects that should not be placed in cache. This is an advisory directive that specifies objects that should use non-temporal reads and writes.

This directive overrides the automatic cache management level that was specified using the `-O cache`*n* option on the compiler command line. This directive may be overridden locally by use of a `LOOP_INFO` directive.

[NO]FUSION

```
!DIR$ FUSION
!DIR$ NOFUSION
```

Scope: Local

The `nofusion` directive instructs the compiler to not attempt loop fusion on the following loop even when the -h fusion option was specified on the compiler command line. The fusion directive instructs the compiler to attempt loop fusion on the following loop unless `-h nofusion` was specified on the compiler command line.

ID

```
!DIR$ ID "character_string"
```

The ID directive inserts a character string into the *`file`*`.o` produced for a Fortran source file.

`character_string`

The character string to be inserted into *`file`*`.o`. The syntax box shows quotation marks as the *`character_string`* delimiter, but either apostrophes (' ') or quotation marks (" ") can be used.

The *`character_string`* can be obtained from *`file`*`.o` in one of the following ways:

- Method 1 - Using the `what` command. To use the `what` command to retrieve the character string, begin the character string with the characters @(#). For example, assume that `id.f` contains the following source code:

```
!DIR$ ID '@(#)file.f 03 February 1999'
      PRINT *, 'Hello, world'
      END
```

- The next step is to use file id.o as the argument to the `what` command, as follows:

```
% what id.o
% id.o:
%   file.f 03 February 1999
```

- Note that `what` does not include the special sentinel characters in the output.
- In the following example, *`character_string`* does not begin with the characters @(#). The output shows that `what` does not recognize the string.
- Input file `id2.o` contains the following:

```
!DIR$ ID  'file.f 03 February 1999'
      PRINT *, 'Hello, world'
      END
```

- The `what` command generates the following output:

```
% what id2.o
% id2.o:
```

- Method 2 - Using `strings` or `od`. The following example shows how to obtain output using the `strings` command.
- Input file `id.f` contains the following:

```
!DIR$ ID  "File id.f  Date: 03 February 1999"
      PRINT *, 'Hello, world'
      END
```

- The `strings` command generates the following output:

```
% strings id.o
02/03/9913:55:52f90
3.3cn
$MAIN
@CODE
@DATA
@WHAT
$MAIN
$STKOFEN
f$init
_FWF
$END
*?$F(6(
Hello, world
$MAIN
File: id.f  Date: 03 February 1999
% od -tc id.o
... portion of dump deleted
0000000001600  \0  \0  \0  \0  \0  \0  \0  \n   F   i   l   e   :       i   d
0000000001620   .   f           D   a   t   e   :       0   3       F   e   b
0000000001640   r   u   a   r   y       1   9   9   9  \0  \0  \0  \0  \0  \0
... portion of dump deleted
```

IGNORE_TKR

`!DIR$ IGNORE_TKR (`*`letter`*`) `*`dummy_arg`*` ...`

letter The *`letter`* can be T, K, or R, or any combination of these letters (for example, TK or KR). The *`letter`* applies only to the dummy argument it precedes. If *`letter`* appears, *`dummy_arg`* must appear.

dummy_arg If specified, it indicates the dummy arguments for which TKR rules should be ignored. If not specified, TKR rules are ignored for all dummy arguments in the procedure that contains the directive.

The `IGNORE_TKR` directive directs the compiler to ignore the type, kind, and/or rank (TKR) of specified dummy arguments in a procedure interface.

The directive causes the compiler to ignore the type, kind, and/or rank of the specified dummy arguments when resolving a generic call to a specific call. The compiler also ignores the type, kind, and/or rank on the specified dummy arguments when checking all the specifics in a generic call for ambiguities.

The following directive instructs the compiler to ignore type, kind, and/or rank rules for the dummy arguments of the following subroutine fragment:

```
subroutine example(A,B,C,D)
!DIR$ IGNORE_TKR A, (R) B, (TK) C, (K) D
```

Table 7. Explanation of Ignored TKRs

Dummy Argument	Ignored
A	Type, kind and rank is ignored
B	Only rank is ignored
C	Type and kind is ignored
D	Only kind is ignored

memory

```
!dir$ memory (list-of-attributes) [if(if-expr)]
heap-allocation-statement

!dir$ memory (list-of-attributes) [if(if-expr) ::] list-of-vars
variable-declaration

component-declaration
!dir$ memory (list-of-attributes) [if(if-expr) ::] list-of-components
```

The `memory` directive is a mechanism that allows developers to place variables and memory allocations in a specific type of memory. This directive is intended for systems with more than one type of explicitly-addressable memory, particularly when the entire application footprint will not fit into the desired type of memory (or the developer would like to place different memory regions in different types of memory). For example, the Intel Xeon Phi (codenamed Knights Landing or KNL) has both "normal" DDR memory and a limited capacity of "high-bandwidth" MCDRAM memory. Please refer to the `memory(7)` man page for full details on using this directive.

NAME

```
!DIR$ NAME (fortran_name="external_name", fortran_name="external_name" ... )
```

fortran_name The name used for the object throughout the Fortran program.

external_name The external form of the name.

The NAME directive allows the specification of a case-sensitive external name, or a name that contains characters outside of the Fortran character set, in a Fortran program. The case-sensitive external name is specified on the NAME directive.

Rules for Fortran naming do not apply to the *external_name* string; any character sequence is valid. Use this directive, for example, when writing calls to C routines.

```
      PROGRAM MAIN
!DIR$ NAME (FOO="XyZ")
      CALL FOO           ! XyZ is really being called
      END PROGRAM
```

The Fortran standard BIND feature provides some of the capability of the NAME directive.

NOFISSION

```
!DIR$ NOFISSION
```

Scope: Local

Instructs the compiler not to split statements in a given loop into distinct loops. Fission is prevented only for the loop specified; loops nested within the indicated loop remain fission candidates unless likewise annotated.

PREFETCH

```
!DIR$ PREFETCH [([lines(num)][, level(num)][, write][, nt])] var[, var] ...
```

lines(*num*) Specifies the number of cache lines to be prefetched. num is an expression that evaluates to an integer constant at compilation time. By default, the number of cache lines prefetched is 1.

level(*num*) Specifies the level of cache into which data is loaded. *num* is an expression that evaluates to an integer constant at compilation time. The cache level defaults to 1, the level closest to the processing unit. This level specification has little effect for current x86 targets.

write Specifies that the prefetch is for data to be written. When data is to be written, a prefetch instruction can move a block into the cache so that the expected store will be to the cache. Prefetch for write generally brings the data into the cache in an exclusive or modified state. By default, the prefetch is for data to be read. If the target architecture does not support prefetch for write, the prefetch will automatically become a prefetch for read.

nt Specifies that the prefetch is for non-temporal data. By default, the prefetch is for temporal data. Data with temporal locality (persistence), is expected to be accessed multiple times.

The general prefetch directive instructs the compiler to generate explicit prefetch instructions which load data from memory into cache prior to read or write access. The memory location to be prefetched is defined by *var*, which specifies any valid variable, member, or array element reference.

The compiler issues the prefetch instruction when it encounters the prefetch directive. The directive allows the user to influence almost every aspect of prefetch behavior. The default behavior prefetches one cache line, into L1 cache, for read access, and assumes temporal locality. The prefetch directive can be used inside and outside of loops, in a loop preamble, or before a function call to reduce cache-miss memory latency. The compiler will attempt to avoid multiple prefetches to the same cache line, which can be created as a result of optimization. All variables specified on the same prefetch directive line share the same behavior. If different behavior is needed for different variables, use multiple `prefetch` directive lines. The general `prefetch` directive supersedes the effects of any relevant `loop_info [no]prefetch` directives and the `-h [no]autoprefetch` command line option. The Cray Fortran compiler command line option `-x prefetch` can be used to disable all general

prefetch directives in Fortran source code. The compiler command line option `-h nopragma=prefetch` can be used to disable all general prefetch directives in C and C++ source code.

PREFETCH directive

```
real*8 a(m,n), b(n,p), c(m,p), arow(n)
...
do j = 1, p
!dir$ prefetch (lines(3), nt) arow(1),b(1,j)
    do k = 1, n, 4
!dir$ prefetch (nt) arow(k+24),b(k+24,j)
        c(i,j) = c(i,j) + arow(k) * b(k,j)
        c(i,j) = c(i,j) + arow(k+1) * b(k+1,j)
        c(i,j) = c(i,j) + arow(k+2) * b(k+2,j)
        c(i,j) = c(i,j) + arow(k+3) * b(k+3,j)
    enddo
enddo
```

PREPROCESS

`!DIR$ PREPROCESS expand_macros`

The PREPROCESS directive allows an include file to be processed when the compilation does not specify the preprocessing command line option. This directive does not cause preprocessing of included files, unless they too use the directive. If the preprocessing command line option is used, preprocessing occurs normally for all files.

To use the directive, it must be the first line in the include file and in each included file that needs to be preprocessing.

The optional `expand_macros` clause allows the compiler to expand all macros within the include files. Without this clause, macro expansion occurs only within preprocessing directives.

SAME_TBS

`!DIR$ SAME_TBS (`*`array`*`, `*`array`*`, `*`array`*`)`

array Two or more *array* arguments are required. *array* is the name of an assumed-shape dummy array. The arrays specified must not have the TARGET attribute. All arrays, specified on a single SAME_TBS directive must have same element type, bounds, and strides.

The SAME_TBS directive informs the compiler that the specified assumed size arrays are of the same rank and type, and that they have identical low-bound, extent, and stride multiplier information for corresponding dimensions. See the `SAME_TBS(7)` man page.

This information allows the compiler to generate more efficient code by reducing the number of potentially distinct intermediate values required for array element accesses. This may offer significant execution performance improvement when using assumed-shape dummy arrays of corresponding type, low-bound, extent, and stride.

The SAME_TBS directive applies only to the program unit in which it appears.

Ordinarily, for multidimensional assumed-shape arrays, some classes of loop optimizations cannot be performed when an assumed-shape dummy array is referenced or defined in a loop or an array assignment statement. The lost optimizations and the additional copy operations performed can significantly reduce the performance of a procedure that uses assumed-shape dummy arrays when compared to an equivalent procedure that uses explicit-shape array dummy arguments. This directive may provide significant performance improvement depending on certain factors such as greater numbers of assumed-shape arrays and smaller array sizes.

WEAK

```
!DIR$ WEAK procedure_name [, procedure_name] ...
!DIR$ WEAK procedure_name = stub_name[, procedure_name1 = stub_name1] ...
```

Sometimes, the code path of a program never executes at run time because of some condition. If this code path references a procedure that is external to the program (for example, a library procedure), the linker will add the binary for the procedure to the compiled program, resulting in a larger program.

When statically linking, the `weak` directive specifies an external identifier that may remain unresolved throughout the compilation. This directive has no effect when dynamically linking. The `WEAK` directive can prevent the compiler driver from adding the binary to the program, resulting in a smaller program and less use of memory.

The `WEAK` directive is used with procedures and variables to declare weak objects. The use of a weak object is referred to as a weak reference. The existence of a weak reference does not cause the compiler driver to add the appropriate binaries into a compiled program, so executing a weak reference will cause the program to fail. The compiler support for determining if the binary of a weak object is linked is deferred. To cause the compiler driver to add the binaries so the weak reference will work, a strong reference (a normal reference) must be somewhere in the program.

The following example illustrates the reason the `WEAK` directive is used. The startup code, which is compiled into every Fortran program, calls the SHMEM initialization routine, which causes the linker to add the binary of the initialization routine to every compiled program if a strong reference to the routine is used. This binary is unnecessary if a program does not use SHMEM. To avoid linking unnecessary code, the startup code uses the `WEAK` directive for the initialization routine. In this manner, if the program does not use SHMEM, the linker does not add the binary of the initialization routine even though the startup code calls it. However, if the program calls the SHMEM routines using strong references, the linker adds the necessary binaries, including the initialization binary into the compiled program.

The first form allows the specification of one or more weak objects. This form requires the implementation of code that senses that the *procedure_name* procedure is linked before calling it. The second form allows the pointing of a weak reference (*procedure_name*) to a stub procedure that exists in the code. This allows for calling of the stub if a strong reference to *procedure_name* does not exist. If a strong reference to *procedure_name* exists, it is called instead of the stub. The *stub_name* procedure must have the same name and dummy argument list as *procedure_name*.

The linker does not issue an unresolved reference error message for weak procedure references.

6.4 PGAS Directive

defer_sync

```
!DIR$ pgas defer_sync
```

PGAS (Partitioned Global Address Space) language data references made by the single statement immediately following the `pgas defer_sync` directive are not synchronized until the next fence instruction. Normally the compiler synchronizes the references in a statement as late as possible without violating program semantics. The purpose of the `pgas defer_sync` directive is to synchronize the references even later, beyond where the compiler can determine it is safe. Use this directive to force all references in the next statement to be non-blocking. This helps for cases where the compiler cannot prove that it is safe.

For example, if there is a remote-memory access (RMA) put near the end of a subroutine, the compiler must guard against the put value being read back immediately after the subroutine returns, so the put is synchronized just before returning. The programmer, however, may know that the value is not read back and can insert a `pgas defer_sync` directive.

`defer_sync` directive in Coarray Fortran

```
subroutine my_put( x, image, value )
    integer :: x[*], image, value
    !dir$ pgas defer_sync
    x[image] = value
    end subroutine
```

6.5 Scalar Optimization Directives

Scalar optimization directives control aspects of code generation, register storage, and other scalar operations.

[NO]COLLAPSE

`!DIR$ COLLAPSE (`*`loop-number1, loop-number2`* `[,`*`loop-number3`*`] ...)`

loop-number

Specify a value greater than or equal to 0 for the primary cache.

`!DIR$ NOCOLLAPSE`
Scope: Local

When the `collapse` directive is applied to a loop nest, the loop numbers of the participating loops must be listed in order of increasing access stride. Loop numbers range from 1 to the nesting level of the most deeply nested loop. The directive enables the compiler to assume appropriate conformity between trip counts. The compiler diagnoses misuse at compile time (when able); or, if `-h dir_check` is specified, at run time.

The `nocollapse` directive disqualifies the immediately following loop from collapsing with any other loop. Collapse is almost always desirable, so use this directive sparingly. Loop collapse is a special form of loop coalesce. Any perfect loop nest may be coalesced into a single loop, with explicit rediscovery of the intermediate values of original loop control variables. The rediscovery cost, which generally involves integer division, is quite high. Therefore, coalesce is rarely suitable for vectorization. It may be beneficial for multithreading. By definition, loop collapse occurs when loop coalesce may be done without the rediscovery overhead. To meet this requirement, all memory accesses must have uniform stride.

[NO]INTERCHANGE

`!DIR$ interchange(`*`loop_number1, loop_number2`*`[,`*`loop_number3`*`] ...)`

loop_number

Number from 1 to nesting depth of the most deeply nested loop

`!DIR$ nointerchange`
Scope: Local

The `interchange` control directives specify whether or not the order of the following two or more, perfectly nested loops should be interchanged. These directives apply to the subsequent loops.

The `interchange` directive specifies two or more loop numbers, ranging from 1 to the nesting depth of the most deeply nested loop, specified in any order. The compiler reorders perfectly nested loops. If they are not perfectly nested, unexpected results may occur.

The `nointerchange` directive inhibits loop interchange on the loop that immediately follows the directive.

NOSIDEEFFECTS

`!DIR$ NOSIDEEFFECTS` *`f`* `,` *`f`* `...`

f Symbolic name of a subprogram that the user is sure has no side effects. *f* must not be the name of a dummy procedure, module procedure, or internal procedure.

The `NOSIDEEFFECTS` directive allows the compiler to keep information in registers across a single call to subprogram without reloading the information from memory after returning from the subprogram. The directive is not needed for intrinsic functions.

`NOSIDEEFFECTS` declares that a called subprogram does not redefine any variables that meet the following conditions:

- Local to the calling program
- Passed as arguments to the subprogram
- Accessible to the calling subprogram through host association
- Declared in a common block or module
- Accessible through USE association

A procedure declared `NOSIDEEFFECTS` should not define variables in a common block or module shared by a program unit in the calling chain. All arguments should have the `INTENT(IN)` attribute; that is, the procedure must not modify its arguments. If these conditions are not met, results are unpredictable.

The `NOSIDEEFFECTS` directive must appear in the specification part of a program unit and must appear before the first executable statement.

The compiler may move invocations of a `NOSIDEEFFECTS` subprogram from the body of a DO loop to the loop preamble if the arguments to that function are invariant in the loop. This may affect the results of the program, particularly if the `NOSIDEEFFECTS` subprogram calls functions such as the random number generator or the real-time clock.

The effects of the `NOSIDEEFFECTS` directive are similar to those that can be obtained by specifying the `PURE` prefix on a function or a subroutine declaration. For more information about the `PURE` prefix, refer to the Fortran Standard.

SUPPRESS

`!DIR$ SUPPRESS` *`var`* `,` *`var`* `...`

var Variable that is to be stored to memory. If no variables are listed, all variables in the program unit are stored. If more than one variable is specified, use a comma to separate *var*s.

The `SUPPRESS` directive suppresses scalar optimization for all variables or only for those specified at the point where the directive appears. This often prevents or adversely affects vectorization of any loop that contains `SUPPRESS`.

At the point at which `!DIR$ SUPPRESS` appears in the source code, variables in registers are stored to memory (to be read out at their next reference), and expressions containing any of the affected variables are recomputed at their next reference after `!DIR$ SUPPRESS`. The effect on optimization is equivalent to that of an external

subroutine call with an argument list that includes the variables specified by `!DIR$ SUPPRESS` (or, if no variable list is included, all variables in the program unit).

SUPPRESS takes effect only if it is on an execution path. Optimization proceeds normally if the directive path is not executed because of a `GOTO` or `IF`.

```
      SUBROUTINE SUB (L)
      LOGICAL L
      A = 1.0          ! A is local
      IF (L) THEN
!DIR$ SUPPRESS         ! Has no effect if L is false
        CALL ROUTINE()
      ELSE
        PRINT *, A
      END IF
      END
```

In this example, optimization replaces the reference to A in the PRINT statement with the constant 1.0, even though `!DIR$ SUPPRESS` appears between A=1.0 and the PRINT statement. The IF statement can cause the execution path to bypass `!DIR$ SUPPRESS`. If SUPPRESS appears before the IF statement, A in PRINT* is not replaced by the constant 1.0.

6.6 Storage Directives

The directives described in this topic specify aspects of storing common blocks, variables, or arrays.

BLOCKABLE

`!DIR$ BLOCKABLE (`*`do_variable, do_variable`* `[,`*`do_variable`*`] ... )`

do_variable Loop control variable

The `BLOCKABLE` directive specifies that it is legal and desirable to cache block the subsequent loop nest, even when the compiler has not made such a determination. To be legally blockable, the nest must be perfect (without code between constituent loops), rectangular (trip counts of member loops are fixed over the life time of nest), and fully permutable (loop interchange and unrolling is legal at all levels). This directive both permits and requests blocking of the indicated loop nest.

If a `BLOCKINGSIZE` directive is also provided for the indicated loop, the following rules apply:

- If `BLOCKINGSIZE` is at least two, the indicated blockingsize is used.
- If `BLOCKINGSIZE` is zero, the loop itself is not blocked and it is treated as an inner loop (as part of the nest that traverses the cache block tile).
- If `BLOCKINGSIZE` is one, the loop itself is not blocked and it is treated as an outer loop (as a loop in the nest that moves from tile to tile).

When no blockingsize directive is supplied the compiler chooses the blockingsize according to its own heuristics.

BLOCKINGSIZE

`!DIR$ BLOCKINGSIZE (`*`n1`* `[,`*`n2`*`])`

```
!DIR$ noblocking
```

n1

Specify a value greater than or equal to 0 for the primary cache.

n2

Specify a value less than or equal to 2**30 for the secondary cache.

If *n1* or *n2* are 0, the loop is not blocked, but the entire loop is inside the block.

The `BLOCKINGSIZE` directive asserts that the loop following the directive is involved in a cache blocking situation for the primary or secondary cache.

The `NOBLOCKING` directive prevents the compiler from involving the subsequent loop in a cache blocking situation.

If the loop is involved in a blocking situation, it will have a block size of *n1* for the primary cache and *n2* for the secondary cache. The compiler attempts to include this loop within such a block but cannot guarantee inclusion.

`BLOCKINGSIZE` Directive

The compiler makes 20 x 20 blocks when blocking, but it could block the loop nest such that loop K is not included in the file.

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO K = 1, N
!DIR$ BLOCKABLE(J,I)
!DIR$ BLOCKING SIZE (20)
         DO J = 1, M
!DIR$ BLOCKING SIZE (20)
            DO I = 1, MM
               Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
            END DO
         END DO
      END DO
      END
```

If K is excluded, add a `BLOCKINGSIZE(0)` directive just before loop K to specify that the compiler should generate a loop such as the following example:

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO JJ = 1, M, 20
         DO II = 1, MM, 20
            DO K = 1, N
               DO J = JJ, MIN(M, JJ+19)
                  DO I = II, MIN(MM, II+19)
                     Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
                  END DO
               END DO
            END DO
         END DO
      END DO
      END
```

NOBLOCKING

```
!DIR$ NOBLOCKING
```

Asserts that the loop following the directive should not be cache blocked for the primary or secondary cache. It is an error to place a noblocking directive before a loop that is part of a blockable collection.

STACK

```
!DIR$ STACK
```

The STACK directive causes storage to be allocated to the stack in the program unit that contains the directive. This directive overrides the `-ev` command line option in specific program units of a compilation unit. For more information about the `-ev` command line option, see *Miscellaneous Fortran Specific Options*.

Data specified in the specification part of a module or in a DATA statement is always allocated to static storage. This directive has no effect on this static storage allocation.

All SAVE statements are honored in program units that also contain a STACK directive. This directive does not override the SAVE statement.

If the compiler finds a STACK directive and a SAVE statement without any objects specified in the same program unit, a warning message is issued.

The following rules apply when using this directive:

- It must be specified within the scope of a program unit.
- If it is specified in the specification part of a module, a message is issued. The STACK directive is allowed in the scope of a module procedure.

6.7 Vectorization Directives

Because vector operations cannot be expressed directly in the compiler, the compiler must be capable of transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures. Compiler directives may be used to control vectorization.

COPY_ASSUMED_SHAPE

```
!DIR$ COPY_ASSUMED_SHAPE array , array ...
```

array The name of an array to be copied to temporary storage. If no *array* names are specified, all assumed-shape dummy arrays are copied to temporary contiguous storage upon entry to the procedure. When the procedure is exited, the arrays in temporary storage are copied back to the dummy argument arrays. If one or more arrays are specified, only those arrays specified are copied. The arrays specified must not have the TARGET attribute.

All arrays specified, or all assumed-shape dummy arrays (if specified without *array* arguments), on a single COPY_ASSUMED_SHAPE directive must be shape conformant with each other. Incorrect code may be generated if the arrays are not. The `-R c` command line option can be used to verify whether the arrays are shape conformant.

The COPY_ASSUMED_SHAPE directive copies assumed-shape dummy array arguments into contiguous local temporary storage upon entry to the procedure in which the directive appears. During execution, it is the temporary storage that is used when the assumed-shape dummy array argument is referenced or defined.

The COPY_ASSUMED_SHAPE directive applies only to the program unit in which it appears.

Assumed-shape dummy arguments cannot be assumed to be stored in contiguous storage. In the case of multidimensional arrays, the elements cannot be assumed to be stored with uniform stride between each element of the array. These conditions can arise, for example, when an actual array argument associated with an assumed-shape dummy array is a non-unit strided array slice or section.

If the compiler cannot determine whether an assumed-shape dummy array is stored contiguously or with a uniform stride between each element, some optimizations are inhibited in order to ensure that correct code is generated. If an assumed-shape dummy array is passed to a procedure and becomes associated with an explicit-shape dummy array argument, additional copy-in and copy-out operations may occur at the call site. For multidimensional assumed-shape arrays, some classes of loop optimizations cannot be performed when an assumed-shape dummy array is referenced or defined in a loop or an array assignment statement. The lost optimizations and the additional copy operations performed can significantly reduce the performance of a procedure that uses assumed-shape dummy arrays when compared to an equivalent procedure that uses explicit-shape array dummy arguments.

The `COPY_ASSUMED_SHAPE` directive causes a single copy to occur upon entry and again on exit. The compiler generates a test at run time to determine wheteher the array is contiguous. If the array is contiguous, the array is not copied. This directive allows the compiler to perform all the optimizations it would otherwise perform if explicit-shape dummy arrays were used. If there is sufficient work in the procedure using assumed-shape dummy arrays, the performance improvements gained by the compiler outweigh the cost of the copy operations upon entry and exit of the procedure.

CONCURRENT

`!DIR$ CONCURRENT [ SAFE_DISTANCE=`*n*`]`

n An integer number that represents the number of additional consecutive loop iterations that can be executed in parallel without danger of data conflict. *n* must be an integer constant > 0. If SAFE_DISTANCE=*n* is not specified, the distance is assumed to be infinite, and the compiler ignores all cross-iteration data dependencies. The CONCURRENT directive is ignored if the SAFE_DISTANCE argument is used and vectorization is requested on the command line.

The `CONCURRENT` directive indicates that no data dependence exists between array references in different iterations of the loop. This directive affects the loop that immediately follows it. This can be useful for vectorization optimizations.

Consider the following code:

```
!DIR$ CONCURRENT SAFE_DISTANCE=3
      DO I = K+1, N
        X(I) = A(I) + X(I-K)
      ENDDO
```

The `CONCURRENT` directive in this example informs the optimizer that the relationship K>3 is true. This allows the compiler to load all of the following array references safely during the *I*th loop iteration:

```
X(I-K)
X(I-K+1)
X(I-K+2)
X(I-K+3)
```

HAND_TUNED

```
!DIR$ HAND_TUNED
```

Assert that the code in the next loop nest has been arranged by hand for maximum performance, and the compiler should restrict some of the more aggressive automatic expression rewrites. The compiler should still fully optimize and vectorize the loop within the constraints of the directive. The hand_tuned directive applies to the next loop in the same manner as the `CONCURRENT` and `SAFE_ADDRESS` directives.

Use of this directive may severely impede performance. Use carefully and evaluate performance before and after employing this directive.

IVDEP

```
!DIR$ IVDEP [ SAFEVIL=vlen | INFINITEVL]
```

vlen Specifieds a vector length in which no dependency will occur. *vlen* must be an integer between 1 and 1024 inclusive.

INFINITEVL Specifies an infinite safe vector length. That is, no dependency will occur at any vector length.

When the `IVDEP` directive appears before a loop, the compiler ignores vector dependencies, including explicit dependencies, in any attempt to vectorize the loop. ivdep applies only to the first for loop or while loop that follows the directive within the same program unit.

For array operations, Fortran requires that the complete right-hand side (RHS) expression be evaluated before the assignment to the array or array section on the left-hand side (LHS). If possible dependencies exist between the RHS expression and the LHS assignment target, the compiler creates temporary storage to hold the RHS expression result. If an `IVDEP` directive appears before an array syntax statement, the compiler ignores potential dependencies and suppresses the creation and use of array temporaries for that statement. Using array syntax statements allows the reference of referencing arrays in a compact manner. Array syntax allows the use of either the array name, or the array name with a section subscript, to specify actions on all the elements of an array, or array section, without using DO loops.

If no vector length is specified, the vector length used is infinity.

If a loop with an `IVDEP` directive is enclosed within another loop with an `IVDEP` directive, the `IVDEP` directive on the outer loop is ignored. When the Cray compiler vectorizes a loop, it may reorder the statements in the source code to remove vector dependencies. When `IVDEP` is specified, the statements in the loop or array syntax statement are assumed to contain no dependencies as written, and the Cray compiler does not reorder loop statements.

LOOP_INFO

```
!DIR$ LOOP_INFO min_trips(c) est_trips(c) max_trips(c) cache( symbol , symbol ... )
cache_nt( symbol , symbol ... ) prefetch noprefetch
```

c

An expression that evaluates to an integer constant at compilation time.

min_trips

Specifies guaranteed minimum number of trips.

est_trips

Specifies estimated or average number of trips.

max_trips

Specifies guaranteed maximum number of trips.

cache

Specifies that *symbol* is to be allocated in cache; this is the default if no hint is specified and the cache_nt directive is not specified.

cache_nt

Specifies that *symbol* is to use non-temporal reads and writes.

symbol

The base name of the object that should (cache) or should not (cache_nt) be placed into cache. This can be the base name of any object such as an array or scalar structure without member references. If specifying a pointer in the list, only the references, not the pointer itself, are subject to the cache or cache_nt instruction.

prefetch

Specifies a preference that prefetches be performed for the following loop.

noprefetch

Specifies a preference that no prefetches be performed for the following loop.

The LOOP_INFO directive allows additional information to be specified about the behavior of a loop, including run time trip count, hints on cache allocation strategy, and threading preference. The LOOP_INFO directive provides information to the optimizer and can produce faster code sequences.

Use LOOP_INFO immediately before a for loop to indicate minimum, maximum, estimated trip count. The compiler will diagnose misuse at compile time when able, or when option -h dir_check is specified at run time.

For cache allocation hints, use the LOOP_INFO directive to override default settings, cache or cache_nt directives, or override automatic cache management decisions. The cache hints are local and apply only to the specified loop nest.

Use the LOOP_INFO PREFER_THREAD directive to indicate the preference that the loop following the directive be threaded. The LOOP_INFO PREFER_NOTHREAD indicates the preference that the loop following the directive should not be threaded.

The prefetch clause instructs the compiler to preload scalar data into the first-level cache to improve the frequency of cache hits and to lower latency. They are generated in situations where the compiler expects them to improve performance. Strategic use of prefetch instructions can hide latency for scalar loads that feed vector instructions or scalar loads in purely scalar loops. Prefetch instructions are generated at default and higher levels of optimization. Thus, they are turned off at -O0 or -O1. Prefetch can be turned off at the loop level via the following directive:

```
!DIR$ LOOP_INFO NOPREFETCH
     DO I = 1, N
```

LOOP_INFO PREFER_[NO]THREAD

```
!DIR$ LOOP_INFO PREFER_THREAD
     DO I = 1, N
```

```
!DIR$ LOOP_INFO PREFER_THREAD
     DO J = 1, N
```

The PREFER_THREAD and PREFER_NOTHREAD directives are special cases of the LOOP_INFO advisory directive. Use these directives to indicate a preference for turning threading on or off for the subsequent loop. Use the LOOP_INFO PREFER_THREAD directive to indicate the preference that the loop following the directive be threaded. Use the LOOP_INFO PREFER_NOTHREAD directive to indicate that the loop should not be threaded.

NEXTSCALAR

```
!DIR$ NEXTSCALAR
```

The NEXTSCALAR directive disables vectorization for the first DO loop or DO WHILE loop that follows the directive. The directive applies to only one loop, the first loop that appears after the directive within the same program unit. NEXTSCALAR is ignored if vectorization has been disabled.

If the NEXTSCALAR directive appears prior to any array syntax statement, it disables vectorization for the array syntax statement.

[NO]PATTERN

```
!DIR$ PATTERN
!DIR$ NOPATTERN
```

The nopattern directive disables pattern matching for the loop immediately following the directive. 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, use the nopattern directive to disable pattern matching and cause the compiler to generate inline code.

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

By default, the compiler would detect that the following loop is a matrix multiply and replace it with a call to a matrix multiply library routine. By preceding the loop with a !DIR$ NOPATTERN directive, however, pattern matching is inhibited and no replacement is done.

```
!DIR$ NOPATTERN
       DO k= 1,n
         DO i= 1,n
           DO j= 1,m
             A(i,j) = A(i,j) + B(i,k) * C(k,j)
           END DO
         END DO
       END DO
```

[NO]VECTOR

```
!DIR$ VECTOR [clause[, clause]... ]
!DIR$ NOVECTOR
!dec$ vector [clause[, clause]... ]
```

ALWAYS Vectorize the loop that immediately follows the directive. This directive states a vectorization preference and does not guarantee that the loop has no memory-dependence hazard. This directive has the same effect as the prefervector directive.

ALIGNED Directs the compiler to generate aligned data movement instructions for array references when vectorizing. For current Intel processors, data alignment is necessary for efficient vectorization. Use with care to improve performance. If some of the access patterns are actually unaligned, using the ALIGNED clause may generate incorrect code. This directive also directs the compiler to ignore explicit and implicit vector dependencies.

UNALIGNED Directs the compiler to generate unaligned data movement instructions for all array references when vectorizing.

The `novector` directive suppresses compiler attempts to vectorize loops and array syntax statements. It overrides any other vectorization-related directives, as well as the `-h vector` and `-O vector`*n* command line options. These directives are ignored if vectorization or scalar optimization has been disabled.

In Fortran, `NOVECTOR` applies to the rest of the program unit unless it is superseded by a `VECTOR` directive. When `NOVECTOR` has been used within the same program unit, `VECTOR` causes the compiler to resume its attempts to vectorize loops and array syntax statements. After a `VECTOR` directive is specified, automatic vectorization is enabled for all loop nests.

PERMUTATION

```
!DIR$ PERMUTATION (symbol [, symbol ] ...)
```

ia Integer array that has no repeated values for the entire routine.

Specifies that an integer array has no repeated values. This directive is useful when the integer array is used as a subscript for another array (vector-valued subscript). This directive may improve code performance.

In a sequence of array accesses that read array element values from the specified symbols with no intervening accesses that modify the array element values, each of the accessed elements will have a distinct value.

When an array with a vector-valued subscript appears on the left side of the equal sign in a loop, many-to-one assignment is possible. Many-to-one assignment occurs if any repeated elements exist in the subscripting array. If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that many-to-one assignment does not exist with that array reference.

`permutation` Directive

Sometimes a vector-valued subscript is used as a means of indirect addressing because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to select only the desired elements, and no repeated elements exist in the integer array. The permutation directive does not apply to the array a. Rather, it applies to the pointer used to index into it, `ipnt`. By knowing that ipnt is a permutation, the compiler can safely generate an unordered scatter for the write to a.

```
!DIR$ PERMUTATION(IPNT) ! IPNT has no repeated values
      ...
      DO I = 1, N
         A(IPNT(I)) = B(I) + C(I)
      END DO
```

[NO]PIPELINE

```
!DIR$ PIPELINE
!DIR$ NOPIPELINE
```

Software-based vector pipelining (software vector pipelining) provides additional optimization beyond the normal hardware-based vector pipelining. In software vector pipelining, the compiler analyzes all vector loops and automatically attempts to pipeline a loop if doing so can be expected to produce a significant performance gain. This optimization also performs any necessary loop unrolling.

In some cases the compiler either does not pipeline a loop that could be pipelined or pipelines a loop without producing performance gains. In these situations, use the `PIPELINE` or `NOPIPELINE` directive to advise the compiler to pipeline or not pipeline the loop immediately following the directive.

Software vector pipelining is valid only for the innermost loop of a loop nest. These directives are advisory only. While the `NOPIPELINE` directive can be used to inhibit automatic pipelining, and the `PIPELINE` directive can be used to attempt to override the compiler's decision not to pipeline a loop, the compiler cannot be forced to pipeline a loop that cannot be pipelined.

Vector loops that have been pipelined generate compile-time messages to that effect, if optimization messaging is enabled (`-O msgs`).

PREFERVECTOR

```
!DIR$ PREFERVECTOR
```

Directs the compiler to vectorize the loop immediately following 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.

In the following example, both loops can be vectorized, but the compiler generates vector code for the outer `DO I` loop:

```
!DIR$ PREFERVECTOR
      DO I = 1, N
        DO J = 1, M
          A(I) = A(I) + B(J,I)
        END DO
      END DO
```

PROBABILITY

```
!DIR$ probability const
!DIR$ probability_almost_always
!DIR$ probability_almost_never
```

const Expression that evaluates to a floating point constant at compilation time. (0.0 <= *const* <= 1.0.)

The probability directives specify information used by interprocedure analysis (IPA) and the optimizer to produce faster code sequences. The specified probability is a hint, rather than a statement of fact. This information is used to guide inlining decisions, branch elimination optimizations, branch hint marking, and the choice of the optimal algorithmic approach to the vectorization of conditional code. These directives can appear anywhere executable code is legal. Each directive applies to the block of code where it appears. It is important to realize that the directive should not be applied to a conditional test directly; rather, it should be used to indicate the relative probability of a `THEN` or `ELSE` branch being executed.

Specify `almost_never` and `almost_always` by using the `probability const` values 0.0 and 1.0, respectively.

PROBABILITY directive

This example states that the probability of entering the block of code with the assignment statement is 0.3 or 30%. This also means that `a[i]` is expected to be greater than `b[i]` 30% of the time. Note that the probability directive appears within the conditional block of code, rather

than before it. This removes some of the ambiguity that has plagued other implementations that tie the directive directly to the conditional code.

```
     IF ( A(I) > B(I) ) THEN
!DIR$ PROBABILITY 0.3
         A(I) = B(I)
      ENDIF
```

For vector IF code, a probability of very low (<0.1) or `probability_almost_never` will cause the compiler to use the vector gather/scatter methods used for sparse IF vector code instead of the vector merge methods used for denser IF code. For example:

```
      DO I = 1,N
         IF ( A(I) > 0.0 ) THEN
!DIR$ PROBABILITY_ALMOST_NEVER
            B(I) = B(I)/A(I) + A(I)/B(I) ! EVALUATE USING SPARSE
METHODS
         ENDIF
      ENDDO
```

Note that the PROBABILITY directive appears within the conditional, rather than before the condition. This removes some of the ambiguity of tying the directive directly to the conditional test.

SAFE_ADDRESS

Specifies that it is safe to speculatively execute memory references within all conditional branches of a loop; these memory references can be safely executed in each iteration of the loop. For most code, this directive can improve performance significantly by preloading vector expressions. However, most loops do not require this directive to have preloading performed. SAFE_ADDRESS is required only when the safety of the operation cannot be determined or index expressions are very complicated.

The SAFE_ADDRESS directive is an advisory directive. That is, the compiler may override the directive if it determines the directive is not beneficial. If the directive is not used on a loop and the compiler determines that it would benefit from the directive, it issues a message indicating such. The message is similar to this:

```
  DO I = 1,N
FTN-6375 FTN_DRIVER.EXE: VECTOR X7, FILE = 10928.F, LINE = 110
  A LOOP STARTING AT LINE 110 WOULD BENEFIT FROM "!DIR$ SAFE_ADDRESS".
```

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

To see the messages, use the `-O msgs` option.

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

SAFE_ADDRESS directive

In this example, the compiler will not preload vector expressions, because the value of `j` is unknown. However, if it is known that references to `b (i,j)` are safe to evaluate for all iterations of the loop, regardless of the condition, the SAFE_ADDRESS directive can be used. With the directive, the compiler can load `b (i,j)` with a full vector mask, merge `0.0` where the condition is true, and store the resulting vector using a full mask.

```
SUBROUTINE X3( A, B, N, M, J )
REAL A(N), B(N,M)

!DIR$ SAFE_ADDRESS
DO I = 1,64            ! VECTORIZED LOOP
   IF ( A(I).NE.0.0 ) THEN
      B(I,J) = 0.0     ! VALUE OF 'J' IS UNKNOWN
   ENDIF
ENDDO
END
```

SAFE_CONDITIONAL

`!DIR$ SAFE_CONDITIONAL`

Specifies that it is safe to execute all memory references and arithmetic operations within all conditional branches of the subsequent scalar or vector loop nest. It can improve performance by allowing the hoisting of invariant expressions from conditional code and by allowing prefetching of memory references.

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

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

SAFE_CONDITIONAL directive

In the example below, the compiler cannot precompute the invariant expression s1*s2 because these values are unknown and may cause an arithmetic trap if executed unconditionally. However, if the condition is known to be true at least once, then it is safe to use the `SAFE_CONDITIONAL` directive and execute s1*s2 speculatively. With the directive, the compiler evaluates s1*s2 outside of the loop, rather than under control of the conditional code. In addition, all control flow is removed from the body of the vector loop as s1*s2 no longer poses a safety risk.

```
      SUBROUTINE SAFE_COND( A, N, S1, S2 )
      REAL A(N), S1, S2

!DIR$ SAFE_CONDITIONAL
      DO I = 1,N
         IF ( A(I) /= 0.0 ) THEN
            A(I) = A(I) + S1*S2
         ENDIF
      ENDDO
      END
```

SAME_TBS

```
!DIR$ SAME_TBS array, array, array)
```

array Two or more *array* arguments are required. *array* is the name of an assumed-shape dummy array. The arrays specified must not have the TARGET attribute. All arrays, specified on a single SAME_TBS directive must have the same element type, bounds, and strides. Use the `-Rd` command line option to verify that the arrays have the same element type, bounds, and strides.

The SAME_TBS directive informs the compiler that the specified assumed size arrays are of the same rank and type, and that they have identical low-bound, extent, and stride multiplier information for corresponding dimensions. See the `SAME_TBS(7)` man page.

This information allows the compiler to generate more efficient code by reducing the number of potentially distinct intermediate values required for array element accesses. This may offer significant execution performance improvement when using assumed-shape dummy arrays of corresponding type, low-bound, extent, and stride.

The SAME_TBS directive applies only to the program unit in which it appears.

Oridnarily, for multidimensional assumed-shape arrays, some classes of loop optimizations cannot be performed when an assumed-shape dummy array is referenced or defined in a loop or an array assignment statement. The lost optimizations and the additional copy operations performed can significantly reduce the performance of a procedure that uses assumed-shape dummy arrays when compared to an equivalent procedure that uses explicit-shape array dummy arguments. This directive may provide significant performance improvement depending on certain factors such as greater numbers of assumed-shape arrays and smaller array sizes.

[NO]UNROLL

```
!DIR$ UNROLL
!DIR$ NOUNROLL
```

n Specifies the total number of loop body copies to be generated. *n* is an integer value from 0 through 1024.

If a value for *n* is specified, the compiler unrolls the loop by that amount. If *n* is not specified, the compiler determines if it is appropriate to unroll the loop, and if so, the unroll amount.

The subsequent `DO` loop is not unrolled if `UNROLL0`, `UNROLL1`, or `NOUNROLL` are specified. These directives are equivalent.

Scope: Local

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

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

Disable loop unrolling for the next loop. The `NOUNROLL` directive is functionally equivalent to the `UNROLL 0` and `UNROLL 1` directives. The *n* argument applies only to the `UNROLL` directive. If a value for *n* is not specified, the compiler will determine the number of copies to generate based on the number of statements in the loop nest. Note: The compiler cannot always safely unroll non-innermost loops due to data dependencies. In these cases, the directive is ignored. The `UNROLL` 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.

Unroll outer loops

Assume that the outer loop of the following nest will be unrolled by two:

```
!DIR$ UNROLL 2
      DO I = 1, 10
        DO J = 1,100
              A(J,I) = B(J,I) + 1
        END DO
      END DO
```

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

```
DO I = 1, 10, 2
  DO J = 1,100
        A(J,I) = B(J,I) + 1
  END DO
  DO J = 1,100
        A(J,I+1) = B(J,I+1) + 1
  END DO
END DO
```

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

```
DO I = 1, 10, 2
  DO J = 1,100
        A(J,I)   = B(J,I) + 1
        A(J,I+1) = B(J,I+1) + 1
  END DO
END DO
```

Illegal unrolling of outer loops

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):

```
!DIR$ UNROLL 2
      DO I = 1, 10
        DO J = 1,100
              A(J,I) = A(J-1,I+1) + 1
        END DO
      END DO
```

Unrolling nearest neighbor pattern

The following example shows unrolling with nearest neighbor pattern. This allows register reuse and reduces memory references from 2 per trip to 1.5 per trip.

```
!DIR$ UNROLL 2
      DO J = 1,N
         DO I = 1,N      ! VECTORIZE
            A(I,J) = B(I,J) + B(I,J+1)
```

```
         ENDDO
      ENDDO
```

The preceding code fragment is converted to the following code:

```
DO J = 1,N,2       ! UNROLLED FOR REUSE OF B(I,J+1)
   DO I = 1,N      ! VECTORIZED
      A(I,J) = B(I,J) + B(I,J+1)
      A(I,J+1) = B(I,J+1) + B(I,J+2)
   END DO
END DO
```

7 Source Preprocessing

Source preprocessing helps port a program from one platform to another by allowing source text to be specified that is platform specific.

For a source file to be preprocessed automatically, it must have an uppercase extension, either .F or .FOR (for a file in fixed source form), or .F90, F95,.F03, .F08, or `.FTN` (for a file in free source form). To specify preprocessing of source files with other extensions, including lowercase ones, use the -eP or -eZ options described in *Command Line Options*.

General Rules

Alter the source code through source preprocessing directives. These directives are fully explained below in *Directives*. The directives must be used according to the following rules:

- Do not use source preprocessor (#) directives within multiline compiler directives (CDIR$, !DIR$, CSD$, !CSD $, C$OMP, or !$OMP).
- A source file that contains an #if directive can not be included without a balancing #endif directive within the same file.
- The #if directive includes the #ifdef and #ifndef directives.
- If a directive is too long for one source line, the backslash character (\) is used to continue the directive on successive lines. Successive lines of the directive can begin in any column.
- The backslash character (\) can appear in any location within a directive in which white space can occur. A backslash character (\) in a comment is treated as a comment character. It is not recognized as signaling continuation.
- Every directive begins with the pound character (#), and the pound character (#) must be in column 1.
- Blank and tab (HT) characters can appear between the pound character (#) and the directive keyword.
- Form feed (FF) or vertical tab (VT) characters cannot be written to separate tokens on a directive line. That is, a source preprocessing line must be continued, by using a backslash character (\), if it spans source lines.
- Blanks are significant, so the use of spaces within a source preprocessing directive is independent of the source form of the file. The fields of a source preprocessing directive must be separated by blank or tab (HT) characters.
- Any user-specified identifier that is used in a directive must follow Fortran rules for identifier formation. The exceptions to this rule are as follows:
  - The first character in a source preprocessing name (a macro name) can be an underscore character (_).
  - Source preprocessing names are significant in their first 132 characters whereas a typical Fortran identifier is significant only in its first 63 characters.
- Source preprocessing identifier names are case sensitive.
- Numeric literal constants must be integer literal constants or real literal constants, as defined for Fortran.

- Comments written in the style of the C language, beginning with /* and ending with */, can appear anywhere within a source preprocessing directive in which blanks or tabs can appear. The comment, however, must begin and end on a single source line.
- Directive syntax allows an identifier to contain the ! character. Therefore, placing the ! character to start a Fortran comment on the same line as the directive should be avoided.

Directives

The blanks shown in the syntax descriptions of the source preprocessing directives are significant. The tab character (HT) can be used in place of a blank. Multiple blanks can appear wherever a single blank appears in a syntax description.

- `#include` Directive

  The `#include` directive directs the system to use the content of a file. Just as with the `INCLUDE` line path processing defined by the Fortran standard, an `#include` directive effectively replaces that directive line by the content of *`filename`*. This directive has the following formats:

  ```
  #include "filename"
  ```

  ```
  #include <filename>
  ```

  `filename` A file or directory to be used.

  In the first form, if filename does not begin with a slash (/) character, the system searches for the named file, first in the directory of the file containing the `#include` directive, then in the sequence of directories specified by the `-I` option(s) on the `ftn` command line, and then the standard (default) sequence. If *`filename`* begins with a slash (/) character, it is used as is and is assumed to be the full path to the file.

  The second form directs the search to begin in the sequence of directories specified by the `-I` option(s) on the `ftn` command line and then search the standard (default) sequence.

  The Fortran standard prohibits recursion in `INCLUDE` files, so recursion is also prohibited in the `#include` form.

  The `#include` directives can be nested.

  When the compiler is invoked to do only source preprocessing, not compilation, text will be included by `#include` directives but not by Fortran `INCLUDE` lines.

- `#define` Directive

  The `#define` directive allows for the declaration of a variable and assign a value to the variable. It also allows the definition of a function-like macro. This directive has the following format:

  ```
  #define identifier value
  ```

  ```
  #define identifier (dummy_arg_list) value
  ```

  The first format defines an object-like macro (also called a source preprocessing variable), and the second defines a function-like macro. In the second format, the left parenthesis that begins the *`dummy_arg_list`* must immediately follow the identifier, with no intervening white space.

  `identifier` The name of the variable or macro being defined.

Rules for Fortran variable names apply; that is, the name cannot have a leading underscore character (_). For example, `ORIG` is a valid name, but `_ORIG` is invalid.

dummy_arg_list A list of dummy argument identifiers.

value The *value* is a sequence of tokens. The *value* can be continued onto more than one line using backslash (\) characters.

If a preprocessor *identifier* appears in a subsequent `#define` directive without being the subject of an intervening `#undef` directive, and the *value* in the second `#define` directive is different from the value in the first `#define` directive, then the preprocessor issues a warning message about the redefinition. The second directive's *value* is used.

When an object-like macro's identifier is encountered as a token in the source file, it is replaced with the value specified in the macro's definition. This is referred to as an invocation of the macro.

The invocation of a function-like macro is more complicated. It consists of the macro's identifier, immediately followed by a left parenthesis with no intervening white space, then a list of actual arguments separated by commas, and finally a terminating right parenthesis. There must be the same number of actual arguments in the invocation as there are dummy arguments in the `#define` directive. Each actual argument must be balanced in terms of any internal parentheses. The invocation is replaced with the value given in the macro's definition, with each occurrence of any dummy argument in the definition replaced with the corresponding actual argument in the invocation.

For example, the following program prints `Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING 'Hello, world.'
       PRINT *, GREETING
       END PROGRAM P
```

The following program prints `Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING(str1, str2) str1, str1, str2
       PRINT *, GREETING('Hello, ', 'world.')
       END PROGRAM P
```

- `#undef` Directive

  The `#undef` directive sets the definition state of *identifier* to an undefined value. If *identifier* is not currently defined, the `#undef` directive has no effect. This directive has the following format:

  ```
  #undef identifier
  ```

  identifier The name of the variable or macro being defined.

- # (Null) Directive

  The null directive simply consists of the pound character (#) in column 1 with no significant characters following it. That is, the remainder of the line is typically blank or is a source preprocessing comment. This directive is generally used for spacing out other directive lines.

- Conditional Directives

  Conditional directives cause lines of code to either be produced by the source preprocessor or to be skipped. The conditional directives within a source file form if-groups. An if-group begins with an `#if`, `#ifdef`, or

`#ifndef` directive, followed by lines of source code that may or may not be skipped. Several similarities exist between the Fortran `IF` construct and if-groups:

- The `#elif` directive corresponds to the `ELSE IF` statement.
- The `#else` directive corresponds to the `ELSE` statement.
- Just as an `IF` construct must be terminated with an `END IF` statement, an if-group must be terminated with an `#endif` directive.
- Just as with an `IF` construct, any of the blocks of source statements in an if-group can be empty. For example, the following directives can be written:

```
#if MIN_VALUE == 1
#else
  ...
#endif
```

Determining which group of source lines (if any) to compile in an if-group is essentially the same as the Fortran determination of which block of an `IF` construct should be executed.

- `#if` Directive

The `#if` directive has the following format:

```
#if expression
```

expression An expression. The values in *`expression`* must be integer literal constants or previously defined preprocessor variables. The expression is an integer constant expression as defined by the C language standard. All the operators in the expression are C operators, not Fortran operators. The *`expression`* is evaluated according to C language rules, not Fortran expression evaluation rules.

Note that unlike the Fortran `IF` construct and `IF` statement logical expressions, expression in an `#if` directive need not be enclosed in parentheses.

The `#if` expression can also contain the unary `defined` operator, which can be used in either of the following formats:

- ```
  defined identifier
  ```
- ```
  defined (identifier)
  ```

When the `defined` subexpression is evaluated, the value is 1 if *`identifier`* is currently defined, and 0 if it is not.

All currently defined source preprocessing variables in *`expression`*, except those that are operands of `defined` unary operators, are replaced with their values. During this evaluation, all source preprocessing variables that are undefined evaluate to 0.

Note that the following two directives are **not** equivalent:

- `#if X`
- `#if defined(X)`

In the first case, the condition is true if X has a nonzero value. In the second case, the condition is true only if X has been defined (has been given a value that could be 0).

- `#ifdef` Directive

  The `#ifdef` directive is used to determine if *`identifier`* is predefined by the source preprocessor, has been named in a `#define` directive, or has been named in a `ftn -D` command line option. This directive has the following format:

  ```
  #ifdef identifier
  ```

  The `#ifdef` directive is equivalent to either of the following two directives:

  - ```
    #ifdefined identifier
    ```
  - ```
    #ifdefined (identifier)
    ```

- `#ifndef` Directive

  The `#ifndef` directive tests for the presence of an *`identifier`* that is not defined. This directive has the following format:

  ```
  #ifndef identifier
  ```

  This directive is equivalent to either of the following two directives:

  - ```
    #if ! defined identifier
    ```
  - ```
    #if ! defined (identifier)
    ```

- `#elif` Directive

  The `#elif` directive serves the same purpose in an if-group as does the `ELSE IF` statement of a Fortran `IF` construct. This directive has the following format:

  ```
  #elif expression
  ```

  `expression` The expression follows all the rules of the integer constant expression in an `#if` directive.

- `#else` Directive

  The `#else` directive serves the same purpose in an if-group as does the `ELSE` statement of a Fortran `IF` construct. This directive has the following format:

  ```
  #else
  ```

- `#endif` Directive

  The `#endif` directive serves the same purpose in an if-group as does the `END IF` statement of a Fortran `IF` construct. This directive has the following format:

  ```
  #endif
  ```

Predefined Macros

The Cray Fortran compiler source preprocessing supports a number of predefined macros. They are divided into groups as follows:

- Macros based on the host machine
- Macros based on CLE system targets
- Macros based on the Cray Fortran compiler
- Macros based on the source file

The following predefined macros are based on the host system (the system upon which the compilation is being done):

unix, __unix, __unix__	Always defined. (The leading characters in the second form consist of 2 consecutive underscores; the third form consists of 2 leading and 2 trailing underscores.)

The following predefined macros are based on CLE systems as targets:

_ADDR64	Defined for CLE systems as targets. The target system must have 64-bit address registers.
_OPENMP	Defined as the publication date of the OpenMP standard supported, as a string of the form *yyyymm*.
_MAXVL_8	Defined as 16, the number of 8-bit elements that fit in an XMM register ("vector length"). On targets that support AVX, defined as 32, the number of 8-bit elements that fit in a YMM register ("vector length").
_MAXVL_16	Defined as 8. On targets that support AVX, defined as 16.
_MAXVL_32	Defined as 4. On targets that support AVX, defined as 8.
_MAXVL_64	Defined as 2. On targets that support AVX, defined as 4.
_MAXVL_128	Defined as 0. On targets that support AVX, defined as 2.

The following macros are based on the Cray Fortran Compiler:

_CRAYFTN	Defined as 1.
_CRAY_COARRAY	Defined as 1 if -hcaf is specified on the command line. If -hnocaf is specified, this macro is undefined.

The following predefined macros are based on the source file:

__line__, __LINE__	Defined to be the line number of the current source line in the source file.
__file__, __FILE__	Defined to be the name of the current source file.
__date__, __DATE__	Defined to be the current date in the form *mm/dd/yy*.
__time__, __TIME__	Defined to be the current in the form *hh:mm:ss*.

Command Line Options

The following `ftn` command line options affect source preprocessing:

- The `-D` *identifier=value* option, which defines variables used for source preprocessing. For more information about this option, see *-D identifier=value*.
- The `-dF` option, which controls macro expansion in Fortran source statements. For more information about this option, see *-d disable_opt and -e enable_opt* .

- The `-eP` option, which performs source preprocessing on *`file`*.f90,*`file`*.F90, *`file`*.F95, *`file`*.F03, *`file`*.F08, *`file`*`.ftn`, or *`file`*`.FTN` but does not compile. The -eP option produces *`file`*`.i`. For more information about this option, see *-d disable_opt and -e enable_opt* .
- The -eZ option, which performs source preprocessing and compilation on *`file`*.f90, *`file`*.F90, *`file`*.F95, *`file`*.F03, *`file`*.F08, *`file`*`.ftn`, or *`file`*`.FTN`. The -eZ option produces *`file`*`.i`. For more information about this option, see *-d disable_opt and -e enable_opt* .
- The `-U identifier` , *`identifier`* ... option, which undefines variables used for source preprocessing. For more information about this option, see *-U identifier ,`identifier`...*.

  The `-D identifier` =*`value`* and -U *`identifier`* , *`identifier`* ... options are ignored unless one of the following is true:

  - The Fortran input source file is specified as either *`file`*.f90,*`file`*.F90, *`file`*.F95, *`file`*.F03, *`file`*.F08, *`file`*`.ftn`, or *`file`*`.FTN`.
  - The -eP or -eZ options have been specified.

8 OpenMP Overview

The OpenMP API provides a parallel programming model that is portable across shared memory architectures from Cray and other vendors. The OpenMP specification is accessible at *http://openmp.org/wp/openmp-specifications/*.

Supported Version

CCE supports the OpenMP API, Version 4.5, with the following exceptions. The most up-to-date exceptions are listed on the man pages:

- Support for OpenMP Random Access Iterators (RAIs) in the C++ Standard Template Library (STL) is deferred.
- Cancellation does not destruct/deallocate implicitly private local variables. It correctly handles explicitly private variables.
- The `device` clause is not supported. The other mechanisms for selecting a default device are supported: `OMP_DEFAULT_DEVICE` and `omp_set_default_device`.
- The only API calls allowed in target regions are: `omp_is_initial_device`, `omp_get_thread_num`, `omp_get_num_threads`, `omp_get_team_num`, and `omp_get_num_teams`.
- User-defined reductions are not supported in `target` regions.
- Individual structure members are not supported in the map clause or the target update construct. Instead, CCE only supports mapping and updating entire structure variables, handling all of the members together as a single aggregate object.

The following additional notes apply to CCE's OpenMP support:

- Task migration is not implemented. An untied task that starts execution on a thread and suspends will always resume execution on that same thread.
- `simd` functions will not vectorize if inlining is disabled or the function definition is not visible at the callsite.
- `simd` loops containing function calls will not vectorize if inlining is disabled or the function definitions are not visible.
- Parallel constructs are supported in target regions, but they are limited to one thread.

Compiling

By default, the CCE compiler recognizes OpenMP directives. These CCE options affect OpenMP applications:

- `-h [no]omp`
- `-h thread`*n*

Executing

For OpenMP applications, use both the `OMP_NUM_THREADS` environment variable to specify the number of threads and the `aprun -d`*`depth`* option to specify the number of CPUs hosting the threads. The number of threads specified by `OMP_NUM_THREADS` should not exceed the number of cores in the CPU. If neither the `OMP_NUM_THREADS` environment variable nor the `omp_set_num_threads` call is used to set the number of OpenMP threads, the system defaults to 1 thread. For further information, including example OpenMP programs, see the *Cray Application Developer's Environment User's Guide*.

Debugging

The `-g` option provides debugging support for OpenMP directives. The `-g` option, when specified with no optimization options or with `-O0`, provides debugging support identical to specifying the `-G0` option. If any optimization is specified, `-g` is ignored. This level of debugging implies `-homp` which means that most optimizations disabled but OpenMP directives are recognized, and `-h fp0`. To debug without OpenMP, use `-g -xomp` or `-g -hnoomp`, which will disable OpenMP and turn on debugging.

OpenMP Implementation Defined Behavior

The *OpenMP Application Program Interface Specification*, presents a list of implementation defined behaviors. The Cray implementation is described in the following sections.

Atomicity of memory access by multiple threads

When multiple threads access the same shared memory location and at least one thread is a write, threads should be ordered by explicit synchronization to avoid data race conditions and the potential for non-deterministic results. Always use explicit synchronization for any access smaller than one byte.

Internal Control Variables

Table 8. Initial Values of OpenMP ICVs

ICV	Initial Value	Note
`nthreads-var`	1	
`dyn-var`	TRUE	Behaves according to Algorithm 2-1 of the specification.
`run-sched-var`	static	
`stacksize-var`	128 MB	
`wait-policy-var`	ACTIVE	
`thread-limit-var`	64	Threads may be dynamically created up to an upper limit which is 4 times the number of cores/node. It is up to the programmer to try to limit oversubscription.
`max-active-levels-var`	4095	

ICV	Initial Value	Note
def-sched-var	static	The chunksize is rounded up to improve alignment for vectorized loops.

Dynamic Adjustment of Threads

The ICV *dyn-var* is enabled by default. Threads may be dynamically created up to an upper limit which is 4 times the number of cores/node. It is up to the programmer to try to limit oversubscription.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads specified for the parallel region exceeds the number that the runtime system can supply, the program terminates. The number of physical processors actually hosting the threads at any given time is fixed at program startup and is specified by the `aprun -d` *depth* option. The `OMP_NESTED` environment variable and the `omp_set_nested()` call control nested parallelism. To enable nesting, set `OMP_NESTED` to true or use the omp_set_nested() call. Nesting is disabled by default.

Directives and Clauses

- `atomic` directive

  When supported by the target architecture, atomic directives are lowered into hardware atomic instructions. Otherwise, atomicity is guaranteed with a lock. OpenMP `atomic` directives are compatible with C11 and C++11 atomic operations, as well as GNU atomic builtins.

- `do` and `parallel do` directives

  For the `schedule(guided,`*chunk*`)` clause, the size of the initial chunk for the master thread and other team members is approximately equal to the trip count divided by the number of threads.

  For the `schedule(runtime)` clause, the schedule type and, optionally, chunk size can be chosen at runtime by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the default behavior of the `schedule(runtime)` clause is as if the `schedule(static)` clause appeared instead.

  In the absence of the `schedule` clause, the default `schedule` is `static` and the default chunk size is approximately the number of iterations divided by the number of threads.

  The integer type or kind used to compute the iteration count of a collapsed loop are signed 64-bit integers, regardless of how the original induction variables and loop bounds are defined. If the schedule specified by the runtime schedule clause is specified and *run-sched-var* is auto, then the Cray implementation generates a static schedule.

- `parallel` directive

  If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads specified for the parallel region exceeds the number that the runtime system can supply, the program terminates.

  The number of physical processors actually hosting the threads at any given time is fixed at program startup and is specified by the `aprun -d` *depth* option.

  The `OMP_NESTED` environment variable and the `omp_set_nested()` call control nested parallelism. To enable nesting, set `OMP_NESTED` to `true` or use the `omp_set_nested()` call. Nesting is disabled by default.

- `private` clause

  If a variable is declared as private, the variable is referenced in the definition of a statement function, and the statement function is used within the lexical extent of the directive construct, then the statement function references the private version of the variable.

- `sections` construct

  Multiple structured blocks within a single sections construct are scheduled in lexical order and an individual block is assigned to the first thread that reaches it. It is possible for a different thread to execute each section block, or for a single thread to execute multiple section blocks. There is not a guaranteed order of execution of structured blocks within a section.

- `single` directive

  A single block is assigned to the first thread in the team to reach the block; this thread may or may not be the master thread.

- `threadprivate` directive

  The threadprivate directive specifies that variables are replicated, with each thread having its own copy. If the dynamic threads mechanism is enabled, the definition and association status of a thread's copy of the variable is undefined, and the allocation status of an allocatable array is undefined.

- `thread_limit` clause

  The `thread_limit` clause places a limit on the number of threads that a teams construct may create. For NVIDIA GPU accelerator targets, this clause controls the number of CUDA threads per thread block. Only constant integer expressions are supported. If CCE does not support a `thread_limit` expression, then it will issue a warning message indicating the default value that will be used instead.

Library Routines

- `omp_set_num_threads`

  Sets `nthreads-var` to a positive integer. If the argument is < 1, then set `nthreads-var` to 1.

- `omp_set_schedule`

  Sets the schedule type as defined by the current specification. There are no implementation defined schedule types.

- `omp_set_max_active_levels`

  Sets the *`max-active-levels-var`* ICV. Defaults to 4095. If argument is < 1, then set to 1.

- `omp_set_dynamic()`

  The `omp_set_dynamic()` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by setting the value of the *`dyn-var`* ICV. The default is `on`.

- `omp_set_nested()`

  The `omp_set_nested()` routine enables or disables nested parallelism, by setting the *`nest-var`* internal control variable (ICV). The default is `false`.

- `omp_get_max_active_levels`

  There is a single max-active-levels-var ICV for the entire runtime system. Thus, a call to `omp_get_max_active_levels` will bind to all threads, regardless of which thread calls it.

Runtime Library Definitions

It is implementation defined whether the include file omp_lib.h or the module omp_lib (or both) is provided. It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different KIND type can be Fortran accommodated . Both omp_lib.h and the module omp_lib are provided. Cray Fortran uses generic interfaces for routines. If an OMP runtime library routine is defined to be generic, use of arguments of kind other than those specified by `OMP_*_KIND` constants is undefined.

Environment Variables

`OMP_SCHEDULE`

The default value for this environment variable is `static`. For the `schedule(runtime)` clause, the schedule type and, optionally, chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.

`OMP_NUM_THREADS`

If this environment variable is not set and the `omp_set_num_threads()` routine is not used to set the number of OpenMP threads, the default is 1 thread. The maximum number of threads per compute node is 4 times the number of allocated processors. If the requested value of `OMP_NUM_THREADS` is more than the number of threads an implementation can support, the behavior of the program depends on the value of the `OMP_DYNAMIC` environment variable. If `OMP_DYNAMIC` is false, the program terminates. If `OMP_DYNAMIC` is true, it uses up to 4 times the number of allocated processors.

`OMP_PROC_BIND`

When set to `false`, the OpenMP runtime does not attempt to set or change affinity binding for OpenMP threads. When not `false`, this environment variable controls the policy for binding threads to places. Care must be taken when using OpenMP affinity binding with other binding mechanisms. For example, when launching an application with ALPS aprun, the `-cc` cpu affinity binding option (the default) should only be used with `OMP_PROC_BIND=false` or `OMP_PROC_BIND=auto` –otherwise, the ALPS/CLE binding will severely over-constrain OpenMP binding. When setting `OMP_PROC_BIND` to a value other than `false` or `auto`, applications should be launched with `-cc` depth or `-cc` none. Using `-cc` depth is particularly important when running multiple PEs per compute node, since it will allow each PE to bind to CPUs in non-overlapping subsets of the node. Valid values for this environment variable are `true`, `false`, or `auto`; or, a comma-separated list of `spread`, `close`, and `master`. A value of `true` is maps to `spread`.

The default value for `OMP_PROC_BIND` is `auto`, a Cray-specific extension. The `auto` binding policy directs the OpenMP runtime library to select the affinity binding setting that it determines to be most appropriate for a given situation. If there is only a single place in the `place-partition-var` ICV, and that place corresponds to the initial affinity mask of the master thread, then the `auto` binding policy maps to `false` (i.e., binding is disabled). Otherwise, the `auto` binding

	policy causes threads to bind in a manner that partitions the available places across OpenMP threads.
OMP_PLACES	This environment variable has no effect if `OMP_PROC_BIND=false`; when `OMP_PROC_BIND` is not false, then `OMP_PLACES` defines a set of places, or CPU affinity masks, to which threads are bound. When using the `threads`, `cores` and `sockets` keywords, places are constructed according to the CPU topology presented by Linux. However, the place list is always constrained by the initial affinity mask of the master thread. As a result, specific numeric CPU identifiers appearing in `OMP_PLACES` will map onto CPUs in the initial CPU affinity mask. If an application is launched with `-cc` none, then numeric CPU identifiers will exactly match Linux CPU numbers. If instead it is launched with `-cc` depth, then numeric CPU identifier `0` will map to the first CPU in the initial affinity mask for the master thread; identifier 1 will map to the second CPU in the initial mask, and so on. This allows the same `OMP_PLACES` environment variable for all PEs to be used, even when launching multiple PEs per node – the `-cc` depth setting ensures that each PE begins executing with a non-overlapping initial affinity mask, allowing each instance of the OpenMP runtime to assign thread affinity within those non-overlapping affinity masks. The default value of `OMP_PLACES` depends on the value of `OMP_PROC_BIND`. If `OMP_PROC_BIND` is `auto`, then the default value for `OMP_PLACES` is cores. Otherwise, the default value of `OMP_PLACES` is `threads`.
OMP_DYNAMIC	The default value is true.
OMP_NESTED	The default value is false.
OMP_STACKSIZE	The default value for this environment variable is 128 MB.
OMP_WAIT_POLICY	Provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the `wait-policy-var` ICV. Possible values are `ACTIVE` and `PASSIVE`, as defined by the OpenMP specification, and `AUTO`, a Cray-specific extension. The default value for this environment variable is `AUTO`, which direct the OpenMP runtime library to select the most appropriate wait policy for the situation. In general, the `AUTO` policy behaves like `ACTIVE`, unless the number of OpenMP threads or affinity binding results in over subscription of the available hardware processors. If over subscription is detected, the `AUTO` policy behaves like `PASSIVE`.
OMP_MAX_ACTIVE_LEVELS	The default value is 4095.
OMP_THREAD_LIMIT	Sets the number of OpenMP threads to use for the entire OpenMP program by setting the thread-limit-var ICV. The Cray implementation defaults to 4 times the number of available processors.

Cray-specific OpenMP API

This section describes Open MP API specific to Cray.

cray_omp_set_wait_policy

```
subroutine cray_omp_set_wait_policy ( policy )
          character(*), intent(in) :: policy
```

This routine allows dynamic modification of the wait-policy-var ICV value, which corresponds to the `OMP_WAIT_POLICY` environment variable. The policy argument provides a hint to the OpenMP runtime library environment about the desired behavior of waiting threads; acceptable values are `AUTO`, `ACTIVE`, or `PASSIVE` (case insensitive). It is an error to call this routine in an active parallel region. The OpenMP runtime library supports a "wait policy" and a "contention policy," both of which can be set with the following environment variables:

```
OMP_WAIT_POLICY=(AUTO|ACTIVE|PASSIVE)
CRAY_OMP_CONTENTION_POLICY=(Automatic|Standard|MonitorMwait)
```

These environment variables allow the policies to be set once at program launch for the entire execution. However, in some circumstances it would be useful for the programmer to explicitly change the policy at various points during a program's execution. This Cray-specific routine allows the programmer to dynamically change the wait policy (and potentially the contention policy). This addresses the situation when an application needs OpenMP for the first part of program execution, but there is a clear point after which OpenMP is no longer used. Unfortunately, the idle OpenMP threads still consume resources since they are waiting for more work, resulting in performance degradation for the remainder of the application. A passive-waiting policy might eliminate the performance degradation after OpenMP is no longer needed, but the developer may still want an active-waiting policy for the OpenMP-intensive region of the application. This routine notifies all threads of the policy change at the same time, regardless of whether they are idle or active (to avoid deadlock from waiting and signaling threads using different policies).

CRAY_OMP_CHECK_AFFINITY

Set the `CRAY_OMP_CHECK_AFFINITY` variable to `TRUE` at execution time to display affinity binding for each OpenMP thread. The messages contain hostname, process identifier, OS thread identifier, OpenMP thread identifier, and affinity binding.

omp_lib

If the `omp_lib` module is not used and the kind of the actual argument does not match the kind of the dummy argument, the behavior of the procedure is undefined.

omp_get_wtime omp_get_wtick

These procedures return `real(kind=8)` values instead of double precision values.

OpenMP Accelerator Support

The OpenMP 4.5 `target` directives are supported for targeting NVIDIA GPUs or the current CPU target. An appropriate accelerator target module must be loaded to use `target` directives.

When targeting NVIDIA GPUs, teams constructs are mapped to CUDA thread blocks and simd constructs are mapped to CUDA threads within a thread block. For teams regions that do not contain any simd constructs, CCE will still take advantage of all available CUDA parallelism, either by automatically parallelizing nested loops across CUDA threads, or by mapping the teams parallelism across both CUDA thread blocks and threads. Currently, parallel constructs appearing within a teams construct are executed with a single thread. CCE will attempt to

select an appropriate number CUDA threads and thread blocks for each construct based on the code that appears in it. For a given teams construct, users may use the `num_teams` and `thread_limit` clauses to specify the number of CUDA thread blocks and threads per thread block, respectively.

Optimizations

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible, as is shown in the following examples.

Consider the following code:

```
DO K = 1, N
     DO I = 1, N
        DO J = 1, N
           A(I,J) = A(I,J) + B(I,K) * C(K,J)
        END DO
     END DO
END DO
```

For the preceding code fragment, parallelize the `J` loop or the `I` loop. The `K` loop cannot be parallelized because different iterations of the `K` loop read and write the same values of `A(I,J)`. Try to parallelize the outermost `DO` loop if possible, because it encloses the most work. In this example, that is the `I` loop. For this example, use the technique called loop interchange. Although the parallelizable loops are not the outermost ones, the loops can be reordered to make one of them outermost.

Thus, loop interchange would produce the following code fragment:

```
!$OMP PARALLEL DO PRIVATE(I, J, K)
        DO I = 1, N
           DO K = 1, N
              DO J = 1, N
                 A(I,J) = A(I,J) + B(I,K) * C(K,J)
              END DO
           END DO
        END DO
```

Now the parallelizable loop encloses more work and shows better performance. In practice, relatively few loops can be reordered in this way. However, it does occasionally happen that several loops in a nest of loops are candidates for parallelization. In such a case, it is usually best to parallelize the outermost one.

Occasionally, the only loop available to be parallelized has a fairly small amount of work. It may be worthwhile to force certain loops to run without parallelism or to select between a parallel version and a serial version, on the basis of the length of the loop.

The loop is worth parallelizing if `N` is sufficiently large. To overcome the parallel loop overhead, `N` needs to be around 1000, depending on the specific hardware and the context of the program. The optimized version would use an `IF` clause on the `PARALLEL DO` directive:

```
!$OMP PARALLEL DO IF (N .GE. 1000), PRIVATE(I)
        DO I = 1, N
           A(I) = A(I) + X*B(I)
        END DO
```

aprun Options

The `-d` *depth* option of the `aprun` command is required to reserve more than one physical processor for an OpenMP process. For best performance, *depth* should be the same as the maximum number of threads the program uses. The maximum number of threads per compute node is 4 times the number of allocated processors.

This example shows how to reserve the physical processors:

```
aprun -d depth ompProgram
```

If neither the `OMP_NUM_THREADS` environment variable nor the `omp_set_num_threads()` call is used to set the number of OpenMP threads, the system defaults to 1 thread.

The `aprun` options `-n` *processes* and `-N` *processes_per_node* are compatible with OpenMP but do not directly affect the execution of OpenMP programs.

9 OpenACC Use

OpenACC is a parallel programming model which facilitates the use of an accelerator device attached to a host CPU. The OpenACC API allows the programmer to supplement information available to the compilers in order to offload code from a host CPU to an attached accelerator device.

This release supports the *OpenACC Application Programming Interface, Version 2.0* standard developed by PGI, Cray Inc., NVIDIA, with support from CAPS entreprise.

Refer to the OpenACC home page at *http://www.openacc-standard.org*. Under the Downloads link, select the *OpenACC 2.0 Specification*.

For the most current information regarding the Cray implementation of OpenACC, see the `intro_openacc(7)` man page. See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

OpenACC Execution Model

The CPU host offloads compute intensive regions to the accelerator device. The accelerator executes parallel regions, which contain work sharing loops executed as kernels on the accelerator. The CPU host manages execution on the accelerator by allocating memory on the accelerator, initiating data transfer, sending code, passing arguments to the region, waiting for completion, transferring accelerator results back to the CPU host and releasing memory.

The accelerator on the Cray system supports multiple levels of parallelism. The accelerator executes a kernel composed of parallel threads or vectors. Vectors (threads) are grouped into sets called workers. Threads in a set of workers are scheduled together and execute together. Workers are grouped into larger sets called gangs. One or more gangs may comprise a kernel. To summarize, a kernel is executed as a set of gangs of workers of vectors.

The compiler determines the number of gangs/workers/vectors based on the problem and then maps the vectors, workers, and gangs onto the accelerator architecture. Specifying the number of gangs, workers, or vectors is optional but may permit tuning to a particular target architecture. The way that the compiler maps a particular problem onto a constellation of gangs, workers, and vectors which are then mapped onto the accelerator architecture is implementation defined.

OpenACC terminology is situated in the context of the PGAS programming model. In the PGAS model, there may be one or more Processing Elements (PEs) per node. Each PE is multi-threaded and each thread can execute vector instructions. The PGAS thread concept is not the same as the OpenACC thread concept.

OpenACC Memory Model

The memory on the accelerator is separate from host memory. Accelerator device memory is not mapped onto the host's virtual memory space. All data movement between host and accelerator memory is initiated by the host through the library functions that move data. Also, it is not assumed that the accelerator can access host memory, though it is supported by some devices. In this model, data movement between memories is managed by the compiler according to OpenACC directives. The programmer needs to be aware of device memory size, as well as memory bandwidth between host and device in order to effectively accelerate a region of code.

Current accelerators implement a weak memory model; they do not support memory coherence between operations executed by different execution units - an execution unit is a hardware abstraction which can execute one or more gangs. If an operation updates a memory location and another reads from the same location, or two operations store a value to the same location, the hardware may not guarantee repeatable results. Some potential errors of this type are prevented by the compiler, but it is possible to write an accelerator parallel region that produces inconsistent results. Memory coherence is guaranteed when memory operations referencing the same location are separated by an explicit barrier.

Map the OpenACC Programming Model onto Accelerator Components

The compiler maps the OpenACC execution model (kernels, gangs, workers, vectors) onto the accelerator architecture as described in the following sections.

Stream Multiprocessors (SM) and Scalar Processor (SP) cores

On the Cray XC system, there is one accelerator per node. The accelerator architecture is comprised of two main components - global memory and some number of streaming multiprocessors (SM). Each SM contains multiple scalar processor (SP) cores, schedulers, special-function units, and memory which is shared among all the SP cores. An SP core contains floating point, integer, logic, branching, and move and compare units. Each thread/vector is executed by a core. The SM manages thread execution.

The OpenACC execution model maps to the NVIDIA GPU hardware as follows (GPU terms are in parenthesis): One or more OpenACC kernels may execute on an GPU. The compiler divides a kernel into one or more gangs (blocks) of vectors (threads). Several concurrent gangs (blocks) of threads may execute on one SM depending on several factors, including memory requirements, compiler optimizations, or user directives. A single block (gang) does not span SMs and will remain on one SM until completion. When the SM encounters a block (gang), each gang (block) is further broken up into workers (warps) which are groups of threads to execute in parallel. Scheduling occurs at the granularity of the worker (warp). Individual threads within a warp start together and execute one common instruction at a time. If conditional branching occurs within a worker (warp), the warp serially executes each branch path taken causing some threads to wait until threads converge back to the same instruction. Data dependent conditional code within a warp usually has negative performance impact. Worker (warp) threads also fetch data from memory together and when accessing global memory, the accesses of the threads within a warp are grouped to minimize transactions. Each thread in a worker (warp) is executed on a different SP core.

There may be up to 32 threads in a worker (warp) - a limit defined by the hardware.

See the `intro_openacc(7)` man page for more detail on Partition Mapping.

Memory

There is a hierarchy of memory spaces used by OpenACC threads. Each thread has its own private local memory. Each gang of workers of threads has shared memory visible to all threads of the gang. All OpenACC threads running on a GPU have access to the same global memory. Global memory on the accelerator is accessible to the host CPU.

Mixed Model Support

OpenMP directives may appear inside of OpenACC data or host data regions only. OpenMP directives are not allowed inside of any other OpenACC directives.

OpenACC may not appear inside OpenMP directives. To have OpenACC directives nested inside of OpenMP constructs, place them in calls that are not inlined.

Compile with OpenACC

The CCE compiler recognizes OpenACC directives, by default. Use either the `ftn` or `cc` command to compile.

The CCE compiler does not produce CUDA code. It generates PTX (Parallel Thread Execution) instructions which are then translated into assembly.

Note the following interactions between directives and command line options.

- `-x`

  The `-x` option accepts one or more directives as arguments. Directives specified with the `-x` option are ignored during compilation. To ignore all directives, specify `-x all`. To ignore accelerator directives, specify `-x acc`.

- `-h [no]acc`

  `-h noacc` disables OpenACC directives.

- -h acc_model=*`option`* [:*`option`* ...]

  Explicitly controls the execution and memory model utilized by the accelerator support system. The option arguments identify the type of behavior desired. There are three option sets. Only one member of a set may be used at a time; however, all three sets may be used together.

  Default: `auto_async_kernel:fast_addr:no_deep_copy`

  `option` Set 1:

auto_async_none	Execute kernels and updates synchronously, unless there is an async clause present on the kernels or update directive.
auto_async_kernel	(Default) Execute all kernels asynchronously ensuring program order is maintained.
auto_async_all	Execute all kernels and data transfers asynchronously, ensuring program order is maintained.

- *`option`* Set 2:

no_fast_addr	Use default types for addressing.
fast_addr	(Default) Attempt to use 32 bit integers in all addressing to improve performance. Base addresses remain as 64 bit. The performance is improved by potentially using fewer registers and faster arithmetic for offset calculations. This optimization may result in incorrect behavior for codes that make use within accelerator regions of any of the following: very large arrays (offsets would require greater than 32 bits), very large array lower bounds (max offset plus lower bound is greater than 32 bits), bitfields/other bit operations.

- *`option`* Set 3:

no_deep_copy	Do not look inside of an object type to transfer sub-objects. Allocatable members of derived type objects will not be allocated on the device.
deep_copy	Look inside of derived type objects and recreate the derived type on the accelerator recursively. A derived type object that contains an allocatable member will have memory allocated on the device for the member.

Module Support

To compile, ensure that `PrgEnv-cray` module is loaded and that it includes CCE 8.7 or later. Then, load either the `craype-accel-nvidia35` module for Kepler support or the `craype-accel-nvidia60` module for Pascal support.

The `craype-accel-host` module supports compiling and running an OpenACC application on the host X86 processor. This provides source code portability between systems with and without an accelerator. The accelerator directives are automatically converted at compile time to OpenMP equivalent directives.

Use either the `ftn` or `cc` command to compile.

Debug

Use either Allinea DDT or Rogue Wave TotalView.

The following applies to all debuggers:

- To enable debugging, compile use the `-g` option.
- When compiling with the debug option (`-g`), CCE may require additional memory from the accelerator heap, exceeding the 8MB default. In this case, there will be malloc failures during compilation. The environment variable *`CRAY_ACC_MALLOC_HEAPSIZE`* specifies the accelerator heap size in bytes. It may be necessary to increase the accelerator heap size to 32MB (33554432), 64MB (67108864), or greater by setting *`CRAY_ACC_MALLOC_HEAPSIZE`* accordingly. The accelerator heap size defaults to 8MB.
- Debug one rank/image/thread/PE per node.
- CCE does not generate CUDA code, but generates PTX code. Debuggers will not display CUDA intermediate code.
- To enter an OpenACC region using a debugger, breakpoints may be set inside the OpenACC region. It is not possible to do a single step into the region from the code immediately prior to the start of an OpenACC directive.

OpenACC Directives

For information on the OpenACC directives, see the *OpenACC 2.0 Specification* available at *http://www.openacc-standard.org*.

For the most current information regarding the Cray implementation of OpenACC, see the `intro_openacc(7)` man page. See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

Runtime Routines

Runtime routines defined by the standard specification are supported unless otherwise noted in the `intro_openacc(7)` man page.

Extended OpenACC Run Time Library Routines

Extended OpenACC run time library routines are Cray-specific low level routines that give object oriented programmers a mechanism for moving objects from the host CPU to the accelerator and copying memory between the host and the accelerator. These routines are implemented in C. See the `intro_openacc(7)` man page.

Cray Fortran provides a wrapper interface to the C routines using ISO C bindings. To use these routine bindings from Fortran, include the header file openacc_lib.h or use the openacc_lib module. Please see the example "Using_OPENACC_LIB" on the `OpenACC.EXAMPLES(7)` man page.

Environment Variables

The following are environment variables are defined by the API specification:

- `ACC_DEVICE_NUM`
- `ACC_DEVICE_TYPE`

The following environment variable is Cray specific:

- `CRAY_ACC_MALLOC_HEAPSIZE`

  Specifies the accelerator heap size in bytes. The accelerator heap size defaults to 8MB. When compiling with the debug option (`-g`), CCE may require additional memory from the accelerator heap, exceeding the 8MB default. In this case, there will be `malloc` failures during compilation. It may be necessary to increase the accelerator heap size to 32MB (33554432), 64MB (67108864), or greater.

OpenACC Examples

See the `OpenACC.EXAMPLES(7)` man page for example OpenACC codes.

10 Conformance Checks

The amount of error-checking of edit descriptors with input/output (I/O) list items during formatted READ and WRITE statements can be selected through a compiler driver option or through an environment variable.

By default, the compiler provides only limited error-checking.

Use the compiler driver options to choose the table to be used for the conformance check. The table is then part of the executable and no environment variable is required. The compiler driver options allow a choice of checking or no checking with a particular version of the Fortran standard for formatted READ and WRITE. See the following tables: *RELAXED Compatibility Beteween Data Types and Data Edit Descriptors*, *STRICT77 Compatibility Between Data Types and Data Edit Descriptors*, and *STRICT90 and STRICT95 Compatibility Between Data Types and Data Edit Descriptors*.

The environment variable *FORMAT_TYPE_CHECKING* is evaluated during execution. The environment variable overrides a table chosen through the compiler driver option. The environment variable provides an intermediate type of checking that is not provided by the compiler driver option. The environment variable *FORMAT_TYPE_CHECKING* is described in *Set Environment Variables to the Cray Fortran Compiler*.

To select the least amount of checking, use one or more of the following `ftn` command line options.

- On Cray Linux Environment (CLE) systems with formatted READ, use:

```
ftn -W1,--defsym,_RCHK=_RNOCHK *.f(note the double dashes that precede defsym)
```

- On CLE systems with formatted WRITE, use:

```
ftn -W1,--defsym,_WCHK=_WNOCHK *.f
```

- On CLE systems with both formatted READ and WRITE, use:

```
ftn -W1,--defsym,_WCHK=_WNOCHK -W1,--defsym,_RCHK=_RNOCHK *.f
```

To select strict amount of checking for either FORTRAN 77 or Fortran 90, use one or more of the following `ftn` command line options.

- On CLE systems with formatted READ, use:

```
ftn -W1,--defsym,_RCHK=_RCHK77 *.f
ftn -W1,--defsym,_RCHK=_RCHK90 *.f
```

- On CLE systems with formatted WRITE, use:

```
ftn -W1,--defsym,_WCHK=_WCHK77 *.f
ftn -W1,--defsym,_WCHK=_WCHK90 *.f
```

- On CLE systems with both formatted READ and WRITE, use:

```
ftn -W1,--defsym,_WCHK=_WCHK77 -W1,--defsym,_RCHK=_RCHK77 *.f
ftn -W1,--defsym,_WCHK=_WCHK90 -W1,--defsym,_RCHK=_RCHK90 *.f
```

11 Cray Fortran Language Extensions

The Cray Fortran Compiler supports extended features beyond those specified by the current standard. Some of these extensions are widely implemented in other compilers and likely to become standard features in the future, while others are unique and specific to Cray systems. The implementation of any extension may change in order to conform to future language standards.

The listings provided by the compiler identify language extensions when the `-e n` command line option is specified.

128-Bit Precision

The Fortran Compiler supports 128-bit floating point and 256-bit complex predefined types using the X86-64 ABI definitions for type names and data layout. These types are sometimes referred to as "quad-precision". In Fortran, use real(kind=16) and complex(kind=16) to declare variables of these types. In C and C++, use __float128, and __float128 complex.

Fortran and C forms of intrinsic math functions (for example, `QSIN`, `QCOS`, `QTAN`, `QSQRT`, `sinq`, `cosq`, `tanq`) offer full support for quad-precision types. See the `intro_quad_precision(3i)` man page for a complete list of intrinsic functions that support quad-precision.

The base type itself uses 128 bits of storage with a guaranteed minimum alignment on a 128-bit boundary, little endian, has a 15-bit exponent, a 113-bit mantissa, and an exponent bias of 16383, and is compatible with the gcc implementation.

11.1 Characters, Lexical Tokens, and Source Form

Characters Allowed in Names

Variables, named constants, program units, common blocks, procedures, arguments, constructs, derived types (types for structures), namelist groups, structure components, dummy arguments, and function results are among the elements in a program that have a name. As extensions, the Cray Fortran compiler permits the following characters in names:

alphanumeric_character	*currency_symbol*
currency_symbol	$

A name must begin with a letter and can consist of letters, digits, and underscores. The Cray Fortran compiler permits use of the dollar sign ($) in a name, but it cannot be the first character of a name.

Cray does not recommend using $ in user names because it can cause conflicts with the names of internal variables or library routines.

Switch Source Forms

The Cray Fortran compiler allows switching between fixed and free source forms within a source or include file by using the FIXED and FREE compiler directives.

Continuation Line Limit

The Cray Fortran compiler allows a statement to have an unlimited number of continuation lines. The Fortran standard allows only 255 continuation lines.

D Lines in Fixed Source Form

The Cray Fortran compiler allows a D or d character to occur in column one in fixed source form. Typically, the compiler treats a line with a D or d character in column one as a comment line. When the `-e d` command line option is in effect, however, the compiler replaces the D or d character with a blank and treats the rest of the line as a source statement. This can be used, for example, for debugging purposes if the rest of the line contains a PRINT statement.

This functionality is controlled through the `-e d` and `-d d` options on the compiler command line. For more information about these options, see the `ftn(1)` man page.

11.2 Types

The Cray Fortran compiler supports the following additional data types. This preserves compatibility with other vendor's systems.

- Cray pointer
- Cray character pointer
- Boolean (or typeless)

  The Cray Fortran compiler also supports the TYPEALIAS statement as a means of creating alternate names for existing types and supports an expanded form of the ENUM statement.

Alternate Form of LOGICAL Constants

The Cray Fortran compiler accepts .T. and .F. as alternate forms of .true. and .false., respectively.

Cray Pointer Type

The Cray POINTER statement declares one variable to be a Cray pointer (that is, to have the Cray pointer data type) and another variable to be its pointee. The value of the Cray pointer is the address of the pointee. This POINTER statement has the following format:

```
POINTER (pointer_name,  pointee_name  (array_spec) )
, (pointer_name, pointee_name  (array_spec) )  ...
```

pointer_name

Pointer to the corresponding *pointee_name*. *pointer_name* contains the address of *pointee_name*. Only a scalar variable can be declared type Cray pointer; constants, arrays, coarrays, statement functions, and external functions cannot.

pointee_name

Pointee of corresponding *pointer_name*. Must be a variable name, array declarator, or array name. The value of *pointer_name* is used as the address for any reference to *pointee_name*; therefore, *pointee_name* is not assigned storage. If *pointee_name* is an array declarator, it can be explicit-shape (with either constant or nonconstant bounds) or assumed-size.

array_spec

If present, this must be either an *explicit_shape_spec_list*, with either constant or nonconstant bounds) or an *assumed_size_spec*. A codimension used to indicate a coarray may not appear in *array_spec*.

Fortran pointers are declared as follows:

```
POINTER ::  object-name-list
```

Cray Fortran pointers and Fortran standard pointers cannot be mixed.

Example:

```
POINTER(P,B),(Q,C)
```

This statement declares Cray pointer P and its pointee B, and Cray pointer Q and pointee C; the pointer's current value is used as the address of the pointee whenever the pointee is referenced.

An array that is named as a pointee in a Cray POINTER statement is a pointee array. Its array declarator can appear in a separate type or DIMENSION statement or in the pointer list itself. In a subprogram, the dimension declarator can contain references to variables in a common block or to dummy arguments. As with nonconstant bound array arguments to subprograms, the size of each dimension is evaluated on entrance to the subprogram, not when the pointee is referenced. For example:

```
POINTER(IX, X(N,0:M))
```

In addition, pointees must not be deferred-shape or assumed-shape arrays. An assumed-size pointee array is not allowed in a main program unit.

Pointers can be used to access user-managed storage by dynamically associating variables and arrays to particular locations in a block of storage. Cray pointers do not provide convenient manipulation of linked lists because, for optimization purposes, it is assumed that no two pointers have the same value. Cray pointers also allow the accessing of absolute memory locations.

The range of a Cray pointer or Cray character pointer depends on the size of memory for the machine in use.

Restrictions on Cray pointers are as follows:

- A Cray pointer variable should only be used to alias memory locations by using the LOC intrinsic.
- A Cray pointer cannot be pointed to by another Cray or Fortran pointer; that is, a Cray pointer cannot also be a pointee or a target.
- A Cray pointer cannot appear in a PARAMETER statement or in a type declaration statement that includes the PARAMETER attribute.
- A Cray pointer variable cannot be declared to be of any other data type.

- A Cray character pointer cannot appear in a DATA statement.
- An array of Cray pointers is not allowed.
- A Cray pointer cannot be a component of a structure.

Restrictions on Cray pointees are as follows:

- A Cray pointee cannot appear in a SAVE, STATIC, DATA, EQUIVALENCE, COMMON, AUTOMATIC, or PARAMETER statement or Fortran pointer statement.
- A Cray pointee cannot be a dummy argument; that is, it cannot appear in a FUNCTION, SUBROUTINE, or ENTRY statement.
- A function value cannot be a Cray pointee.
- A Cray pointee cannot be a structure component.
- An equivalence object cannot be a Cray pointee.

  Cray pointees can be of type character, but their Cray pointers are different from other Cray pointers; the two kinds cannot be mixed in the same expression.

  The Cray pointer is a variable of type Cray pointer and can appear in a COMMON list or be a dummy argument in a subprogram.

  The Cray pointee does not have an address until the value of the Cray pointer is defined; the pointee is stored starting at the location specified by the pointer. Any change in the value of a Cray pointer causes subsequent references to the corresponding pointee to refer to the new location.

Cray pointers can be assigned values in the following ways:

- A Cray pointer can be set as an absolute address. For example:

```
Q = 0
```

- Cray pointers can have integer expressions added to or subtracted from them and can be assigned to or from integer variables. For example:

```
P = Q + 100
```

  However, Cray pointers are not integers. For example, assigning a Cray pointer to a real variable is not allowed.

  The (nonstandard) LOC intrinsic function generates the address of a variable and can be used to define a Cray pointer, as follows:

```
P = LOC(X)
```

  The following example uses Cray pointers in the ways just described:

```
SUBROUTINE SUB(N)
 INTEGER WORDS
 COMMON POOL(100000), WORDS(1000)
 INTEGER BLK(128), WORD64
 REAL A(1000), B(N), C(100000-N-1000)
 POINTER(PBLK,BLK), (IA,A), (IB,B), &
       (IC,C), (ADDRESS,WORD64)
 ADDRESS = LOC(WORDS) + 64*KIND(WORDS)
 PBLK = LOC(WORDS)
```

```
 IA = LOC(POOL)
 IB = IA + 1000*KIND(POOL)
 IC = IB + N*KIND(POOL)
```

BLK is an array that is another name for the first 128 words of array WORDS. A is an array of length 1000; it is another name for the first 1000 elements of POOL. B follows A and is of length N. C follows B. A, B, and C are associated with POOL. WORD64 is the same as BLK(65) because BLK(1) is at the initial address of WORDS.

If a pointee is of a noncharacter data type that is one machine word or longer, the address stored in a pointer is a word address. If the pointee is of type character or of a data type that is less than one word, the address is a byte address. The following example also uses Cray pointers:

```
PROGRAM TEST
REAL X(*), Y(*), Z(*), A(10)
POINTER (P_X,X)
POINTER (P_Y,Y)
POINTER (P_Z,Z)
INTEGER*8 I,J

!USE LOC INTRINSIC TO SET POINTER MEMORY LOCATIONS
!*** RECOMMENDED USAGE, AS PORTABLE CRAY POINTERS ***
P_X = LOC(A(1))
P_Y = LOC(A(2))

!USE POINTER ARITHMETIC TO DEMONSTRATE COMPILER AND COMPILER
!FLAG DIFFERENCES
!*** USAGE NOT RECOMMENDED, HIGHLY NON-PORTABLE ***
P_Z = P_X + 1

I = P_Y
J = P_Z

IF ( I .EQ. J ) THEN
   PRINT *, 'NOT A BYTE-ADDRESSABLE MACHINE'
ELSE
   PRINT *, 'BYTE-ADDRESSABLE MACHINE'
ENDIF

END
```

On Cray systems, this prints the following:

```
Byte-addressable machine
```

Cray does not recommend the use of pointer arithmetic because it is not portable.

For purposes of optimization, the compiler assumes that the storage of a pointee is never overlaid on the storage of another variable; that is, it assumes that a pointee is not associated with another variable or array. This kind of association occurs when a Cray pointer has two pointees, or when two Cray pointers are given the same value. Although these practices are sometimes used deliberately (such as for equivalencing arrays), results can differ depending on whether optimization is turned on or off. Be responsible for preventing such association. For example:

```
POINTER(P,B), (P,C)
REAL X, B, C
P = LOC(X)
```

```
B = 1.0
C = 2.0
PRINT *, B
```

Because B and C have the same pointer, the assignment of 2.0 to C gives the same value to B; therefore, B will print as 2.0 even though it was assigned 1.0.

As with a variable in common storage, a pointee, pointer, or argument to a LOC intrinsic function is stored in memory before a call to an external procedure and is read out of memory at its next reference. The variable is also stored before a RETURN or END statement of a subprogram.

Cray Character Pointer Type

If a pointee is declared as a character type, its Cray pointer is a Cray character pointer.

Restrictions for Cray pointers also apply to Cray character pointers. In addition, the following restrictions apply:

- When included in an I/O statement iolist, a Cray character pointer is treated as an integer.
- If the length of the pointee is explicitly declared (that is, not of an assumed length), any reference to that pointee uses the explicitly declared length.
- If a pointee is declared with an assumed length (that is, as CHARACTER(*)), the length of the pointee comes from the associated Cray character pointer.
- A Cray character pointer can be used in a relational operation only with another Cray character pointer. Such an operation applies only to the character address and bit offset; the length field is not used.

Boolean Type

A Boolean constant represents the literal constant of a single storage unit. There are no Boolean variables or arrays, and there is no Boolean type statement. Binary, octal, and hexadecimal constants are used to represent Boolean values. For more information about Boolean expressions, see *Expressions and Assignment*.

Alternate Form of ENUM Statement

An enumeration defines the name of a group of related values and the name of each value within the group. The Cray Fortran compiler allows the following additional form for *enum_def* (enumerations):

enum_def_stmt	is	ENUM, ,BIND(C) :: *type_alias_name*
	or	ENUM *kind_selector* :: *type_alias_name*

- *kind_selector*. If it is not specified, the compiler uses the default integer kind.
- *type_alias_name* is the name to assign to the group. This name is treated as a type alias name.

TYPEALIAS Statement

A TYPEALIAS statement allows another name to be defined for an intrinsic data type or user-defined data type. Thus, the type alias and the type specification it aliases are interchangeable. Type aliases do not define a new type.

This is the form for type aliases:

	type_alias_stmt	is	TYPEALIAS :: *type_alias_list*
	type_alias	is	*type_alias_name* => *type_spec*

This example shows how a type alias can define another name for an intrinsic type, a user-defined type, and another type alias:

```
TYPEALIAS :: INTEGER_64 => INTEGER(KIND = 8), &
             TYPE_ALIAS => TYPE(USER_DERIVED_TYPE), &
             ALIAS_OF_TYPE_ALIAS => TYPE(TYPE_ALIAS)

INTEGER(KIND = 8) :: I
TYPE(INTEGER_64) :: X, Y
TYPE(TYPE_ALIAS) :: S
TYPE(ALIAS_OF_TYPE_ALIAS) :: T
```

A type alias or the data type it aliases can be used interchangeably. That is, explicit or implicit declarations that use a type alias have the same effect as if the data type being aliased was used. For example, the above declarations of I, X, and Y are the same. Also, S and T are the same.

If the type being aliased is a derived type, the type alias name can be used to declare a structure constructor for the type.

The following are allowed as the *type_spec* in a TYPEALIAS statement:

- Any intrinsic type defined by the Cray Fortran compiler.
- Any type alias in the same scoping unit.
- Any derived type in the same scoping unit.

11.3 Data Object Declarations and Specifications

The Cray Fortran compiler accepts the following extensions to declarations. The maximum rank is equal to 31. The standard requires a maximum rank of 15.

BOZ Constraints in DATA Statements

The Cray Fortran compiler permits a default real object to be initialized with a BOZ, typeless, or character (used as Hollerith) constant in a DATA statement. BOZ constants are formatted in binary, octal, or hexadecimal. No conversion of the BOZ value, typeless value, or character constant takes place.

The Cray Fortran compiler permits an integer object to be initialized with a BOZ, typeless, or character (used as Hollerith) constant in a type declaration statement. The Cray Fortran compiler also allows an integer object to be initialized with a typeless or character (used as Hollerith) constant in a DATA statement.

If the last item in the *data_object_list* is an array name, the value list can contain fewer values than the number of elements in the array. Any element that is not assigned a value is undefined.

The following alternate forms of BOZ constants are supported:

literal-constant	is	*typeless-constant*
typeless-constant	is	*octal-typeless-constant*
octal-typeless-constant	is	*digit digit*... B
	or	" *digit digit*... "O
	or	' *digit digit*... 'O
hexadecimal-typeless-constant	is	X' *hex-digit hex-digit*... '
	or	X" *hex-digit hex-digit*... "
	or	' *hex-digit hex-digit*... 'X
	or	" *hex-digit hex-digit*... "X

AUTOMATIC Attribute and Statement

The Cray Fortran AUTOMATIC attribute specifies stack-based storage for a variable or array. Such variables and arrays are undefined upon entering and exiting the procedure. The following is the format for the AUTOMATIC specification:

```
type, AUTOMATIC  , attribute-list  :: entity-list
```

automatic-stmt	is	AUTOMATIC :: *entity-list*

entity-list

For *entity-list*, specify a variable name or an array declarator. If an *entity-list* item is an array, it must be declared with an *explicit-shape-spec* with constant bounds. If an *entity-list* item is a pointer, it must be declared with a *deferred-shape-spec*.

If an *entity-list* item has the same name as the function in which it is declared, the *entity-list* item must be scalar and of type integer, real, logical, complex, or double precision.

If the *entity-list* item is a pointer, the AUTOMATIC attribute applies to the pointer itself and not to any target that may become associated with the pointer.

Subject to the rules governing combinations of attributes, *attribute-list* can contain the following:

- DIMENSION
- TARGET
- POINTER
- VOLATILE

The following entities cannot have the AUTOMATIC attribute:

- Pointers or arrays used as function results
- Dummy arguments
- Statement functions
- Automatic array or character data objects

An *entity-list* item cannot have the following characteristics:

- It cannot be defined in the scoping unit of a module.
- It cannot be a common block item.
- It cannot be specified more than once within the same scoping unit.
- It cannot be initialized with a DATA statement or with a type declaration statement.
- It cannot also have the SAVE or STATIC attribute.
- It cannot be specified as a Cray pointee.

IMPLICIT Statement

Implicit Extensions	The Cray Fortran compiler accepts the IMPLICIT AUTOMATIC or IMPLICIT STATIC syntax. It is recommended that none of the IMPLICIT extensions be used in new code.

Storage Association of Data Objects

EQUIVALENCE Statement Extensions	The Cray Fortran compiler allows equivalencing of character data with noncharacter data. The Fortran standard does not address this. It is recommended that equivalencing is not performed in this manner, however, because alignment and padding differs across platforms, thus rendering the code less portable.

COMMON Statement Extensions

The Cray Fortran compiler treats named common blocks and blank common blocks identically, as follows:

- Variables in blank common and variables in named common blocks can be initialized.
- Named common blocks and blank common are always saved.
- Named common blocks of the same name and blank common can be of different sizes in different scoping units.

11.4 Expressions and Assignment

Expressions

In Fortran, calculations are specified by writing expressions. Expressions look much like algebraic formulas in mathematics, particularly when the expressions involve calculations on numerical values.

Expressions often involve nonnumeric values, such as character strings, logical values, or structures; these also can be considered to be formulas that involve nonnumeric quantities rather than numeric ones.

Rules for Forming Expressions

The Cray Fortran compiler supports exclusive disjunct expressions of the form:

exclusive-disjunct-expr	is	*exclusive-disjunct-expr* .XOR*inclusive-disjunct-expr*

Intrinsic and Defined Operations

Cray supports the following intrinsic operators as extensions:

less_greater_op	is	.LG.
	or	<>
not_op	is	.N.
and_op	is	.A.
or_op	is	.O.
exclusive_disjunct_op	is	.XOR.
	or	.X.

The Cray Fortran less than or greater than intrinsic operation is represented by the <> operator and the .LG. keyword. This operation is suggested by the IEEE standard for floating-point arithmetic, and the Cray Fortran compiler supports this operator. Only values of type real can appear on either side of the <> or .LG. operators. If the operands are not of the same kind type value, the compiler converts them to equivalent kind types. The <> and .LG. operators perform a less-than-or-greater-than operation as specified in the IEEE standard for floating-point arithmetic.

The Cray Fortran compiler allows abbreviations for the logical and masking operators. The abbreviations .A., .O., .N., and .X. are synonyms for .AND., .OR., .NOT., and .XOR., respectively.

The masking of Boolean operators and their abbreviations, which are extensions to Fortran, can be redefined as defined operators. If a masking operator is redefined, the definition overrides the intrinsic masking operator definition. See *Bitwise Logical Expressions* for a list of the operators.

Intrinsic Operations

In the following table, the symbols I, R, Z, C, L, B, and P stand for the types integer, real, complex, character, logical, Boolean, and Cray pointer, respectively. Where more than one type for *x*2 is given, the type of the result of the operation is given in the same relative position in the next column. Boolean and Cray pointer types are extensions of the Fortran standard.

Table 9. Operand Types and Results for Intrinsic Operations

Intrinsic operator	Type of *x*1	Type of *x*2	Type of result
Unary +, -		I, R, Z, B, P	I, R, Z, I, P
Binary +, -, *, /, **	I	I, R, Z, B, P	I, R, Z, I, P
	R	I, R, Z, B	R, R, Z, R
	Z	I, R, Z	Z, Z, Z
	B	I, R, B, P	I, R, B, P
	P	I, B, P	P, P, P
	(For Cray pointer, only + and - are allowed.)		

Intrinsic operator	Type of *x1*	Type of *x2*	Type of result
//	C	C	C
.EQ., ==, .NE., /=	I	I, R, Z, B, P	L, L, L, L, L
	R	I, R, Z, B, P	L, L, L, L, L
	Z	I, R, Z, B, P	L, L, L, L, L
	B	I, R, Z, B, P	L, L, L, L, L
	P	I, R, Z, B, P	L, L, L, L, L
	C	C	L
.GT., >, .GE., >=, .LT., <, .LE., <=	I	I, R, B, P	L, L, L, L
	R	I, R, B	L, L, L
	C	C	L
	P	I, P	L, L
.LG., <>	R	R	L
.NOT.		L	L
		I, R, B	B
.AND., .OR., .EQV., .NEQV., .XOR.	L	L	L
	I, R, B	I, R, B	B

The operators .NOT., .AND., .OR., .EQV., and .XOR. can also be used in the Cray Fortran compiler's bitwise masking expressions; these are extensions to the Fortran standard. The result is Boolean (or typeless) and has no kind type parameters.

Bitwise Logical Expressions

A bitwise logical expression (also called a masking expression) is an expression in which a logical operator operates on individual bits within integer, real, Cray pointer, or Boolean operands, giving a result of type Boolean. Each operand is treated as a single storage unit. The result is a single storage unit, which is either 32 or 64 bits depending on the `-s` option specified during compilation. Boolean values and bitwise logical expressions use the same operators but are different from logical values and expressions.

Table 10. Cray Fortran Intrinsic Bitwise Operators and the Allowed Types of their Operands

Operator category	Intrinsic operator	Operand types
Bitwise masking (Boolean) expressions	.NOT., .AND., .OR., .XOR., .EQV., .NEQV.	Integer, real, typeless, or Cray pointer.

Bitwise logical operators can also be written as functions; for example A .AND. B can be written as IAND(A,B) and .NOT. A can be written as NOT(A).

Table 11. Data Types in Bitwise Logical Operations

***x*1 *x*2**[7]	**Integer**	**Real**	**Boolean**	**Pointer**	**Logical**	**Character**
Integer	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Not valid	Not valid[8]
Real	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Not valid	Not valid[8]
Boolean	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Not valid	Not valid[8]
Pointer	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Masking, operation, Boolean result.	Not valid	Not valid[8]
Logical	Not valid[8]	Not valid[8]	Not valid[8]	Not valid[8]	Logical operation logical results	Not valid[8]
Character	Not valid[8]	Not valid[8]	Not valid[8]	Not valid[8]	Not valid	Not valid[8]

Bitwise logical expressions can be combined with expressions of Boolean or other types by using arithmetic, relational, and logical operators. Evaluation of an arithmetic or relational operator processes a bitwise logical expression with no type conversion. Boolean data is never automatically converted to another type.

A bitwise logical expression performs the indicated logical operation separately on each bit. The interpretation of individual bits in bitwise *`multiplication-expr`*s, *`summation-expr`*s, and general expressions is the same as for logical expressions. The results of binary 1 and 0 correspond to the logical results TRUE and FALSE, respectively, in each of the bit positions. These values are summarized as follows:

```
.NOT. 1100          1100          1100          1100          1100
    =0011     .AND. 1010     .OR. 1010    .XOR. 1010    .EQV. 1010
                    ----          ----          ----          ----
                    1000          1110          0110          1001
```

[7] ***x*1 and *x*2 represent operands for a logical or bitwise expression, using operators .NOT., .AND., .OR., .XOR., .NEQV., and .EQV..**

[8] Indicates that if the operand is a character operand of 32 or fewer characters, the operand is treated as a Hollerith constant and is allowed.

Assignment

The Cray Fortran compiler supports Boolean and Cray pointer intrinsic assignments. The Cray Fortran compiler supports type Boolean or BOZ constants in assignment statements in which the variable is of type integer or real. The bits specified by the constant are moved into the variable with no type conversion.

Array Reference

The Cray Fortran compiler allows arrays to be referenced with fewer than the declared number of dimensions. The subscripts specified in the array reference are used for the leftmost dimensions, and the lower bounds are used for the rightmost subscripts that were omitted. This extension to the Fortran standard applies to both arrays and coarrays.

When the option to note deviations from the Fortran standard is in effect (-en), this type of an array reference will cause compilation errors.

11.5 Input/Output Statements

The Fortran standard does not specifically describe the implementation of I/O processing. This section provides information about processor-dependent areas and the implementation of the support for I/O.

File Connection

OPEN Statement

The OPEN statement specifies the connection properties between the file and the unit. The *Values for Keyword Specifier Variables in an OPEN Statement* table indicates the keyword specifiers in an OPEN statement that are Cray Fortran compiler extensions.

Table 12. Values for Keyword Specifier Variables in an OPEN Statement

Specifier	Possible Values	Default Value
FORM	SYSTEM	Unformatted with no records marks
CONVERT	LITTLE_ENDIAN, BIG_ENDIAN, CRAY, NATIVE	NATIVE

The FORM specifier has the following format:FORM= *scalar-char-expr*

A file opened with FORM="SYSTEM" is unformatted and has no record marks.

The CONVERT specifier converts unformatted data between BIG- and LITTLE-ENDIAN representation. Overrides any numeric conversion specified via assign or by compilation option.

The CONVERT specifier has the following format:CONVERT="*format-specifier*"*format-specifier* describes the format of the file being opened and it only applies for that single file. It may be one of the following strings:

- LITTLE_ENDIAN
- Specifies little endian integer data and IEEE floating-point data. Has no effect except to override any numeric conversion specified via assign statement or by compilation option.

- BIG_ENDIAN
- Specifies big endian integer data and IEEE floating-point data. This has the same effect as specifying `-hbyteswapio` on the compilation, but it applies on a per file basis. The assign -Nswap_endian f:*`filename`* command also converts the named file to BIG_ENDIAN format.
- CRAY
- Indicates BIG_ENDIAN integer data and Cray floating point data of size REAL(8) or COMPLEX(8). It has the same effect as the `assign` command: assign -Ncray f:*`filename`*.
- NATIVE
- Default. Same effect as "LITTLE_ENDIAN".

11.6 Error, End-of-record, and End-of-file Conditions

End-of-file Condition and the END-specifier

Multiple End-of-file Records

The file position prior to data transfer depends on the method of access: sequential or direct. Although the Fortran standard does not allow files that contain an end-of-file to be positioned after the end-of-file prior to data transfer, the Cray Fortran compiler permits more than one end-of-file for some file structures.

11.7 Input/Output Editing

Data Edit Descriptors

Integer Editing

The Cray Fortran compiler allows *w* to be zero for the G edit descriptor, and it permits *w* to be omitted for the I, B, O, Z, or G edit descriptors.

The Cray Fortran compiler allows signed binary, octal, or hexadecimal values as input.

If the minimum digits (*m*) field is specified, the default field width is increased, if necessary, to allow for that minimum width.

Real Editing

The Cray Fortran compiler allows the use of B, O, and Z edit descriptors of REAL data items. The Cray Fortran compiler accepts the D*w*.*d*E*e* edit descriptor.

The Cray Fortran compiler accepts the *`ZERO_WIDTH_PRECISION`* environment variable, which can be used to modify the default size of the width w field. This environment variable is examined only upon program startup. Changing the value of the environment variable during program execution has no effect. For more information about the *`ZERO_WIDTH_PRECISION`* environment, see *ZERO_WIDTH_PRECISION*.

The Cray Fortran compiler allows *w* to be zero or omitted for the D, E, EN, ES, or G edit descriptors.

The Cray Fortran compiler does not restrict the use of E*w*.*d* and D*w*.*d* to an exponent less than or equal to 999. The E*w*.*d*E*e* form must be used.

Table 13. Default Fractional and Exponent Digits

Data Size and Representation	*w*	*d*	*e*
4-byte (32-bit) IEEE	17	9	2
8-byte (64-bit) IEEE	26	17	3

Logical Editing

The Cray Fortran compiler allows *w* to be zero or omitted on the L or G edit descriptors.

Character Editing

The Cray Fortran compiler allows *w* to be zero or omitted on the G edit descriptor.

Control Edit Descriptors

Q Editing

The Cray Fortran supports the Q edit descriptor. The Q edit descriptor is used to determine the number of characters remaining in the input record. It has the following format:Q

When a Q edit descriptor is encountered during execution of an input statement, the corresponding input list item must be of type integer. Interpretation of the Q edit descriptor causes the input list item to be defined with a value that represents the number of characters remaining to be read in the formatted record.

For example, if *c* is the character position within the current record of the next character to be read, and the record consists of *n* characters, then the item is defined with the following value MAX(*n*-*c*+1,0).

If no characters have yet been read, then the item is defined as *n* (the length of the record). If all the characters of the record have been read (*c*>*n*), then the item is defined as zero.

The Q edit descriptor must not be encountered during the execution of an output statement.

The following example code uses Q on input:

```
INTEGER N
CHARACTER LINE * 80
READ (*, FMT='(Q,A)') N, LINE(1:N)
```

List-directed Formatting

List-directed Input

Input values are generally accepted as list-directed input if they are the same as those required for explicit formatting with an edit descriptor. The exceptions are as follows:

- When the data list item is of type integer, the constant must be of a form suitable for the I edit descriptor. The Cray Fortran compiler permits binary, octal, and hexadecimal based values in a list-directed input record to correspond to I edit descriptors.

Namelist Formatting

Namelist Extensions

The Cray Fortran compiler has extended the namelist feature. The following additional rules govern namelist processing:

- An ampersand (&) or dollar sign ($) can precede the namelist group name or terminate namelist group input. If an ampersand precedes the namelist group name, either the slash (/) or the ampersand must terminate the namelist group input. If the dollar sign precedes the namelist group name, either the slash or the dollar sign must terminate the namelist group input.
- Octal and hexadecimal constants are allowed as input to integer and single-precision real namelist group items. An error is generated if octal and hexadecimal constants are specified as input to character, complex, or double-precision real namelist group items.
- Octal constants must be of the following form:
  - O"123"
  - O'123'
  - o"123"
  - o'123'
- Hexadecimal constants must be of the following form:
  - Z"1a3"
  - Z'1a3'
  - z"1a3"
  - z'1a3'

I/O Editing

Usually, data is stored in memory as the values of variables in some binary form. On the other hand, formatted data records in a file consist of characters. Thus, when data is read from a formatted record, it must be converted from characters to the internal representation. When data is written to a formatted record, it must be converted from the internal representation into a string of characters.

The tables below list the control and data edit descriptor extensions supported by the Cray Fortran compiler and provide a brief description of each.

Table 14. Summary of Control Edit Descriptors

Descriptor	Description
$ or \	Suppress carriage control

Table 15. Summary of Data Edit Descriptors

Descriptor	Description
Q	Return number of characters left in record

The following tables show the use of the Cray Fortran compiler's edit descriptors with all intrinsic data types. In these tables:

- NA indicates invalid usage that is not allowed.
- I,O indicates that usage is allowed for both input and output.
- I indicates legal usage for input only.
- NA indicates invalid usage that is not allowed.
- I,O indicates that usage is allowed for both input and output.
- I indicates legal usage for input only.

Table 16. Default Compatibility Between I/O List Data Types and Data Edit Descriptors

Data types	Q	Z	R	O	L	I	G	F	ES	EN	E	D	B	A
Integer	I	I,O	I,O	I,O	NA	I,O	I,O	NA	NA	NA	NA	NA	I,O	I,O
Real	NA	I,O	I,O	I,O	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Complex	NA	I,O	I,O	I,O	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Logical	NA	I,O	I,O	I,O	I,O	NA	I,O	NA	NA	NA	NA	NA	I,O	I,O
Character	NA	NA	NA	NA	NA	NA	I,O	NA	NA	NA	NA	NA	NA	I,O

The table below, *RELAXED Compatibility Between Data Types and Data Edit Descriptors* shows the restrictions for the various data types that are allowed when the `FORMAT_TYPE_CHECKING` environment variable is set to RELAXED. Not all data edit descriptors support all data sizes; for example, a 16-byte real variable with an I edit descriptor cannot be read/write.

Table 17. RELAXED Compatibility Between Data Types and Data Edit Descriptors

Data types	Q	Z	R	O	L	I	G	F	ES	EN	E	D	B	A
Integer	I	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	NA	I,O	I,O
Real	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Complex	NA	I,O	I,O	I,O	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Logical	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	NA	I,O	I,O

Data types	Q	Z	R	O	L	I	G	F	ES	EN	E	D	B	A
Character	NA	NA	NA	NA	NA	NA	I,O	NA	NA	NA	NA	NA	NA	I,O

STRICT77 Compatibility Between Data Types and Data Edit Descriptors shows the restrictions for the various data types that are allowed when the `FORMAT_TYPE_CHECKING` environment variable is set to STRICT77.

Table 18. STRICT77 Compatibility Between Data Types and Data Edit Descriptors

Data types	Q	Z	R	O	L	I	G	F	ES	EN	E	D	B	A
Integer	NA	I,O	NA	I,O	NA	I,O	NA	NA	NA	NA	NA	NA	I,O	NA
Real	NA	NA	NA	NA	NA	NA	I,O	I,O	NA	NA	I,O	I,O	NA	NA
Complex	NA	NA	NA	NA	NA	NA	I,O	I,O	NA	NA	I,O	I,O	NA	NA
Logical	NA	NA	NA	NA	I,O	NA	NA	NA	NA	NA	NA	NA	NA	NA
Character	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	I,O

STRICT90 and STRICT95 Compatibility Between Data Types and Data Edit Descriptors shows the restrictions for the various data types that are allowed when the `FORMAT_TYPE_CHECKING` environment variable is set to STRICT90 or STRICT95.

Table 19. STRICT90 or STRICT95 Compatibility Between Data Types and Data Edit Descriptors

Data types	Q	Z	R	O	L	I	G	F	ES	EN	E	D	B	A
Integer	NA	I,O	NA	I,O	NA	I,O	I,O	NA	NA	NA	NA	NA	I,O	NA
Real	NA	NA	NA	NA	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O	NA	NA
Complex	NA	NA	NA	NA	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O	NA	NA
Logical	NA	NA	NA	NA	I,O	NA	I,O	NA	NA	NA	NA	NA	NA	NA
Character	NA	NA	NA	NA	NA	NA	I,O	NA	NA	NA	NA	NA	NA	I,O

11.8 Program Units

Main Program

Program Statement Extension	The Cray Fortran compiler supports the use of a parenthesized list of *args* at the end of a program statement. The compiler ignores any *arg*s specified after *program-name*.

Block Data Program Units

Block Data Program Unit Extension	The Cray Fortran compiler permits named common blocks to appear in more than one block data program unit.

11.9 Procedures

Procedure Interface

Interface Duplication	The Cray Fortran compiler allows specification of an interface body for the program unit being compiled if the interface body matches the program unit definition.

Procedure Definition

Recursive Function Extension	The Cray Fortran compiler allows direct recursion for functions that do not specify a RESULT clause on the FUNCTION statement.
Empty CONTAINS Sections	The Cray Fortran compiler allows a CONTAINS statement with no internal or module procedure following.

11.10 Intrinsic Procedures and Modules

Intrinsic Procedures

The Cray Fortran compiler has implemented intrinsic procedures in addition to the ones required by the standard. These procedures have the status of intrinsic procedures, but programs that use them may not be portable. It is recommended that such procedures be declared INTRINSIC to allow other processors to diagnose whether or not they are intrinsic for those processors.

The nonstandard intrinsic procedures supported by the Cray Fortran compiler are summarized in the following list. For more information about a particular procedure, see its man page.

ACOSD	Arccosine, value in degrees

AMO_AADD	Atomic memory add
AMO_AADDF	Atomic memory add, return new
AMO_AFADD	Atomic memory add, return old
AMO_AAX	Atomic memory AND and XOR
AMO_AFAX	Atomic memory AND and XOR, return old
AMO_AANDF	Atomic memory AND, return new
AMO_AFAND	Atomic memory AND, return old
AMO_ANANDF	Atomic memory NAND, return new
AMO_AFNAND	Atomic memory NAND, return old
AMO_AORF	Atomic memory OR, return new
AMO_AFOR	Atomic memory OR, return old
AMO_AXORF	Atomic memory XOR, return new
AMO_AFXOR	Atomic memory XOR, return old
AMO_ACSWAP	Atomic memory swap, return old
AMO_ASWAP	Atomic memory swap, return new
AMO_AFLUSH	Atomic memory flush, flush local cache to the global address space for a particular address on a Cray XC system.
ASIND	Arcsine, value in degrees
ATAND	Arctangent, value in degrees
ATAND2	Arctangent, value in degrees
CO_BCAST	Broadcast a coarray to all images in an application.
CO_SUM	Sum of corresponding elements on all images in a coarray application
CO_MIN, CO_MAX	Maximum or minimum value of corresponding elements on all images in a coarray application
COSD	Cosine, argument in degrees
COT	Cotangent
EXIT	Program termination
FREE	Free Cray pointee memory
GET_BORROW_S@	Get scalar borrow bit
GSYNC	Complete outstanding memory references
IBCHNG	Reverse bit within a word
ILEN	Length in bits of an integer
INT_MULT_UPPER	Upper bits of integer product
LOC	Address of argument

MALLOC	Allocate Cray pointee memory
MASK	Creates a bit mask in a word
SET_BORROW_S@	Set scalar borrow bits
SET_CARRY_S@	Set scalar carry bits
SIND	Sin, argument in degrees
SIZEOF	Size of argument in bytes
SUB_BORROW_S@	Subtract scalar with borrow
TAND	Tangent, argument in degrees

CO_BCAST, CO_SUM, CO_MIN, and CO_MAX are collective intrinsic subroutines, which are extensions of the Fortran 2008 standard. Support for teams is deferred. For specific information about these routines, see the `co_bcast(3i)`, `co_max(3i)`, `co_sum(3i)` man pages.

Many intrinsic procedures have both a vector and a scalar version. If a vector version of an intrinsic procedure exists, and the intrinsic is called within a vectorizable loop, the compiler uses the vector version of the intrinsic. For information about which intrinsic procedures vectorize, see the `intro_intrin(3i)` man page.

For more information about the atomic memory intrinsic procedures see the `amo(3i)` man page.

11.11 Exceptions and IEEE Arithmetic

The Exceptions

The intrinsic module IEEE_EXCEPTIONS supplied with the Cray Fortran compiler contains three named constants in addition to those specified by the standard. These are of type IEEE_STATUS_TYPE and can be used as arguments to the IEEE_SET_STATUS subroutine. Their definitions correspond to common combinations of settings and allow for simple and fast changes to the IEEE mode settings. The constants are:

Table 20. Cray Fortran IEEE Intrinsic Module Extensions

Name	Effect of CALL IEEE_SET_STATUS (Name)
ieee_cri_nostop_mode	• Clears all currently set exception flags • Disables halting for all exceptions • Enables setting of all exception flags • Sets rounding mode to round_to_nearest
ieee_cri_default_mode	• Clears all currently set exception flags • Enables halting for overflow, divide_by_zero, and invalid • Disables halting for underflow and inexact • Enables setting of all exception flags

Name	Effect of CALL IEEE_SET_STATUS (Name)
	• Sets rounding mode to round_to_nearest

11.12 Compile and Execute Programs Containing Coarrays

There are various commands, tools, and products available in the programming environment to use for compiling and executing programs containing coarrays.

`ftn` and `aprun` Options Affecting Coarrays

The compiler recognizes coarray syntax by default. The `-h nocaf` disables coarray syntax recognition.

Upon execution of an `a.out` file that has been compiled and linked with the `-h caf` option, an image is created and executed on every processing element assigned to the job. Images 1 through `NUM_IMAGES` are assigned to processing elements 0 through N$PES-1, consecutively. The functions `THIS_IMAGE()` and `NUM_IMAGES()` may be used to retrieve the image number of the current image, or the total number of images at run time, respectively.

Set the number of processing elements assigned to a job at compile time by specifying the -X option on the `ftn` command. The number of processing elements can also be set at run time by executing the `a.out` file by using the `aprun` command with the -n option specified. If mixed `-X` values are used when compiling and linking different object files, or the number of PEs specified at run time differs from that specified when compiling and linking, a run time error will be received.

Bounds checking is performed by specifying the -Rb option on the `ftn` command line. This feature is not implemented for codimensions of coarrays.

For more information about the `ftn` and `aprun` commands, see the `ftn(1)` and `aprun(1)` man pages.

Interoperate with Other Message Passing and Data Passing Models

Coarrays can interoperate with all other message and data passing models. This allows for the introduction of coarrays into existing application codes incrementally. However, while it may work in some cases, mixing language-based PGAS with SHMEM is not officially supported.

These models are implemented through procedure calls, so the language interaction between coarrays and these models is well defined.

MPI and SHMEM generally use processing element numbers, which start at zero, but the coarray model generally deals with image numbers, which start at one.

Coarrays are symmetric for the purposes of SHMEM programming. Pointers in coarrays of derived type, however, may not necessarily point to symmetric data.

For more information about the other message passing and data passing models, see the following man pages:

- `intro_mpi(3)`
- `intro_shmem(3)`

Optimize Programs with Coarrays

Programs containing coarrays benefit from all the usual steps taken to improve run time performance of code that runs on a single image.

12 Cray Fortran Deferred Implementation and Optional Features

ISO_10646 Character Set

The Fortran 2003 features related to supporting the ISO_10646 character set are not supported. This includes declarations, constants, and operations on variables of character(kind=4) and I/O operations. Support for this feature is optional in Fortran 2008.

Restrictions on Unlimited Polymorphic Variables

Unlimited polymorphic variables whose dynamic types are integer(1), integer(2), logical(1), or logical(2) are not supported, unless the `-d h` is specified to disable packed storage for short integers and logicals.

13 Cray Fortran Implementation Specifics

The Fortran standard specifies the rules for writing a standard conforming Fortran program. Many of the details of how such a program is compiled and executed are intentionally not specified or are explicitly specified as being processor-dependent. This chapter describes the implementation used by the Cray Fortran compiler. Included are descriptions of the internal representations used for data objects and the values of processor-dependent language parameters.

Companion Processor

For the purpose of C interoperability, the Fortran standard refers to a companion processor. The companion processor for the Cray Fortran compiler is the Cray C compiler.

INCLUDE Line

There is no limit to the nesting level for INCLUDE lines. The character literal constant in an INCLUDE line is interpreted as the name of the file to be included. This case-sensitive name may be prefixed with additional characters based on the `-I` compiler command line option.

INTEGER Kinds and Values

INTEGER kind type parameters of 1, 2, 4, and 8 are supported. The default kind type parameter is 4 unless the `-s default64` or `-s integer64` command line option is specified, in which case the default kind type parameter is 8. The interpretation of kinds 1 and 2 depend on whether the `-d h` command line option is specified. Integer values are represented as two's complement binary values.

REAL Kinds and Values

REAL kind type parameters of 4, 8, and 16 are supported. The default kind type parameter is 4 unless the `-s default64` or `-s real64` command lines option is specified, in which case, the default kind type parameter is 8. Real values are represented in the format specified by the IEEE 754 standard, with kinds 4 and 8 corresponding to the 32 and 64 bit IEEE representations.

DOUBLE PRECISION Kinds and Values

The DOUBLE PRECISION type is an alternate specification of a REAL type. The kind type parameter of that REAL type is twice the value of the kind type parameter for default REAL unless the `-sdefault64` or `-sreal64` command line options are specified, in which case, the kind type parameter for DOUBLE PRECISION and default REAL are the same, and REAL constants with a D exponent are treated as if the D were an E. Note that if the `-sdefault64` or `-sreal64` options are specified, the compiler is not standard conforming.

LOGICAL Kinds and Values

LOGICAL kind type parameters of 1, 2, 4, and 8 are supported. The default kind type parameter is 4 unless the `-s default64` or `-s integer64` command line option is specified, in which case, the default kind type

parameter is 8. The interpretation of kinds 1 and 2 depend on whether the `-d h` command line option is specified. Logical values are represented by a bit sequence in which the low order bit is set to 1 for the value .true. and to 0 for .false., and the other bits in the representation are set to 0.

CHARACTER Kinds and Values

The CHARACTER kind type parameter of 1 is supported. The default kind type parameter is 1. Character values are represented using the 8-bit ASCII character encoding.

Cray Pointers

Cray pointers are 64-bit objects.

ENUM Kind

An enumerator that specifies the BIND(C) attribute creates values with a kind type parameter of 4.

Storage Issues

This section describes how the Cray Fortran compiler uses storage, including how this compiler accommodates programs that use overindexing of blank common.

Storage Units and Sequences

The size of the numeric storage units is 32 bits, unless the `-s default64` option is specified, in which case the numeric storage unit is 64 bits. If the `-s real64` or `-s integer64` option is specified alone, or the `-dp` is specified in addition to `-s default64` or `-s real64`, the relative sizes of the storage assigned for default intrinsic types do not conform to the standard. In this case, storage sequence associations involving variables declared with default intrinsic noncharacter types may be invalid and should be avoided.

Static and Stack Storage

The Cray Fortran compiler allocates variables to storage according to the following criteria:

- Variables in common blocks are always allocated in the order in which they appear in COMMON statements.
- Data in modules are statically allocated.
- User variables that are defined or referenced in a program unit, and that also appear in SAVE or DATA statements, are allocated to static storage, but not necessarily in the order shown in the source program.
- Other referenced user variables are assigned to the stack. If -ev is specified on the Cray Fortran compiler command line, referenced variables are allocated to static storage. This allocation does not necessarily depend on the order in which the variables appear in the source program.
- Compiler-generated variables are assigned to a register or to memory (to the stack or heap), depending on how the variable is used. Compiler-generated variables include DO-loop trip counts, dummy argument addresses, temporaries used in expression evaluation, argument lists, and variables storing adjustable dimension bounds at entries.
- Automatic objects may be allocated to either the stack or to the heap, depending on how much stack space is available when the objects are allocated.
- Heap or stack allocation can be used for TASK COMMON variables and some compiler-generated temporary data such as automatic arrays and array temporaries.

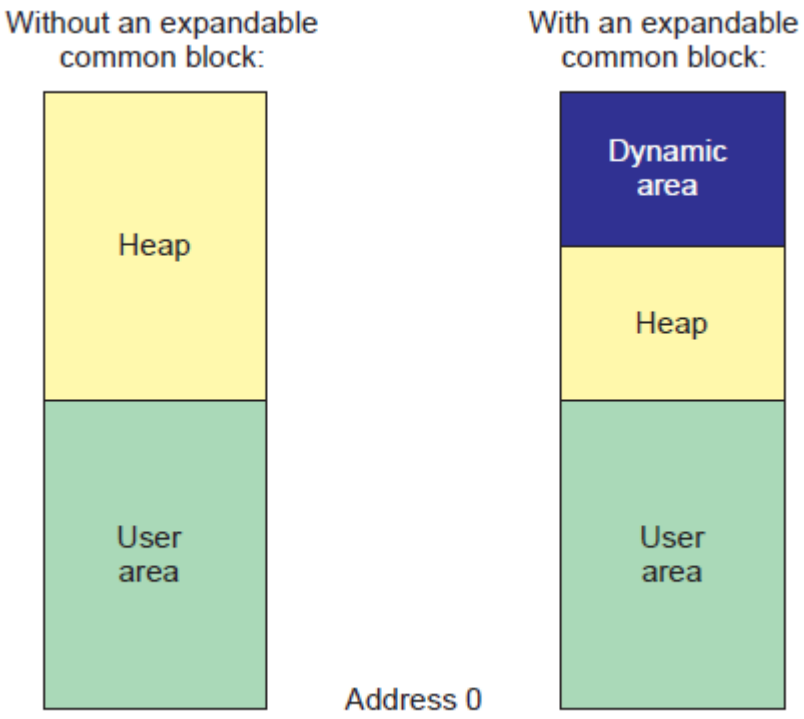
- Unsaved variables may be assigned to a register by optimization and not allocated storage.
- Unreferenced user variables not appearing in COMMON statements are not allocated storage.

Dynamic Memory Allocation

Many FORTRAN 77 programs contain a memory allocation scheme that expands an array in a common block located in central memory at the end of the program. This practice of expanding a blank common block or expanding a dynamic common block (sometimes referred to as *overindexing*) causes conflicts between user management of memory and the dynamic memory requirements of CLE libraries. It is recommended that programs are modified rather than expand blank common blocks, particularly when migrating from other environments.

The image below shows the structure of a program under the CLE operating systems in relation to expanding a blank common block. In both figures, the user area includes code, data and common blocks.

Figure 1. Memory Use



Finalization

A finalizable object in a module is not finalized in the event that there is no longer any active procedure referencing the module.

A finalizable object that is allocated via pointer allocation is not finalized in the event that it later becomes unreachable due to all pointers to that object having their pointer association status changed.

ALLOCATE Error Status

If an error occurs during the execution of an ALLOCATE statement with a stat= specifier, subsequent items in the allocation list are not allocated.

DEALLOCATE Error Status

If an error occurs during the execution of a DEALLOCATE statement with a stat= specifier, subsequent items in the deallocation list are not deallocated.

ALLOCATABLE Module Variable Status

An unsaved allocatable module variable remains allocated if it is allocated when the execution of an END or RETURN statement results in no active program unit having access to the module.

Kind of a Logical Expression

For an expression such as x1 *op* x2 where *op* is a logical intrinsic binary operator and the operands are of type logical with different kind type parameters, the kind type parameter of the result is the larger kind type parameter of the operands.

STOP Code Availability

If a STOP code is specified in a STOP statement, its value is output to stderr when the STOP statement is executed.

When the stop code is a string of digits, only the least-significant 8 bits of the integer value is used as the process exit status. When the stop code is of type character or does not appear, the value zero is the process exit status.

Stream File Record Structure Position

A formatted file written with stream access may be later read as a record file. In that case, embedded newline characters (char(10)) indicate the end of a record and the terminating newline character is not considered part of the record.

The file storage unit for a formatted stream file is a byte. The position is the ordinal byte number in the file; the first byte is position 1. Positions corresponding to newline characters (char(10)) that were inserted by the I/O library as part of record output do not correspond to positions of user-written data.

File Unit Numbers

The values of INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT defined in the ISO_Fortran_env module are 100, 101, and 102, respectively. These three unit numbers are reserved and may not be used for other purposes. The files connected to these units are the same files used by the companion C processor for standard input (stdin), output (stdout), and error (stderr). An asterisk (*) specified as the unit for a READ statement specifies unit 100. An asterisk specified as the unit for a WRITE statement, and the unit for PRINT statements is unit 101. All positive default integer values are available for use as unit numbers.

OPEN Specifiers

If the ACTION= specifier is omitted from an OPEN statement, the default value is determined by the protections associated with the file. If both reading and writing are permitted, the default value is READWRITE.

If the ENCODING= specifier is omitted or specified as DEFAULT in an OPEN statement for a formatted file, the encoding used is ASCII.

The case of the name specified in a FILE= specifier in an OPEN statement is significant.

If the FILE= specifier is omitted, fort. is prepended to the unit number.

If the RECL= specifier is omitted from an OPEN statement for a sequential access file, the default value for the maximum record length is 32767 (2**15-1).

If the file is connected for unformatted I/O, the length is measured in 8-bit bytes.

The FORM= specifier may also be SYSTEM for unformatted files.

If the ROUND= specifier is omitted from an OPEN statement, the default value is NEAREST. Specifying a value of PROCESSOR_DEFINED is equivalent to specifying NEAREST.

If the STATUS= specifier is omitted or specified as UNKNOWN in an OPEN statement, the specification is equivalent to OLD if the file exists, otherwise, it is equivalent to NEW. If STATUS="SCRATCH" is specified the file is placed in the directory specified by the *TMPDIR* environment variable. If *TMPDIR* is not set, or the file cannot be created in the specified directory for some other reason, the file is placed in the `/tmp` directory. If `/tmp` does not exist, or cannot be accessed, the program aborts.

FLUSH Statement

Execution of a FLUSH statement causes memory resident buffers to be flushed to the physical file. Output to the unit specified by ERROR_UNIT in the ISO_Fortran_env module is never buffered; execution of FLUSH on that unit has no effect.

Asynchronous I/O

The ASYNCHRONOUS= specifier may be set to YES to allow asynchronous I/O for a unit or file.

Asynchronous I/O is used if the FFIO layer attached to the file provides asynchronous access.

REAL I/O of an IEEE NaN

An IEEE NaN may be used as an I/O value for the F, E, D, or G edit descriptor or for list-directed or namelist I/O.

Input of an IEEE NaN

The form of NaN is an optional sign followed by the string 'NAN' optionally followed by a hexadecimal digit string enclosed in parentheses. The input is case insensitive. Some examples are:

```
NaN                 - quiet NaN
nAN()               - quiet NaN
-nan(ffffffff)      - quiet NaN
NAn(7f800001)       - signalling NaN
NaN(ffc00001)       - quiet NaN
NaN(ff800001)       - signalling NaN
```

The internal value for the NaN becomes a quiet NaN if the hexadecimal string is not present or is not a valid NaN.

A '+' or '-' preceding the NaN on input is used as the high order bit of the corresponding READ input list item. An explicit sign overrides the sign bit from the hexadecimal string. The internal value becomes the hexadecimal string if it represents an IEEE NaN in the internal data type. Otherwise, the form of the internal value is undefined.

Output of an IEEE NaN

The form of an IEEE NaN for the F, E, D, or G edit descriptor or for list-directed or namelist output is:

- If the field width w is absent, zero, or greater than (5 + 1/4 of the size of the internal value in bits), the output consists of the string 'NaN' followed by the hexadecimal representation of the internal value within a set of parentheses. An example of the output field is:

  ```
  NaN(7fc00000)
  ```

- If the field width w is at least 3 but less than (5 + 1/4 of the size of the internal value in bits), the string 'NaN' will be right-justified in the field with blank fill on the left.
- If the field width w is 1 or 2, the field is filled with asterisks.

The output field has no '+' or '-'; the sign is contained in the hexadecimal string.

To get the same internal value for a NaN, write it with a list-directed write statement and read it with a list-directed read statement.

To write and then read the same NaN, the field width w in D, E, F, or G must be at least the number of hexadecimal digits of the internal datum plus 5.

```
REAL(4):   w >= 13
REAL(8):   w >= 21
REAL(16):  w >= 37
```

List-directed and NAMELIST Output Default Formats

The length of the output value in NAMELIST and list-directed output depends on the value being written. Blanks and unnecessary trailing zeroes are removed unless the `-w` option to the `assign` command is specified, which turns off this compression.

By default, full-precision printing is assumed unless a precision is specified by the *`LISTIO_PRECISION`* environment variable (for more information about the *`LISTIO_PRECISION`* environment variable, see *LISTIO_PRECISION*).

The form of list-directed and NAMELIST output can be changed by using the `assign` command with one of the following options.

Table 21. List-directed and NAMELIST Assign Environment Options

`assign` Option	**Effect**
`-S`	Suppress comma-delimited output; use blank spaces instead
`-W`	Disable compression of floating-point values
`-y`	Disable the repeat-count form; write as many copies of the value as needed
`-U`	Set all three of the above

For example, consider this code:

```
integer(4), dimension(5) :: ia
real(4), dimension(5) :: ra
ia = 102
ra = 200.10
NAMELIST/TNAMEL/ia,ra
write(6,TNAMEL)
print *, ' ia=',ia
print *, ' ra=',ra
print *, iarray, rarray
end
```

When compiled and executed with the default settings, it produces the following output:

```
&TNAMEL  RA = 2*200.100006, IA = 2*102
 ia = 2*102
 ra = 2*200.100006
2*102,  2*200.100006
```

However, if the *FILENV* environment variable is set to a file and uses the *assign -U* command to change the output behavior, as shown below:

```
% setenv FILENV ASGTMP
% assign -U on g:sf
```

The same code now produces the following output:

```
&TNAMEL  RA =    200.1000        200.1000     IA =         102
102 /
 ia =          102         102
 ra =     200.1000        200.1000
          102         102    200.1000        200.1000
```

For more information about the `assign` command and Assign Environment, see *Enhanced I/O: Using the assign Environment*.

Random Number Generator

A multiplicative congruential generator with period 2**46 is used to produce the output of the RANDOM_NUMBER intrinsic subroutine. The seed array contains one 64-bit integer value.

Timing Intrinsics

A call to the SYSTEM_CLOCK intrinsic subroutine with the COUNT argument present translates into the inline instructions that directly access the hardware clock register. See the description of the `-e s` and `-d s` command line options for information about the values returned for the count and count rate. For fine-grained timing, Cray recommends using a 64-bit COUNT argument.

The CPU_TIME subroutine obtains the value of its argument from the `getrusage` system call. Its execution time is significantly longer than for the SYSTEM_CLOCK routine, but the values returned are closer to those used by system accounting utilities.

IEEE Intrinsic Modules

The IEEE intrinsics modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES are supplied. Denormal numbers are not supported on Cray hardware. The IEEE_SUPPORT_DENORMAL inquiry function returns .false. for all kinds of arguments.

At the start of program execution, all floating point exception traps are disabled.

14 Enhanced I/O: Using the assign Environment

Fortran programs often need the ability to alter details of a file connection, such as device residency, an alternative file name, a file space allocation scheme or structure, or data conversion properties. These file connection details taken together comprise the assign environment, and they can be modified by using the `assign` command and `assign` library interface.

The assign environment can also be accessed from C/C++ by using the `ffassign` library interface. For more information, see the `assign(1)`, `assign(3f)`, and `ffassign(3c)` man pages.

14.1 Understand the `assign` Environment

The assign command information is stored in the assign environment file, .assign, or in a shell environment variable. To begin using the assign environment to control a program's I/O behavior, follow these steps.

Set the *`FILENV`* environment variable to the desired path.

```
set FILENV environment-file
```

Run the assign command to define the current assign environment.

```
assign arguments assign-object
```

For example:

```
assign -F cachea g:su
```

Run the program.

If not satisfied with the I/O performance observed during program execution, return to step 2, use the assign command to adjust the assign environment, and try again.

The `assign` command passes information to Fortran `open` statements and to the `ffopen` routine to identify the following elements:

- A list of numbers
- File names
- File name patterns that have attributes associated with them
- The assign object is the file name, file name pattern, unit number, or type of I/O open request to which the assign environment applies. When the unit or file is opened from Fortran, the environment defined by the assign command is used to establish the properties of the connection.

Assign Objects and Open Processing

The I/O library routines apply options to a file connection for all related assign objects.

If the assign object is a unit, the application of options to the unit occurs whenever that unit is connected.

If the assign object is a file name or pattern, the application of options to the file connection occurs whenever a matching file name is opened from a Fortran program.

When any of the library I/O routines opens a file, it uses the specified assign environment options for any assign objects that apply to the open request. Any of the following assign objects or categories can apply to a given open request.

Table 22. Assign Object Processing

Assign-object	Applies To
g:all	All open requests
g:su	Open sequential unformatted
g:du	Open direct unformatted
g:sf	Open sequential formatted
g:df	Open direct formatted
g_ff	ffopen
u:*unit-number*	Open *unit-number*
p:*pattern*	When a file whose name matches *pattern* is opened. The assign environment can contain only one p:*assign-object* that matches the current open file. The exception is that the p:%*pattern* (which uses the % wildcard character) is silently ignored if a more specific *pattern* also matches the current file name being opened.
f:*filename*	Whenever file *filename* is opened.

Options from the assign objects in these categories are collected to create the complete set of options used for any particular open. The options are collected in the listed order, with options collected later in the list of assign objects overriding those collected earlier.

`assign` Command Syntax

Here is the syntax for the *assign* command:assign -I -O -a *actualfile* -b *bs* -f *fortstd* -m *setting* -s *ft* -t -u *bufcnt* -y *setting* -B *setting* -C *charcon* -D *fildes* -F *spec*,*specs* -N *numcon* -R -S *setting* -T *setting* -U *setting* -V -W *setting* -Y *setting* -Z *setting* *assign-object*

The following specifications cannot be used with any other options:assign -R *assign-object*assign -V *assign-object*

A summary of the command options follows. For details, see the `assign(1)` and `intro_ffio(3f)` man pages.

Control options:

-I Specifies an incremental use of assign. All attributes are added to the attributes already assigned to the current *assign-object*. This option and the -O option are mutually exclusive.

-O Specifies a replacement use of assign. This is the default control option. All currently existing assign attributes for the current *assign-object* are replaced. This option and the -I option are mutually exclusive.

-R Removes all assign attributes for *assign-object*. If *assign-object* is not specified, all currently assigned attributes for all *assign-object*s are removed.

-V Views attributes for *assign-object*. If *assign-object* is not specified, all currently assigned attributes for all *assign-object*s are printed.

Attribute options:

-a *actualfile* The file= specifier or the actual file name.

-b *bs* Library buffer size in 4096-byte (512-word) blocks.

-f *fortstd* Specifies compatibility with a Fortran standard, where *fortstd* is either 2003 for the current Cray Fortran or 95 for Cray Fortran 95. If the value 95 is set, the list-directed and namelist output of a floating point will remain 0.E+0.

-m *setting* Special handling of a direct access file that will be accessed concurrently by several processes or tasks. Special handling includes skipping the check that only one Fortran unit be connected to a unit, suppressing file truncation to true size by the I/O buffering routines, and ensuring that the file is not truncated by the I/O buffering routines. Enter either on or off for *setting*.

-s *ft* File type. Enter text, cos, blocked, unblocked, u, sbin, or bin for *ft*. The default is *text*.

-t Temporary file.

-u *bufcnt* Buffer count. Specifies the number of buffers to be allocated for a file.

-y *setting* Suppresses repeat counts in list-directed output. *setting* can be either on or off. The default setting is off.

-B *setting* Activates or suppresses the passing of the O_DIRECT flag to the open(2) system call. Enter either on or off for *setting*. This is an important feature for I/O optimization; if this is on, it enables reads and writes directly to and from the user program buffer.

-C *charcon* Character set conversion information. Enter ascii, or ebcdic for *charcon*. If the -C option is specified, the -F option must also be specified.

-D *fildes* Specifies a connection to a standard file. Enter stdin, stdout, or stderr for *fildes*.

-F *spec* ,*specs* Flexible file I/O (FFIO) specification. See the `assign(1)` man page for details about allowed values for *spec* and for details about hardware platform support. See the `intro_ffio(3f)` man page for details about specifying the FFIO layers.

-N *numcon* Foreign numeric conversion specification. See the `assign(1)` man page for details about allowed values for *numcon* and for details about hardware platform support.

-S *setting* Suppresses use of a comma as a separator in list-directed output. Enter either on or off for *setting*. The default setting is off.

-T *setting* Activates or suppresses truncation after write for sequential Fortran files. Enter either on or off for *setting*.

-U *setting*	Produces a non-UNICOS form of list-directed output. This is a global setting that sets the value for the -y, -eS, and -W options. Enter either on or off for *setting*. The default setting is off.
-W *setting*	Suppresses compressed width in list-directed output. Enter either on or off for *setting*. The default setting is off.
-Y *setting*	Skips unmatched namelist groups in a namelist input record. Enter either on or off for *setting*. The default setting is on.
-Z *setting*	Recognizes -0.0 for IEEE floating-point systems and writes the minus sign for edit-directed, list-directed, and namelist output. Enter either on or off for *setting*. The default setting is on.
assign-object	Specify either a file name or a unit number for *assign-object*. The `assign` command associates the attributes with the file or unit specified. These attributes are used during the processing of Fortran open statements or during implicit file opens.

Use one of the following formats for *assign-object*:

- f:*filename*
- g:*io-type*, where *io-type* can be su, sf, du, df, or ff (for example, g:ff for `ffopen`(3C)
- p:*pattern* (for example, p:file%)
- u:*unit-number* (for example, u:9)
- *filename*

  When the p:*pattern* form is used, the % and _ wildcard characters can be used. The % matches any string of 0 or more characters. The _ matches any single character. The % performs like the * when doing file name matching in shells. However, the % character also matches strings of characters containing the / character.

Use the Library Routines

The `assign`, `asnunit`, `asnfile`, and `asnrm` routines can be called from a Fortran program to access and update the assign environment. The `assign` routine provides an easy interface to assign processing from a Fortran program. The `asnunit` and `asnfile` routines assign attributes to units and files, respectively. The `asnrm` routine removes all entries currently in the assign environment.

The calling sequences for library routines are as follows:

```
call assign (cmd, ier)

call asnunit (iunit,astring,ier)

call asnfile (fname,astring,ier)

call asnrm (ier)
```

Where:

cmd

Fortran character variable containing a complete `assign` command in the format acceptable to the pxfsystem routine.

ier

Integer variable that is assigned the exit status on return from the library interface routine.

iunit

Integer variable or constant that contains the unit number to which attributes are assigned.

astring

Fortran character variable that contains any attribute options and option values from the `assign` command. Control options -I, -O, and -R can also be passed.

fname

Character variable or constant that contains the file name to which attributes are assigned.

A status of 0 indicates normal return. A status of greater than 0 indicates a specific error status. Use the `explain` command to determine the meaning of the error status.

The following calls are equivalent to the `assign -s u f:file` command:

```
call assign('assign -s u f:file',ier)
call asnfile('file','-s u',ier)
```

The following call is equivalent to executing the **`assign -I -n 2 u:99`** command:

```
iun = 99
call asnunit(iun,'-i -n 2',ier)
```

The following call is equivalent to executing the `assign -R` command:

```
call asnrm(ier)
```

14.2 Tune File Connection Behavior

Use Alternative File Names

The -a option specifies the actual file name to which a connection is made. This option allows files to be created in different directories without changing the FILE= specifier on an OPEN statement.

For example, consider the following `assign` command issued to open unit 1:

```
assign -a /tmp/mydir/tmpfile u:1
```

The program then opens unit 1 with any of the following statements:

```
WRITE(1) variable          ! implicit open
OPEN(1)                    ! unnamed open
OPEN(1,FORM='FORMATTED')   ! unnamed open
```

Unit 1 is connected to file `/tmp/mydir/tmpfile`. Without the -a attribute, unit 1 would be connected to file `fort.1`.

When the -a attribute is associated with a file, any Fortran open that is set to connect to the file causes a connection to the actual file name. An `assign` command of the following form causes a connection to file `$FILENV/joe`:

```
assign -a $FILENV/joe ftfile
```

This is true when the following statement is executed in a program:

```
OPEN(IUN,FILE='ftfile')
```

If the following `assign` command is issued and in effect, any Fortran INQUIRE statement whose FILE= specification is foo refers to the file named actual instead of the file named foo for purposes of the EXISTS=, OPENED=, or UNIT= specifiers:

```
assign -a actual f:foo
```

If the following `assign` command is issued and in effect, the -a attribute does not affect INQUIRE statements with a UNIT= specifier:

```
assign -a actual ftfile
```

When the following OPEN statement is executed, `INQUIRE(UNIT=n,NAME=fname)` returns a value of ftfile in *`fname`*, as if no `assign` had occurred:

```
OPEN(n,file='ftfile')
```

The I/O library routines use only the actual file (-a) attributes from the assign environment when processing an INQUIRE statement. During an INQUIRE statement that contains a FILE= specifier, the I/O library searches the assign environment for a reference to the file name that the FILE= specifier supplies. If an *`assign-by-filename`* exists for the file name, the I/O library determines whether an actual name from the -a option is associated with the file name. If the *`assign-by-filename`* supplied an actual name, the I/O library uses that name to return values for the EXIST=, OPENED=, and UNIT= specifiers; otherwise, it uses the file name. The name returned for the NAME= specifier is the file name supplied in the FILE= specifier. The actual file name is not returned.

Specify File Structure

A file structure defines the way records are delimited and how the end-of-file is represented. The assign command supports two mutually exclusive file structure options:

- To select a structure using an FFIO layer, use assign -F
- To select a structure explicitly, use assign -s

  Using FFIO layers is more flexible than selecting structures explicitly. FFIO allows nested file structures, buffer size specifications, and support for file structures not available through the -s option. Better I/O performance is realized by using the -F option and FFIO layers.

  The remainder of this section covers the -s option.

  Fortran sequential unformatted I/O uses four different file structures: f77 blocked structure, text structure, unblocked structure, and COS blocked structure. By default, the f77 blocked structure is used unless a file structure is selected at open time. If an alternative file structure is needed, the user can select a file structure by using the -s or -F option on the `assign` command.

The -s and -F options are mutually exclusive. The following examples show how to use different `assign` command options to select different file structures.

Structure

`assign` Command

F77 blocked

```
assign -F f77
```

text

```
assign -F text
assign -s text
```

unblocked

```
assign -F system
assign -s unblocked
```

COS blocked

```
assign -F cos
assign -s cos
```

The following examples show how to adjust blocking:

- To select an unblocked file structure for a sequential unformatted file:

```
IUN = 1
CALL ASNUNIT(IUN,'-s unblocked',IER)
OPEN(IUN,FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
```

- The `assign -s u` command can also be used to specify the unblocked file structure for a sequential unformatted file. When this option is selected, I/O is unbuffered. Each Fortran READ or WRITE statement results in a `read` or `write` system call such as the following:

```
CALL ASNFILE('fort.1','-s u',IER)
OPEN(1,FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
```

- To assign unit 10 a COS blocked structure:

```
assign -s cos u:10
```

The full set of options allowed with the `assing -s` command are as follows:

- bin (not recommended)
- blocked
- cos
- sbin
- text
- unblocked

Table 23. Fortran Access Methods and Options

Access and Form	assign -s *ft* Defaults	assign -s *ft* Options
Sequential unformatted, BUFFER IN and BUFFER OUT	blocked / cos / f77	**bin** **sbin** **u** **unblocked**
Direct unformatted	unblocked	**bin** **sbin** **u** **unblocked**
Sequential formatted	`text`	**blocked** **cos** **sbin/text**
Direct formatted	`text`	**sbin/text**

Unblocked File Structure

A file with an unblocked file structure contains undelimited records. Because it does not contain any record control words, it does not have record boundaries. The unblocked file structure can be specified for a file opened with either unformatted sequential access or unformatted direct access. It is the default file structure for a file opened as an unformatted direct-access file.

Do not attempt to use a BACKSPACE statement to reposition a file with an unblocked file structure. Since record boundaries do not exist, the file cannot be repositioned to a previous record.

BUFFER IN and BUFFER OUT statements can specify a file having an unbuffered and unblocked file structure. If the file is specified with `assign -s u`, BUFFER IN and BUFFER OUT statements can perform asynchronous unformatted I/O.

There are several ways to use the `assign` command to specify unblocked file structure. All ways result in a similar file structure but with different library buffering styles, use of truncation on a file, alignment of data, and recognition of an end-of-file record in the file. The following unblocked data file structure specifications are available:

Specification	Structure
`assign -s unblocked`	Library-buffered
`assign -F system`	No library buffering
`assign -s sbin`	Buffering that is compatible with standard I/O; for example, both library and system buffering

The type of file processing for an unblocked data file structure depends on the `assign -s ft` option that is declared or assumed for a Fortran file.

For more information about buffering, see *Specify Buffer BehaviorSpecify Foreign File FormatsDefault Buffer SizesLibrary BufferingSystem CacheUnbuffered I/O*.

An I/O request for a file specified using the `assign -s unblocked` command does not need to be a multiple of a specific number of bytes. Such a file is truncated after the last record is written to the file. Padding occurs for files specified with the `assign -s bin` command and the `assign -s unblocked` command. Padding usually occurs when noncharacter variables follow character variables in an unformatted direct-access file.

No padding is done in an unformatted sequential access file. An unformatted direct-access file created by a Fortran program on CLE systems contains records that are the same length. The end-of-file record is recognized in sequential-access files.

`assign -s sbin` File Processing

Use an `assign -s sbin` specification for a Fortran file opened with either unformatted direct access or unformatted sequential access. The file does not contain record delimiters. The file created for `assign -s sbin` in this instance has an unblocked data file structure and uses unblocked file processing.

The `assign -s sbin` option can be specified for a Fortran file that is declared as formatted sequential access. Because the file contains records that are delimited with the new-line character, it is not an unblocked data file structure. It is the same as a text file structure.

The `assign -s sbin` option is compatible with the standard C I/O functions.

Cray discourages the use of `assign - s sbin` because it typically yields poor I/O performance. If an FFIO layer cannot be used, using `assign -s text` for formatted files and `assign -s unblocked` for unformatted files usually produces better I/O performance than using `assign -s sbin`.

`assign -s bin` File Processing

An I/O request for a file that is specified with `assign -s bin` does not need to be a multiple of a specific number of bytes. Padding occurs when noncharacter variables follow character variables in an unformatted record.

The I/O library uses an internal buffer for the records. If opened for sequential access, a file is not truncated after each record is written to the file.

`assign -s u` File Processing

The `assign -s u` command specifies undefined or unknown file processing. An `assign -s u` specification can be specified for a Fortran file declared as unformatted sequential or direct access. Because the file does not contain record delimiters, it has an unblocked data file structure. Both synchronous and asynchronous BUFFER IN and BUFFER OUT processing can be used with u file processing.

Fortran sequential files declared by using `assign -s u` are not truncated after the last word written. The user must execute an explicit ENDFILE statement on the file.

text File Structure

The text file structure consists of a stream of 8-bit ASCII characters. Every record in a text file is terminated by a newline character (\n, ASCII 012). Some utilities may omit the newline character on the last record, but the Fortran library treats such an occurrence as a malformed record. This file structure may be specified for a file that is declared as either formatted sequential access or formatted direct access. It is the default file structure for formatted sequential access and formatted direct access files.

The assign -s text command specifies the library-buffered text file structure. Both library and system buffering are done for all text file structures.

An I/O request for a file using `assign -s text` does not need to be a multiple of a specific number of bytes.

BUFFER IN and BUFFER OUT statements cannot be used with this structure. Use a BACKSPACE statement to reposition a file with this structure.

cos or blocked File Structure

The cos or blocked file structure uses control words to mark the beginning of each sector and to delimit each record. Specify this file structure for a file that is declared as unformatted sequential access. Synchronous BUFFER IN and BUFFER OUT statements can create and access files with this file structure.

Specify this file structure with one of the following `assign` commands:

```
assign -s cos
assign -s blocked
assign -F cos
assign -F blocked
```

These four `assign` commands result in the same file structure.

An I/O request on a blocked file is library buffered.

In a cos file structure, one or more ENDFILE records are allowed. BACKSPACE statements can be used to reposition a file with this structure.

A blocked file is a stream of words that contains control words called Block Control Word (BCW) and Record Control Words (RCW) to delimit records. Each record is terminated by an EOR (end-of-record) RCW. At the beginning of the stream, and every 512 words thereafter (including any RCWs), a BCW is inserted. An end-of-file (EOF) control word marks a special record that is always empty. Fortran considers this empty record to be an endfile record. The end-of-data (EOD) control word is always the last control word in any blocked file. The EOD is always immediately preceded by either an EOR, or by an EOF and a BCW.

Each control word contains a count of the number of data words to be found between it and the next control word. In the case of the EOD, this count is 0. Because there is a BCW every 512 words, these counts never point forward more than 511 words.

A record always begins at a word boundary. If a record ends in the middle of a word, the rest of that word is zero filled; the ubc field of the closing RCW contains the number of unused bits in the last word.

The following illustration and table is a representation of the structure of a BCW.

m	unused	bdf	unused	bn	fwi
(4)	(7)	(1)	(19)	(24)	(9)

Field	Bits	Description
m	0-3	Type of control word; 0 for BCW
bdf	11	Bad Data flag (1-bit, 1=bad data)
bn	31-54	Block number (modulo 224)

Field	Bits	Description
fwi	55-63	Forward index; the number of words to the next control word

The following illustration and table is a representation of the structure of an RCW.

m	ubc	tran	bdf	srs	unused	pfi	pri	fwi
(4)	(6)	(1)	(1)	(1)	(7)	(20)	(15)	(9)

Field	Bits	Description
m	0-3	Type of control word; 108 for EOR, 168 for EOF, and 178 for EOD
ubc	4-9	Unused bit count; number of unused low-order bits in last word of previous record
tran	10	Transparent record field (unused)
bdf	11	Bad data flag (unused)
srs	12	Skip remainder of sector (unused)
pfi	20-39	Previous file index; offset modulo 220 to the block where the current file starts (as defined by the last EOF)
pri	40-54	Previous record index; offset modulo 215 to the block where the current record starts
fwi	55-63	Forward index; the number of words to the next control word

Specify Buffer Behavior

A buffer is a temporary storage location for data while the data is being transferred. Buffers are often used for the following purposes:

- Small I/O requests can be collected into a buffer, and the overhead of making many relatively expensive system calls can be greatly reduced.
- Many data file structures such as cos contain control words. During the write process, a buffer can be used as a work area where control words can be inserted into the data stream (a process called blocking). The blocked data is then written to the device. During the read process, the same buffer work area can be used to remove the control words before passing the data on to the user (called deblocking).
- When data access is random, the same data may be requested many times. A *cache* is a buffer that keeps old requests in the buffer in case these requests are needed again. A cache that is sufficiently large or efficient can avoid a large part of the physical I/O by having the data ready in a buffer. When the data is often

found in the cache buffer, it is referred to as having a high hit rate. For example, if the entire file fits in the cache and the file is present in the cache, no more physical requests are required to perform the I/O. In this case, the hit rate is 100%.

- Running the I/O devices and the processors in parallel often improves performance; therefore, it is useful to keep processors busy while data is being moved. To do this when writing, data can be transferred to the buffer at memory-to-memory copy speed. Use an asynchronous I/O request. The control is then immediately returned to the program, which continues to execute as if the I/O were complete (a process called write-behind). A similar process called read-ahead can be used while reading; in this process, data is read into a buffer before the actual request is issued for it. When it is needed, it is already in the buffer and can be transferred to the user at very high speed.
- When direct I/O is enabled (assign -B on), data is staged in the system buffer cache. While this can yield improved performance, it also means that performance is affected by program competition for system buffer cache. To minimize this effect, avoid public caches when possible.
- In many cases, the best asynchronous I/O performance can be realized by using the FFIO cachea layer (assign -F cachea). This layer supports read-ahead, write-behind, and improved cache reuse.

  The size of the buffer used for a Fortran file can have a substantial effect on I/O performance. A larger buffer size usually decreases the system time needed to process sequential files. However, large buffers increase a program's memory usage; therefore, optimizing the buffer size for each file accessed in a program on a case-by-case basis can help increase I/O performance and minimize memory usage.

  The `-b` option on the `assign` command specifies a buffer size, in blocks, for the unit. The `-b` option can be used with the `-s` option, but it cannot be used with the `-F` option. Use the `-F` option to provide I/O path specifications that include buffer sizes; the `-b`, and `-u` options do not apply when `-F` is specified.

  For more information about the selection of buffer sizes, see the `assign(1)` man page.

The following examples of buffer size specification illustrate using the `assign -b` and `assign -F` options:

- If unit 1 is a large sequential file for which many Fortran `READ` or `WRITE` statements are issued, increase the buffer size to a large value, using the following `assign` command:

  ```
  assign -b buffer-size u:buffer-count
  ```

- If the file `foo` is a small file or is accessed infrequently, minimize the buffer size using the following `assign` command:

  ```
  assign -b 1 f:foo
  ```

Specify Foreign File Formats

The Fortran I/O library can read and write files with record blocking and data formats native to operating systems from other vendors. The `assign -F` command specifies a foreign record blocking; the `assign -C` command specifies the type of character conversion; the -N option specifies the type of numeric data conversion. When -N or -C is specified, the data is converted automatically during the processing of Fortran `READ` and `WRITE` statements. For example, assume that a record in file `fgnfile` contains the following character and integer data:

```
character*4 ch
integer int
open(iun,FILE='fgnfile',FORM='UNFORMATTED')
read(iun) ch, int
```

Use the following `assign` command to specify foreign record blocking and foreign data formats for character and integer data:

```
assign -F ibm.vbs -N ibm -C ebcdic fgnfile
```

One of the most common uses of the `assign` command is to swap big-endian for little-endian files. To access big-endian unformatted files on a little-endian system, use the following command:

```
assign -N swap_endian fgnfile
```

This assumes the file is a normal `f77` unformatted file with 32-bit record control images with a byte count. The library routines swap both the control images and the data when reading or writing the file.

If all unformatted sequential files are the opposite endianness, use the following command:

```
assign -N swap_endian g:su
```

Default Buffer Sizes

The Fortran I/O library automatically selects default buffer sizes according to file access type as shown in the table, *Default Buffer Sizes for Fortran I/O Library Routines*. Override the defaults by using the `assign` command. The following subsections describe the default buffer sizes on various systems.

One block is 4,096 bytes on CLE systems.

Table 24. Default Buffer Sizes for Fortran I/O Library Routines

Access Type	Default Buffer Size
Sequential formatted	16 blocks (65,536 bytes)
Sequential unformatted	128 blocks (524,288 bytes)
Direct formatted	The smaller of: • The record length in bytes + 1 • 16 blocks (65,536 bytes)
Direct unformatted	The larger of: • The record length • 16 blocks (65,536 bytes)

Four buffers of default size are allocated. For more information, see the description of the cachea layer in the `intro_ffio(3F)` man page.

Library Buffering

The term library buffering refers to a buffer that the I/O library associates with a file. When a file is opened, the I/O library checks the access, form, and any attributes declared on the `assign` command to determine the type of processing that should be used on the file. Buffers are an integral part of the processing.

If the file is assigned with one of the following `assign` options, library buffering is used:

- -s blocked
- -F *spec* (buffering as defined by *spec*)
- -s cos
- -s bin
- -s unblocked

The `-F` option specifies flexible file I/O (FFIO), which uses library buffering if the specifications selected include a need for buffering. In some cases, more than one set of buffers might be used in processing a file. For example, the `-F bufa,cos` option specifies two library buffers for a read of a blank compressed COS blocked file. One buffer handles the blocking and deblocking associated with the COS blocked control words, and the second buffer is used as a work area to process blank compression. In other cases (for example, `-F system`), no library buffering occurs.

System Cache

The operating system uses a set of buffers in kernel memory for I/O operations. These are collectively called the system cache. The I/O library uses system calls to move data between the user memory space and the system buffer. The system cache ensures that the actual I/O to the logical device is well formed, and it tries to remember recent data in order to reduce physical I/O requests.

The following `assign` command options can be expected to use system cache:

- -s sbin
- -F *spec* (FFIO, depends on *spec*)

For the `assign -F cachea` command, a library buffer ensures that the actual system calls are well formed and the system buffer cache is bypassed. This is not true for the `assign -s u` option. If `assign -s u` is planned to be used to bypass the system cache, all requests must be well formed.

Unbuffered I/O

The simplest form of buffering is none at all; this unbuffered I/O is known as direct I/O. For sufficiently large, well-formed requests, buffering is not necessary and can add unnecessary overhead and delay. The following `assign` command specifies unbuffered I/O:

```
assign -s u  ...
```

Use the `assign` command to bypass both library buffering and the system cache for all well-formed requests. The data is transferred directly between the user data area and the logical device. Requests that are not well formed will result in I/O errors.

Specify Foreign File Formats

The Fortran I/O library can read and write files with record blocking and data formats native to operating systems from other vendors. The `assign -F` command specifies a foreign record blocking; the `assign -C` command specifies the type of character conversion; the `-N` option specifies the type of numeric data conversion. When `-N` or `-C` is specified, the data is converted automatically during the processing of Fortran `READ` and `WRITE` statements. For example, assume that a record in file `fgnfile` contains the following character and integer data:

```
character*4 ch
integer int
```

```
open(iun,FILE='fgnfile',FORM='UNFORMATTED')
read(iun) ch, int
```

Use the following `assign` command to specify foreign record blocking and foreign data formats for character and integer data:

```
assign -F ibm.vbs -N ibm -C ebcdic fgnfile
```

One of the most common uses of the `assign` command is to swap big-endian for little-endian files. To access big-endian unformatted files on a little-endian system, use the following command:

```
assign -N swap_endian fgnfile
```

This assumes the file is a normal `f77` unformatted file with 32-bit record control images with a byte count. The library routines swap both the control images and the data when reading or writing the file.

If all unformatted sequential files are the opposite endianness, use the following command:

```
assign -N swap_endian g:su
```

Specify Memory Resident Files

The `assign -F mr` command specifies that a file will be memory resident. Because the mr flexible file I/O layer does not define a record-based file structure, it must be nested beneath a file structure layer when record blocking is needed.

For example, if unit 2 is a sequential unformatted file that is to be memory resident, the following Fortran statements connect the unit:

```
CALL ASNUNIT (2,'-F cos,mr',IER)
OPEN(2,FORM='UNFORMATTED')
```

The `-F cos,mr` specification selects COS blocked structure with memory residency.

Use and Suppress File Truncation

The `assign -T` option activates or suppresses truncation after the writing of a sequential Fortran file. The `-T on` option specifies truncation; this behavior is consistent with the Fortran standard and is the default setting for most `assign -s fs` specifications.

The `assign(1)` man page lists the default setting of the `-T` option for each `-s fs` specification. It also indicates if suppression or truncation is allowed for each of these specifications.

FFIO layers that are specified by using the `-F` option vary in their support for suppression of truncation with `-T off`.

The following figure, *Access Methods and Default Buffer Sizes*, summarizes the available access methods and the default buffer sizes.

Figure 2. Access Methods and Default Buffer Sizes

	Blocked			Unblocked			
Access method `assign` option	Blocked `-F f77`	Blocked `-s cos`	Text `-s text`	Undef `-s u`	Binary `-s bin`	Unblocked `-s unblocked`	Buffer size for default*

Formatted sequential I/O
`WRITE(9,20)`
`PRINT`
		Valid	Valid Default				16

Formatted direct I/O `WRITE(9,20,REC=)`			Valid Default	Valid		Valid	min(recl+1, 8) bytes
Unformatted sequential I/O `WRITE(9)`	Valid Default	Valid		Valid	Valid	Valid	128
Unformatted direct I/O `WRITE(9,REC=)`				Valid	Valid	Valid Default	max(16, recl) blocks
Buffer in/buffer out	Valid Default	Valid		Valid	Valid	Valid	16
Control words	Yes	Yes	`NEWLINE`	No	No	No	
Library buffering	Yes	Yes	Yes	No	Yes	Yes	
System cached	Yes	No	Yes	No†	No††	Varies	
`BACKSPACE`	Yes	Yes	Yes	No	No	No	
Record size	Any	Any	Any	Any	8*n	Any	
Default library buffer size*	16	48	16	None	16	16	

† Cached if not well-formed
†† No guarantee when physical size not 512 words
* In units of 4096 bytes, unless otherwise specified

14.3 Define the Assign Environment File

The `assign` command information is stored in the assign environment file. The location of the active assign environment file must be provided by setting the *`FILENV`* environment variable to the desired path and file name.

14.4 Use Local Assign Mode

The assign environment information is usually stored in the `.assign` environment file. Programs that do not require the use of the global `.assign` environment file can activate local assign mode. If local assign mode is selected, the assign environment will be stored in memory. Thus, other processes can not adversely affect the assign environment used by the program.

The `ASNCTL` routine selects local assign mode when it is called by using one of the following command lines:

```
CALL ASNCTL('LOCAL',1,IER)
CALL ASNCTL('NEWLOCAL',1,IER)
```

Local assign mode

In the following example, a Fortran program activates local assign mode and then specifies an unblocked data file structure for a unit before opening it. The `-I` option is passed to `ASNUNIT` to ensure that any assign attributes continue to have an effect at the time of file connection.

```
C     Switch to local assign environment
      CALL ASNCTL('LOCAL',1,IER)
      IUN = 11
C     Assign the unblocked file structure
      CALL ASNUNIT(IUN,'-I -s unblocked',IER)
C     Open unit 11
      OPEN(IUN,FORM='UNFORMATTED')
```

If a program contains all necessary assign statements as calls to `ASSIGN`, `ASNUNIT`, and `ASNFILE`, or if a program requires total shielding from any `assign` commands, use the second form of a call to `ASNCTL`, as follows:

```
C     New (empty) local assign environment
      CALL ASNCTL('NEWLOCAL',1,IER)
      IUN = 11
C     Assign a large buffer size
      CALL ASNUNIT(IUN,'-b 336',IER)
C     Open unit 11
      OPEN(IUN,FORM='UNFORMATTED')
```

15 Interlanguage Communication

The C and C++ compilers provide mechanisms for declaring external functions written in other languages. This enables the writing of portions of an application in C, C++, Fortran, or assembly language, which can be useful in cases where the other languages provide performance advantages or utilities not available in C or C++.

Fortran, C, C++ Interoperability

The Cray Compiler supports interoperability mechanisms specfied in the Fortran 2008 standard, ISO/IEC 1539-1:2010, and *TS 29113 Further Interoperability of Fortran and C*.

The Fortran 2008 standard describes interoperabilty features for:

Intrinsic Types

The Fortran intrinsic module `ISO_C_BINDING` provides interoperability between Fortran intrinsic types and C types. The `ISO_C_BINDING` module provides named constants which can be used as `KIND` type parameters, compatible with C types.

In addition to the named constants required by the Fortran standard, Cray compiler provides, as an extension, definitions for 128-bit floating, and complex types. `C_FLOAT128` and `C_FLOAT128_COMPLEX` correspond to C types `__float128` and `__float128 complex`.

Derived Types and Structures

Use the `BIND` attribute when creating an interoperable type:

```
USE ISO_C_BINDING
TYPE, BIND(C) :: THIS_TYPE
    . . .
END TYPE THIS_TYPE
```

Global Variables

Use the `BIND` attribute with a common block declaration, or module variable:

```
USE ISO_C_BINDING
     INTEGER(C_INT), BIND(C) :: EXTERN
     INTEGER(C_LONG) :: CVAR
     BIND(C, NAME='var') :: CVAR
     COMMON /A/ I, J
     REAL(C_FLOAT) :: I, J
     BIND(C) :: /A/
```

Pointers

`ISO_C_BINDING` provides a derived type, `c_ptr`, that interoperates with any C pointer type. Also, Fortran named constant `c_null_ptr` is equivalent to the C value NULL.

Subroutines and Function

Declare a Fortran procedure with the BIND attribute. Procedure arguments must be of interoperable type. By default the Fortran compiler converts the procedure name to lower-

case (`myfunction`); this is the binding label, or corresponding name which is known to the C compiler.

```
FUNCTION MYFUNCTION(X, Y), BIND(C)
```

Specify a different binding label:

```
FUNCTION MYFUNCTION(X, Y), BIND(C, NAME='C_Myfunction')
```

A function result must be scalar and of interoperable type. A subroutine prototype must have a void result.

TS 29113 describes further interoperability features including:

C descriptors	`ISO_Fortran_binding.h` defines C structure `CFI_cdesc_t` which facilitates using Fortran data objects from within a C function.
`ISO_Fortran binding.h`	Contains additional C structure definitions and macro definitions to interoperate with an allocatable, or data pointer argument.

BIND(C) Syntax

The *`proc-language-binding-spec`* specification allows Fortran programs to interoperate with C objects. The optional commas in FUNCTION name(), BIND(C) are Cray extensions to the Fortran standard.

ISO_C_BINDING

The ISO_C_BINDING module provides interoperability between Fortran intrinsic types and C types. The ISO_C_BINDING module provides named constants which can be used as KIND type parameters, compatible with C types.

In addition to the named constants required by the Fortran 2008 standard, Cray compiler provides, as an extension, definitions for 128-bit floating, and complex types. C_FLOAT128 and C_FLOAT128_COMPLEX correspond to C types __float128 and __float128 complex.

Interlanguage Communication Examples

Interlanguage Communication using Common Block/Global

```
//  common_c.c : example of function called from common.f90

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

//   globals that match up to the common blocks in common.f90
float c_single;

struct common {
  double var1;
  int var2;
} multiple;

int c_int_array[100];
```

```
// c function called from Fortran
void global_var_common()
{
  int i;
  //  just prints and sets the globals


  printf(" In global_var_common\n");
  printf("   c_single: %f\n", c_single);
  printf("   multiple: %f, %d\n", multiple.var1, multiple.var2 );
  printf("   c_int_array: %d, %d\n", c_int_array[0], c_int_array[99]);

  c_single = 2 * c_single;
  multiple.var1 = 77.77;
  multiple.var2 = 17;
  for(i=0; i<100; i++ ) {
     c_int_array[i] = c_int_array[i] * 3;
  }

}  // end of global_var_common
```

```
! common.f90
! Needs common_c.c

program common_block
  use, intrinsic :: iso_c_binding
!  use check_error
  implicit none
!
!  declare the common blocks for c globals
!  one with a single real variable
  real(c_float) r_var
  common /c_single/ r_var
!  one with an integer array
  integer i_array(100)
  common / array / i_array
!  one with two variables
  real(c_double) :: var1
  integer(c_int) :: var2
  common / multiple / var1, var2
!   do the bind c on the common blocks, renaming one
BIND(C,name="c_int_array") :: / array /
BIND(C) :: / multiple /,  /c_single/

call sub1()

end program common_block

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine sub1( )
  use, intrinsic :: iso_c_binding

!  declare the common blocks for c globals
!  one with a single real variable
  real(c_float) r_var
  common /c_single/ r_var
!  one with an integer array
  integer i_array(100)
```

```
  common / array / i_array
  real(c_double) var1
  integer(c_int) var2
  common / multiple / var1, var2
!   do the bind c on the common blocks, renaming array
BIND(C,name="c_int_array") :: / array /
BIND(C) :: / multiple /,  /c_single/

  interface
     subroutine global_var_common( )  bind(c)
       use,intrinsic :: iso_c_binding
       implicit none
     end subroutine global_var_common
  end interface

r_var = -99.3
var1 = 88.88
var2 = -13
i_array = [(i,i=1,100)]


! call the c function
call global_var_common( )

print *, "In sub1"
print *, "  r_var : ", r_var
print *, "  var1  : ", var1
print *, "  var2  : ", var2
print *, "  array : ", i_array(1), i_array(100)

end subroutine
```

Interlanguage Communication using Derived Structure

```
// c program that calls the Fortran subroutine with struct argument,
f2008 C.11.3
//
**********************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <ISO_Fortran_binding.h>

  //  declare the structure type
struct pass {
int lenc, lenf;
float *c, *f;
};

  //  prototype for the Fortran function
void simulation(long alpha, double *beta, long *gamma, double
delta[], struct pass *arrays);

//  program that calls the Fortran subroutine
int main ( )
{
  int i;
  long alpha, gamma;
  double beta, delta[100];
```

```
  struct pass arrays;

  alpha = 1234L;
  gamma = 5678L;
  beta = 12.34;
  for(i=0; i<100; i++ ) {
     delta[i] = i+1;
  }

  //  fill in some of the structure
  arrays.lenc = 100;
  arrays.lenf = 0;
  arrays.c = (float *) malloc( 100*sizeof(float) );
  arrays.f = NULL;
  for(i=0; i<100; i++ ) {
     arrays.c[i] = 2*(i+1);
  }

  //  reference the Fortran subroutine
  simulation(alpha, &beta, &gamma, delta, &arrays);

  printf(" After simulation\n");
  printf("   alpha: %d, beta: %f\n", alpha, beta );
  printf("   gamma: %d\n", gamma );
  printf("   arrays.lenc: %d\n", arrays.lenc);
  printf("   arrays.c[0],[arrays.lenc-1],: %f, %f\n", arrays.c[0],
arrays.c[arrays.lenc-1]);
  printf("   arrays.lenf: %d\n", arrays.lenf);
  printf("   arrays.f[0],[arrays.lenf-1],: %f, %f\n", arrays.f[0],
arrays.f[arrays.lenf-1]);

}  //  end of main
```

```
! Example derived type/structure interoperability, f2008 C.11.3
!*************************************************************
subroutine simulation(alpha, beta, gamma, delta, arrays) bind(c)
  use, intrinsic :: iso_c_binding
  implicit none
  integer (c_long), value :: alpha
  real (c_double), intent(inout) :: beta
  integer (c_long), intent(out) :: gamma
  real (c_double),dimension(*),intent(in) :: delta
  type, bind(c) :: pass
    integer (c_int) :: lenc, lenf
    type (c_ptr) :: c, f
  end type pass
  type (pass), intent(inout) :: arrays
  real (c_float), allocatable, target, save :: eta(:)
  real (c_float), pointer :: c_array(:)
  integer i

  print *, "In simulation"
  print *, "  alpha: ", alpha, ", beta: ", beta
  print *, "  delta(1),(100): ", delta(1), delta(100)

  ! associate c_array with an array allocated in c
  call c_f_pointer (arrays%c, c_array, [arrays%lenc])
  print *, "  c_array(1),(arrays%lenc): ", c_array(1), c_array(arrays
%lenc)
```

```
  ! allocate an array and make it available in c
  arrays%lenf = 100
  allocate (eta(arrays%lenf))
  arrays%f = c_loc(eta)
  eta = [(i*3,i=1,arrays%lenf)]

  ! change argument values
  c_array = c_array * 2.0
  gamma = 77
  beta = -55.66

end subroutine simulation
```

Interlanguage Communication using Module

```
//  c function called from module.f90

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

//   globals that match up to the module variables in module.f90
float r_var;
double var1;
int var2;
int c_int_array[100];

//  c function called from Fortran
void global_var_module()
{
  int i;
  //  just prints and sets the globals


  printf(" In global_var_module\n");
  printf("   r_var      : %f\n", r_var);
  printf("   var1       : %f\n", var1 );
  printf("   var2       : %d\n", var2 );
  printf("   c_int_array: %d, %d\n", c_int_array[0], c_int_array[99]);

  r_var = 2 * r_var;
  var1 = 77.77;
  var2 = 17;
  for(i=0; i<100; i++ ) {
     c_int_array[i] = c_int_array[i] * 3;
  }

}  // end of global_var_module
```

```


! Example of module/global variable interoperability.
! Needs c function from module_c.c
! ********************************************************************
module module_example_mod
  use, intrinsic :: iso_c_binding
  real(c_float) r_var
  integer i_array(100)
  real(c_double) :: var1
```

```
  integer(c_int) :: var2
BIND(C,name="c_int_array") :: i_array
BIND(C) :: r_var, var1, var2
end module module_example_mod

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program module_example
  use module_example_mod
  implicit none

call sub1()

end program module_example

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine sub1( )
  use module_example_mod

  interface  ! for the c function
     subroutine global_var_module( )  bind(c)
       use,intrinsic :: iso_c_binding
       implicit none
     end subroutine global_var_module
  end interface

r_var = -99.3
var1 = 88.88
var2 = -13
i_array = [(i,i=1,100)]


! call the c function
call global_var_module( )

print *, "In sub1"
print *, "  r_var : ", r_var
print *, "  var1  : ", var1
print *, "  var2  : ", var2
print *, "  array : ", i_array(1), i_array(100)

end subroutine
```