# Cray® Fortran Reference Manual

S–3901–60

# New Features

*Cray® Fortran Reference Manual* S–3901–60

This document is a consolidation of the *Cray Fortran Reference Manual*, *Fortran Language Reference Manual, Volume 1*, *Fortran Language Reference Manual, Volume 2*, *Fortran Language Reference Manual, Volume 3*, and *Fortran Application Programmer's I/O Reference Manual*. It documents the Cray Fortran compiler command options and directives and describes how the Cray Fortran compiler differs from the Fortran 2003 standard

The organization of the contents of Chapter 10, page 179 *Cray Fortran Language Extensions*, parallels the organization of the contents of the official manual of the Fortran 2003 Standard, ISO / IEC 1539-1:2004.

The Cray Fortran compiler includes the following new features:

- Abstract type.
- Support for the Cray X2 series system.
- Finalization for non-polymorphic objects. See Section 12.2, page 257.

The Cray Fortran compiler supports the following proposed Fortran 2008 features:

- Submodules
- Separate module procedures.
- `CONTAINS` followed by an `END` statement with no internal or module procedure. See Section 10.10.2.2, page 204.

The following new directive has been documented in this release:

- `!PGO$ loop_info`, a special form of the `!DIR$ loop_info` directive. See Section 3.19.28, page 60.

# Record of Revision

| *Version* | *Description* |
|---|---|
| 5.6 | March 2007<br>Supports the Cray Fortran compiler 5.6 release running on Cray X1 series systems. |
| 6.0 | September 2007<br>Supports the Cray Fortran compiler 6.0 release running on Cray X1 series and Cray X2 systems. |

# Contents

**Page**

**Page**

*Page*

*Page*

*Page*

*Page*

*Page*

*Page*

*Page*

*Page*

*Page*

*Page*



## Figures



## Tables

*Page*

*Page*

# Preface

The information in this preface is common to Cray documentation provided with this software release.

## Accessing Product Documentation

With each software release, Cray provides books and man pages, and in some cases, third-party documentation. These documents are provided in the following ways:

CrayDoc

The Cray documentation delivery system that allows you to quickly access and search Cray books, man pages, and in some cases, third-party documentation. Access this HTML and PDF documentation via CrayDoc at the following locations:

- The local network location defined by your system administrator
- The CrayDoc public website: `docs.cray.com`

Man pages

Access man pages by entering the `man` command followed by the name of the man page. For more information about man pages, see the `man`(1) man page by entering:

```
% man man
```

Third-party documentation

Access third-party documentation not provided through CrayDoc according to the information provided with the product.

## Conventions

These conventions are used throughout Cray documentation:

| Convention | Meaning |
|---|---|
| `command` | This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements. |
| *variable* | Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined. |
| **`user input`** | This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font. |
| [ ] | Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on. |
| . . . | Ellipses indicate that a preceding element can be repeated. |
| `name`(N) | Denotes man pages that provide system and programming reference information. Each man page is referred to by its name followed by a section number in parentheses.<br><br>Enter:<br><br>`% man man`<br><br>to see the meaning of each section number for your particular system. |

## Reader Comments

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

**E-mail:**
`docs@cray.com`

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

**Telephone (outside U.S., Canada):**
+1–715–726–4993 (Cray Customer Support Center)

**Mail:**
Customer Documentation
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

## Cray User Group

The Cray User Group (CUG) is an independent, volunteer-organized international corporation of member organizations that own or use Cray Inc. computer systems. CUG facilitates information exchange among users of Cray systems through technical papers, platform-specific e-mail lists, workshops, and conferences. CUG memberships are by site and include a significant percentage of Cray computer installations worldwide. For more information, contact your Cray site analyst or visit the CUG website at `www.cug.org`.

# Introduction [1]

This manual describes the differences between the language specified by the Fortran 2003 Standard and the language implemented by the Cray Fortran compiler for the Programming Environment 6.0 Release. The Cray Fortran compiler version 6.0 targets both the Cray X1 series systems and the Cray X2 systems using the UNICOS/mp (3.1 release or later) and UNICOS/lc operating systems.

The Cray Fortran compiler was developed to support the Fortran 2003 standard adopted by the International Organization for Standardization (ISO). This standard, commonly referred to in this manual as the Fortran standard, is ISO/IEC 1539-1:2004.

**Note:** The standards organizations continue to interpret the Fortran standard for Cray and other vendors. To maintain conformance to the Fortran standard, Cray may need to change the behavior of certain Cray Fortran compiler features in future releases based on the outcomes of interpretations to the standard.

Because the Fortran 2003 standard is a superset of previous standards, the Cray Fortran compiler compiles code written in accordance with previous Fortran standards.

**Note:** The `ftn`(1) man page may get updated more often than this document. Where the information differs, the information in the man page supersedes the information contained in this manual.

## 1.1 X1-specific and X2-specific Content in this Document

Unless explicitly indicated by the notations defined below, the contents of this manual apply to both the Cray X1 and the Cray X2 systems.

| Convention | Meaning |
|---|---|
| (X1 only) | This notation indicates that the feature applies only to the Cray X1 series system. Depending on context, the notation occurs either before the text (for example, the second paragraph in section 4.2) or after the text (for example, the chapter title for Chapter 2, The Trigger Environment). |
| (X2 only) | This notation indicates that the feature applies only to the Cray X2 system. Depending on context, the notation occurs either before the text (for example, the fourth paragraph in section 4.2) or after the text (for example, the third bullet item in section 3.19.3). |

## 1.2 The Cray Fortran Programming Environment

The Cray Fortran Programming Environment consists of the tools and libraries that you use to develop Fortran applications. To effectively use these tools and libraries, you must have an understanding of the development environment as discussed in the two documents: *Cray X1 Series System Overview* and *Cray X2 System Overview*.

The Cray Fortran Programming Environment provides the following tools and libraries:

- The `ftn` command, which invokes the Cray Fortran compiler. For more information about `ftn`, see Chapter 3, page 15 or the `ftn`(1) man page.
- The CrayLibs libraries, which provides library routines, intrinsic procedures, I/O routines, and data conversion routines.
- The LibSci libraries, which provide scientific library 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 `ld` command, which invokes the Cray loader. See the `ld`(1) man page for more information.

  **Note:** Cray recommends that you use the `ftn` compiler command to invoke the loader, because the compiler calls the loader with the appropriate default libraries. The appropriate default libraries may change from release to release.
- The CrayPat performance analyzer tool, which can help you analyze program performance. See the `pat`(1) man page for more information.
- The Cray Apprentice2 report visualization tool, which can help you further analyze performance data captured by CrayPat. See the `app2`(1) man page for more information.
- The Etnus TotalView debugger, which can help you debug your program. It includes standard debugging capabilities, such as stepping through code and setting breakpoints. The `-g` and `-G` *debug* options to the `ftn` command line generate symbol tables, which can be used by the debugger. For more debugger information, see the `totalview` and `totalviewcli` man pages.

In the most basic case, the Cray Fortran compiler products are used as follows. The `ftn` command invokes the Cray Fortran compiler, processes the input files named on the command line, and generates a binary file. The compiler then invokes the loader, which loads the binary file(s) and generates an executable output file (the default output file is `a.out`). The `ftnlx` command generates a program listing file, if requested.

In the following simple example, the `ftn` command invokes the Fortran compiler. Option `-r s` is specified to generate a source listing. File `pgm.f` is your source code input file. You run the program by entering the output file name as a command; in this example, the default output file name, `a.out`, is used. Figure 1 illustrates this example.

```
% ftn -r s pgm.f
% ./a.out
```

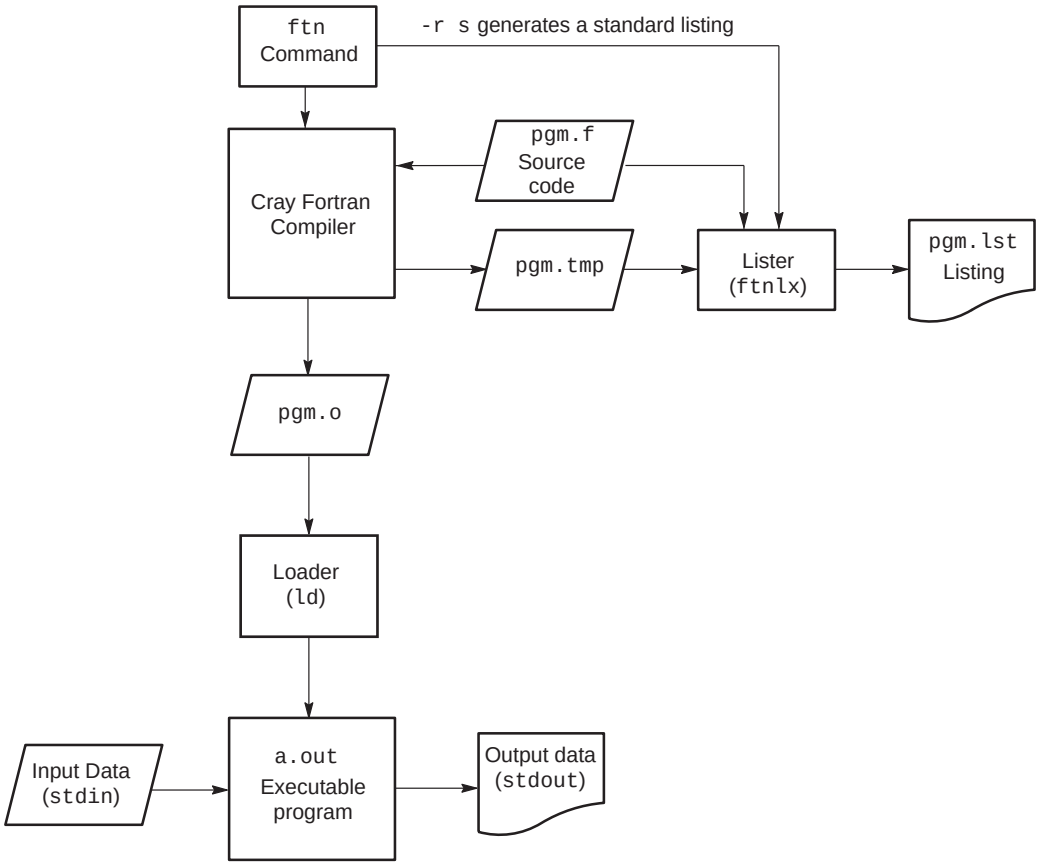


Figure 1. `ftn` Command Example

By default, the Cray Fortran compiler creates files during processing. It attaches various extensions to the base file name and places them into your working directory:

- The compiled code is sent to object file *file*`.o` in the current directory.
- The executable file is `a.out` by default. You can use the `-o` option to specify the name of the executable file.
- If specified, assembly language output is sent to *file*`.s`. Source file names ending with `.s` are assembled, and the assembled code is written to the corresponding *file*`.o`.

You can use the options on the `ftn` command line to modify the default actions; for example, you can change the size of the default data types. For more information about `ftn` command line options, see Chapter 3, page 15.

### 1.2.1 Cross-compiler Platforms

The Cray X1 Series Programming Environment and the Cray X2 Programming Environment also run on cross-compiler platforms. You can use a cross-compiler platform to compile programs and create binaries for subsequent execution on a Cray X1 series system or a Cray X2 system. If your site has the proper licensing in place, you might choose to use one of these other platforms. In the case of the Cray X1 series system, it will afford faster compile time and give you access to the Cray Programming Environment when the X1 system is not available. Supported platforms are listed in the *Cray Programming Environment Releases Overview and Installation Guide.*

## 1.3 Cray Fortran Compiler Messages

You can obtain Cray Fortran compiler message explanations by using the `explain` command. For more information, see the `explain`(1) man page.

## 1.4 Document-specific Conventions

The following conventions are specific to this document:

| Convention | Meaning |
|---|---|
| R*nnn* | The R*nnn* notation indicates that the feature is in the Fortran standard and can be located in the standard via the R*nnn* syntax rule number. |
| Cray pointer | The term *Cray pointer* refers to the Cray pointer data type extension. |

## 1.5 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.

You can direct the compiler to flag and generate messages when nonstandard usage of Fortran is encountered. For more information about this command line option (`ftn -en`), see Section 3.5, page 18 or the `ftn`(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.

### 1.5.1 Fortran 95 Compatibility

No known issues.

### 1.5.2 Fortran 90 Compatibility

No known issues.

### 1.5.3 FORTRAN 77 Compatibility

The format of a floating-point zero written with a `G` edit descriptor is different in Fortran 95. The floating-point zero was written with an `E`*w.d* edit descriptor in FORTRAN 77, but is written with an `F`*w.d* edit descriptor in the Cray Fortran compiler. FORTRAN 77 output cannot be changed. Therefore, different compare files must be retained for FORTRAN 77 and Fortran 95 programs that use the `G` edit descriptor for floating-point output.

## 1.6 Related Cray Publications

The following documentation can aid in the development of your Fortran programs:

- `ftn`(1) man page
- `ftnlx`(1) man page
- *Cray X1 Series System Overview*
- *Cray X2 Series System Overview*
- *Optimizing Applications on Cray X1 Series Systems*
- Loader man page, `ld`(1)

## 1.7 Related Fortran Publications

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

- Fortran 2003 Standard can be downloaded from `http://www.nag.co.uk/sc22wg5/` or `http://j3-fortran.org/`
- Chapman, S. *Fortran 95/2003 for Scientists & Engineers.* McGraw Hill, 2007. ISBN 0073191574.
- Metcalf, M., J. Reid, and M. Cohen. *Fortran 95/2003 Explained.* Oxford University Press, 2004. ISBN 0-19-852693-8.

# Part I: Cray Fortran Commands and Directives

Part I describes the various elements that make up the Cray Fortran programming language. It includes the following chapters:

# The Trigger Environment (X1 Only) [2]

Users of Cray X1 series systems interact with the system as if all elements of the Programming Environment are hosted on the Cray X1 series mainframe, including Programming Environment commands hosted on the Cray Programming Environment Server (CPES). CPES-hosted commands have corresponding commands on the Cray X1 series mainframe that have the same names. These commands are called *triggers*. Triggers (such as the `ftn` command) are required only for the Programming Environment.

In the event that a programming or debugging tool does not work as expected, understanding the trigger environment aids administrators and end users in identifying the part of the system in which the problem has occurred.

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

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

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

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

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

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

The commands that have triggers include:

- `app2`
- `ar`
- `as`
- `c++filt`
- `c89`
- `c99`
- `cc`
- `ccp`
- `CC`
- `ftn`
- `ftnlx`
- `ftnsplit`
- `ld`
- `nm`
- `pat_build`
- `pat_help`
- `pat_report`
- `pat_run`
- `remps`

**Note:** Because of Trigger environment and X11 forwarding issues, the Cray Apprentice2 data visualization tool does not work in high-security environments where the CPES is not accessible through the customer network. This limitation is expected to be removed in a future Cray Programming Environments update package.

## 2.1 Preparing the Trigger Environment

To prepare the trigger environment for use, you must initialize your shell, load the Modules application, and then use the `module` command to load the Programming Environment module. To do so, follow these steps:

1. After you log in to a Cray X1 series system, begin your work session by initializing your shell. Cray provides initialization files for most common shells; by default, these are stored in `/opt/modules/modules/init`. For example, to initialize a C shell, enter this command:

   ```
   % source /opt/modules/modules/init/csh
   ```

2. The Modules application enables you to dynamically modify your user environment by using *modulefiles*. Each module file contains all the information needed to configure the shell for an application. While it is possible to use Cray X1 series systems without using the Modules application, doing so introduces unnecessary complexity and increases the opportunity for operator error. Initialize the Modules application by using this command:

   ```
   % module use /opt/PE/modulefiles
   ```

3. After the Modules application is initialized, use the `module` command to load the complete and current Programming Environment module:

   ```
   % module load PrgEnv
   ```

   The Programming Environment module contains your compilers, libraries, development tools, man pages, and various other component modules, and sets up the environment variables necessary to find the include files, libraries, and product paths on the CPES and the Cray X1 series system.

   As you become more familiar with the Programming Environment, you can choose to add or subtract individual modules, but as a rule, the easiest way to avoid many common problems is to start by loading the complete `PrgEnv` module.

   **Note:** Cray man pages are packaged in the modules with the software they document. The man pages do not become available until **after** you have loaded the appropriate module.

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

```
module list
```

If you have questions about setting up the Programming Environment, contact your system support staff.

## 2.2 Working in the Programming Environment

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

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

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

# Invoking the Cray Fortran Compiler [3]

This chapter describes the `ftn` command, which invokes the Cray Fortran compiler. The `ftn`(1) man page contains information from this chapter in an abbreviated form.

> **Note:** If the information contained in this manual differs from the `ftn`(1) man page, the information in the man page overrides the information in this manual.

The following files are produced by or accepted by the Cray Fortran compiler:

| File | Type |
|---|---|
| `a.out` | Default name of the executable output file. See the `-o` *`out_file`* option for information about specifying a different name for the executable file. |
| *file*`.a` | Library files to be searched for external references or modules. |
| *file*`.cg` and *file*`.opt` | Files containing decompilation of the intermediate representation of the compiler. These listings resemble the format of the source code. These files are generated when the `-rd` option is specified. |
| *file*`.f` or *file*`.F` | Input Fortran source file in fixed source form. If *file* ends in `.F`, the source preprocessor is invoked. |
| *file*`.f90`, *file*`.F90`, *file*`.ftn`, *file*`.FTN` | Input Fortran source file in free source form. If *file* ends in `.F90` or `.FTN`, the source preprocessor is invoked. |
| *file*`.i` | File containing output from the source preprocessor. |
| *file*`.lst` | Listing file. |
| *file*`.o` | Relocatable object file. |
| *file*`.s` | Assembly language file. |
| *file*`.L` | File containing binary code and generated assembly language output. |

*file*.T CIF output file.

*modulename*.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.

The syntax of the ftn command is as follows:

```
ftn [-A module_name[, module_name ] ...] [-b  bin_obj_file]
[-c] [-C cifopts] [-d disable] [-D identifier[= value]]
[-e enable] [-f source_form]
[-F] [-g] [-G debug_lvl ] [-h arg], [-I  incldir]
[-J dir_name] [-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  asm_file] [ -T] [-U identifier[, identifier] ...]
[-v] [-V] [-Wphase,"opt..."]
[-x  dirlist] [-X npes] [-Yphase,dirname] [-Z] [--]
sourcefile [sourcefile ...]
```

**Note:** Some default values shown for ftn command options may have been changed by your site. See your system support staff for details.

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

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

## 3.2 -b *bin_obj_file*

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

Only one input file is allowed when the `-b` *`bin_obj_file`* option is specified. If you have more than one input file, use the `-c` option to disable the load 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 your program is not saved after the load is completed.

If both the `-b` *`bin_obj_file`* and `-c` options are specified on the `ftn` command line, the load 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 Section 3.3, page 17.

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.

## 3.3 `-c`

The `-c` option disables the load step and saves the binary object file version of your 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. By default, this option is off.

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

If both the `-b` *`bin_obj_file`* and `-c` options are specified on the `ftn` command line, the load 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 `-b` *`bin_obj_file`* option, see Section 3.2, page 16.

If both the `-o` *`out_file`* and the `-c` option are specified on the `ftn` command line, the load step is disabled and the binary file is written to the *out_file* specified as an argument to `-o`. For more information about the `-o` *`out_file`* option, see Section 3.20, page 60.

## 3.4 `-C` *`cifopts`*

The `-C` *`cifopts`* option creates one compiler information file (CIF) for each source file. You can specify "`a`" for the *cifopts* argument, which writes all possible CIF information.

The compiler places each CIF in *file*`.T`, where *file* is the name of the source file and `.T` is the CIF suffix. The `-r` option overrides the `-C` option, if both are used.

By default, the `ftn` command does not create a CIF. You must enable the `-C` option to create a CIF. The CIF can be used as input to the `ftnlx` command.

## 3.5 `-d` *`disable`* and `-e` *`enable`*

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`. Table 1 shows the arguments to use for *disable* or *enable*.

Table 1. Compiling Options

| *args* | **Action, if enabled** |
|---|---|
| `0` | Initializes all undefined local numeric stack 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 variables are initialized upon each execution of each procedure. Enabling this option can help identify problems caused by using uninitialized numeric and logical variables.<br><br>Default: disabled |
| `a` | Aborts compilation after encountering the first error.<br><br>Default: disabled |
| `B` | Generates binary output. If disabled, inhibits all optimization and allows only syntactic and semantic checking.<br><br>Default: enabled |
| `c` | Interface checking: use Cray's system modules to check library calls in a compilation. If you have a procedure with the same name as one in the library, you will get errors as the compiler does not skip user-specified procedures when performing the checks.<br><br>Default: disabled |

| *args* | **Action, if 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 appearing in column 1 of your source with a blank and treats the entire line as a valid source line. This feature can be useful, for example, during debugging if you want to insert PRINT statements.<br><br>When disabled, a D or d character in column 1 is treated as a comment character.<br><br>Default: disabled |
| D | Turns on all debugging information. This option is equivalent to specifying these options: -O0, -g, -m2, -R aCEbcdspi, and -rl. See also -ed.<br><br>Default: disabled |
| E | The -eE option allows existing declarations to duplicate the declarations contained in a used module. Therefore, you do not have to modify the older code by removing the existing declarations. Because the declarations are not removed, the use associated objects will duplicate declarations already in the code, which is not standard conforming. However, this option allows the compiler to accept these statements as long as the declarations match the declarations in the module.<br><br>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. |

| *args* | **Action, if enabled** |
|---|---|
| | 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.<br>`module m`<br>`interface`<br>`  subroutine one(r)`<br>`  real :: r`<br>`  end subroutine`<br><br>`  function two()`<br>`  integer :: two`<br>`  end function`<br>`end interface`<br>`end module`<br><br>`program older`<br>`use m  !Or use -Am on the compiler command line`<br>`external one  !Use associated objects`<br>`integer :: two  !in declarative statements`<br><br>`call one(r)`<br>`j = two()`<br>`end program`<br><br>Default: disabled |
| 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 recommend branching into a `DO` construct, but if you specify `-eg`, the code will compile.<br><br>Default: disabled |

| *args* | **Action, if enabled** |
|---|---|
| h | Enables support for 8-bit and 16-bit `INTEGER` and `LOGICAL` types that use explicit kind or star values.<br><br>By default (`-dh`), data objects declared as `INTEGER(kind=1)`, `INTEGER(kind=2)`, `LOGICAL(kind=1)`, or `LOGICAL(kind=2)` are 32 bits long. When this option is enabled (`-eh`), data objects declared as `INTEGER(kind=1)` or `LOGICAL(kind=1)` types are 8 bits long, and objects declared as `INTEGER(kind=2)` and `LOGICAL(kind=2)` are 16 bits long. These objects are fully vectorizable depending on the operations performed, but Cray discourages their use because their resultant performance is less than the performance of their 32-bit counterparts.<br><br>8- and 16-bit objects are fully vectorizable when they are used in one of the following operations within a vector context:<br>• Reads of 8- and 16-bit variables<br>• Writes to 8- and 16-bit variables, except arrays<br>• Use of 8- and 16-bit variables as targets in a reduction loop. For example, c is an 8-bit object in this program fragment:<br>`integer :: i`<br>`integer(kind=1) :: a(100), c`<br><br>`c = 0`<br><br>`do i=1,100`<br>`  c = c + a(i)  ! This will vectorize`<br>`end do`<br><br>Default: disabled |
| 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.<br><br>Default: disabled |
| j | Executes `DO` loops at least once.<br><br>Default: disabled |

| *args* | **Action, if enabled** |
|---|---|
| L | Allows zero-trip shortloops (that is, shortloops that do not execute) and allows the use of the !DIR$ SHORTLOOP directive on loops that may have a zero-trip count. For more information, see Section 5.2.14, page 107.<br><br>Default: disabled |
| m | Causes the compiler to create and search .mod files when compiling modules and satisfying module references.<br><br>**Note:** The compiler creates modules through the MODULE statement. A module is referenced with the USE statement.<br><br>When the option is disabled, the compiler creates and searches .o files when compiling modules and satisfying module references.<br><br>The .mod files are named *modulename*.mod where *modulename* is the name of the module specified in the MODULE statement or the USE statement.<br><br>You cannot mix the .mod files with .o files in the same directory or specify both on the same ftn command line; however, system modules will work with either the -e m or -d m option.<br><br>By default, module files are written to the directory from which the ftn command is entered. You can use the -J *dir_name* option to specify an alternate output directory. For more information about the -J *dir_name* option, see Section 3.13, page 32.<br><br>Default: disabled |
| n | Generates messages to note all nonstandard Fortran usage.<br><br>Default: disabled |
| o | Display to stderr the optimization options used by the compiler for this compilation.<br><br>Default: disabled |

| *args* | **Action, if enabled** |
|---|---|
| p | Enables double precision arithmetic.<br><br>The `-dp` option can only be used when the default data size is 64 bits (that is, the `-s default64` or `-sreal64` option is used).<br><br>When this option is disabled, variables declared on a `DOUBLE PRECISION` statement and constants specified with the `D` exponent are implicitly converted to default real type. This causes arithmetic operations and intrinsics involving these variables to have a default real type rather than a double-precision real type. Similarly, variables declared on a `DOUBLE COMPLEX` statement and complex constants specified with the `D` exponent are implicitly mapped to the complex type in which each part has a default real type. Specific double precision and double complex intrinsic procedure names are mapped to their single precision equivalents.<br><br>Default: enabled |
| P | Performs source preprocessing on Fortran source files, but does not compile (see Section 3.39, page 80 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 applied to the source program. For more information about source preprocessing, see Chapter 7, page 157.<br><br>Default: disabled |
| q | Aborts compilation if 100 or more errors are generated.<br><br>Default: enabled |
| Q | Controls whether or not 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 (for example, library entry point names).<br><br>Default: disabled |

| *args* | **Action, if enabled** |
|---|---|
| R | Compiles all functions and subroutines as if they had been defined with the RECURSIVE attribute.<br><br>Default: disabled |
| s | Scale the values of all KIND=4 *count* and *count_rate* arguments for the SYSTEM_CLOCK intrinsic function. Since the value of a 32-bit *count* argument can quickly wrap around to zero, the value of *count* is scaled down by a factor of 100. KIND=4 *count_rate* is scaled in the same way. The Fortran Standard allows using different kind arguments to *count* and *count_rate*, so this scaling can be disabled. Care should be taken to make sure *count* and *count_rate* are the same kind if this scaling is enabled.<br><br>Default: enabled |
| S | Generates assembly language output and saves it in *file*.s. When the -eS option is specified on the command line with the -S *asm_file* option, the -S *asm_file* option overrides the -eS option.<br><br>Default: disabled |
| 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 upon 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 Section 5.7.3, page 135.<br><br>Default: disabled |

| *args* | **Action, if enabled** |
|---|---|
| w | Enables support for automatic memory allocation for allocatable variables and arrays that are on the left hand side of intrinsic assignment statements.<br><br>The option can potentially decrease run-time performance, even when automatic memory allocation is not needed. It will affect optimizations for a code region containing an assignment to allocatable variables or arrays. For example, it could easily prevent loop fusion for multiple array syntax assignment statements with the same shape.<br><br>Default: disabled. |
| y | Adds information into the binary files that allows the loader to find the modules when used in subsequent compiles. The `-dy` option disables this information.<br><br>Enabling this option is useful if the binary files for the Fortran modules are not moved prior to the load step. The loader can then find these binaries without the user adding them to the load line. If the module binary files will be moved before the load step, this option should be disabled and the module binary files must be explicitly specified on the load line. Often this is the case when module binaries are added to a library archive file.<br><br>Default: enabled |
| z | Performs source preprocessing and compilation on Fortran source files (see Section 3.39, page 80 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 Chapter 7, page 157.<br><br>Default: disabled |

## 3.6 -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. For more information about the -U option, see Section 3.28, page 76.

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, or *file*.FTN.
- The -eP or -eZ options have been specified.

For more information about source preprocessing, see Chapter 7, page 157.

## 3.7 -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 Section 5.6.2, page 132).

The default source form is fixed for input files that have the .f or .F suffix. The default source form is free for input files that have the .f90, .F90, .ftn, or .FTN suffix. Note that the Fortran standard has declared fixed source form to be obsolescent.

If the file has a .F, .F90, or .FTN suffix, the source preprocessor is invoked. See Chapter 7, page 157 about preprocessing.

## 3.8 -F

The -F option enables macro expansion throughout the source file. Typically, macro expansion occurs only on source preprocessing directive lines. By default, this option is off.

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*`.FTN`.
- The `-eP` or `-eZ` option was specified.

For more information about source preprocessing, see Chapter 7, page 157.

## 3.9 `-g`

The `-g` option provides debugging support identical to specifying the `-G0` option. By default, this option is off.

## 3.10 `-G` *`debug_lvl`*

The `-G` *`debug_lvl`* option generates a debug symbol table and establishes a debugging level. The debugging level determines the points at which breakpoints can be set. The frequency and position of breakpoints can curtail optimization partially or totally. At higher debugging levels, fewer breakpoints can be set, but optimization is increased. By default, this option is off. Enter one of the following arguments for *debug_lvl*:

| *debug_lvl* | Support |
|---|---|
| `0` | Breakpoints can be set at each line. This level of debugging is supported when optimization is disabled (when `-O0`, `-O ipa0`, `-O scalar0`, `-O stream0`, `-O task0`, and `-O vector0` are in effect).<br><br>If `-G0` has been specified on the command line along with an optimization level other than `-O0`, `-O ipa0`, `-O scalar0`, `-O stream0`, `-O task0`, or `-O vector0`, the compiler issues a message and disables most optimization. Array syntax statements vectorize at this level. This level can also be obtained by specifying the `-g` option. |

| | |
|---|---|
| 1 | Allows block-by-block debugging, with the exception of innermost loops. Streaming is disabled (equivalent to `-O stream0`) (X1 only). You can place breakpoints at statement labels on executable statements and at the beginning and end of block constructs (such as `IF`/`THEN`/`ELSE` blocks, `DO`/`END DO` blocks, and at `SELECT CASE`/`END SELECT` blocks). This level of debugging can be specified when `-O 0` or `-O 1` is specified. Disables some scalar optimization and all loop nest restructuring.<br><br>This *debug_lvl* allows vectorization of some inner loops and most array syntax statements. Vectorization is equal to that performed when `-O vector1` is in effect. |
| 2 | Allows post-mortem debugging. No breakpoints can be set. 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. |

## 3.11 -h *arg*

The `-h` *arg* allows you to access various compiler functionality. For more information about what to specify for *arg*, see the following subsections.

### 3.11.1 -h command

The `-h command` option provides another way to access the functionality of the `-O command` compiler option. For more information about `-O command`, see Section 3.19.4, page 39.

The `-h command` option is offered as a convenience to those who mix Fortran and C and/or C++ code because the Cray C and Cray C++ compilers have the same option.

### 3.11.2 -h cpu=*target_system*

The `-h cpu=`*target_system* option specifies the Cray X1 or X2 systems on which the absolute binary file is to be executed, where *target_system* can be one of `cray-x1`, `cray-x1e` or `cray-x2`.

Default: `cray-x1 on X1 systems; cray-x2 on X2 systems`

The target system may also be specified using the CRAY_PE_TARGET environment variable. For more information, see Section 4.1.2, page 82.

**Note:** There are no differences between the code produced for the `cray-x1` and `cray-x1e` targets.

### 3.11.3 `-h gen_private_callee` (X1 only)

The `-h gen_private_callee` option provides another way to access the functionality of the `-O gen_private_callee` compiler option. For more information about `-O gen_private_callee`, see Section 3.19.8, page 44.

The `-h gen_private_callee` option is offered as a convenience to those who mix Fortran and C and/or C++ code because the Cray C and Cray C++ compilers have the same option.

### 3.11.4 `-h ieee_nonstop`

The `-h ieee_nonstop` option specifies that the IEEE-754 "nonstop" floating-point environment is used. This environment disables all traps (interrupts) on floating-point exceptions, enables recording of all floating-point exceptions in the floating-point status register, and rounds floating-point operations to nearest. When this option is omitted, invalid, overflow, and divide by zero exceptions will trap and be recorded; underflow and inexact exceptions will neither trap nor be recorded; and floating-point operations round to nearest. For UNICOS/mp, this option requires release 2.5 or later.

### 3.11.5 `-h keepfiles`

The `-h keepfiles` option prevents the removal of the object (`.o`) 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 in order to instrument a program for performance analysis, if you plan to use CrayPat to conduct performance analysis experiments, you can use this option to preserve the object files.

### 3.11.6 `-h mpmd`, `-h nompmd`

The `-h mpmd` option allows program units containing Cray Fortran Co-array (CAF) code to be used with multiple program, multiple data (MPMD) applications. Only components of interrelated applications containing Cray Fortran Co-array (CAF) code must be compiled with the `-h mpmd` compiler option. The `-h nompmd` option does not add MPMD capability to CAF code.

The default is `-h nompmd`.

You can launch multiple interrelated applications with a single `aprun` or `mpirun` command. The applications must have the following characteristics:

- The applications can use MPI, SHMEM, or CAF to perform application-to-application communications. Using UPC for application-to-application communication is not supported.
- Within each application, the supported programming models are MPI, SHMEM, CAF, and OpenMP.
- (X1 only)All applications must be of the same mode; that is, they must all be MSP-mode applications or all SSP-mode applications.
- If one or more of the applications in an MPMD job use a shared memory model (OpenMP or pthreads) and need a depth greater than the default of 1, then all of the applications will have the depth specified by the `aprun` or `mpirun -d` option, whether they need it or not.

To launch multiple applications with one command, you use the `-h mpmd` compiler command option and launch them using `aprun` or `mpirun`.

For example, suppose you have created three MPI applications which contain CAF code as follows:

```
ftn -o multiabc -h mpmd a.o b.o c.o
ftn -o multijkl -h mpmd j.o k.o l.o
ftn -o multixyz -h mpmd x.o y.o z.o
```

**Note:** On Cray X1 series systems, users can launch an executable either by invoking the `aprun` command explicitly:

```
aprun /myapp
```

or implicitly (called `auto aprun`):

```
/myaprun
```

The `auto aprun` feature is not supported on Cray X2 systems.

The number of processing elements required are 128 for `multiabc`, 16 for `multijkl`, and 4 for `multixyz`.

To launch all three applications simultaneously, you would enter:

```
mpirun -np 128 multiabc : -np 16 multijkl : -np 4 multixyz
```

### 3.11.7 `-h msp` (X1 only)

The `-h msp` option provides another way to access the functionality of the `-O msp` compiler option. For more information about `-O msp`, see Section 3.19.14, page 50.

The `-h msp` option is offered as a convenience to those who mix Fortran and C and/or C++ code because the Cray C and Cray C++ compilers have the same option.

### 3.11.8 `-h ssp` (X1 only)

The `-h ssp` option provides another way to access the functionality of the `-O ssp` compiler option. For more information about `-O ssp`, see Section 3.19.21, page 55.

The `-h ssp` option is offered as a convenience to those who mix Fortran and C and/or C++ code because the Cray C and Cray C++ compilers have the same option.

## 3.12 `-I` *`incldir`*

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.

You must specify an `-I` option for each directory you want searched. Directories can be specified in *incldir* as full path names or as path names relative to the working directory. By default, only the system directories are searched.

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

```
% ftn -I /usr/local/sun -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.

## 3.13 -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, the module files are written to the directory from which the ftn command was entered.

The compiler will automatically search the *dir_name* directory for modules to satisfy USE statements by giving the *dir_name* path to the -p *module_site* option. You do not need to explicitly use the -p option for the compiler to do this. The compiler places this -p *module_site* option on the end of the command line.

An error is issued if the -em option is not specified when the -J *dir_name* is used.

## 3.14 -l *libname*

The -l *libname* option directs the loader to search for the specified object library file when loading an executable file. To request more than one library file, specify multiple -l options.

The loader searches for libraries by prepending *ldir*/lib on the front of *libname* and appending .a on the end of it, for each *ldir* that has been specified by using the -L option. It uses the first file it finds. See also the -L option.

For more information about library search rules, see Section 3.15, page 32.

## 3.15 -L *ldir*

The -L *ldir* option directs the loader to look for library files in directory *ldir*. To request more than one library directory, specify multiple -L options.

The loader searches for library files in directory *ldir* before searching the default directories: `/opt/ctl/libs` and `/lib`.

For example, if `-L ../mylib`, `-L /loclib`, and `-l m` are specified, the loader searches for the following files and uses the first one found:

```
../mylibs/libm.a
/loclib/libm.a
/opt/ctl/libs/libm.a
/lib/libm.a
```

See the `ld`(1) man page for more information about library searches.

For information about specifying module locations, see Section 3.21, page 60.

## 3.16 `-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.

| *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 your 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 Section 3.17, page 34.

To obtain messages regarding nonstandard Fortran usage, specify `-e n`. For more information about this option, see Section 3.5, page 18.

## 3.17 `-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. You cannot suppress or alter the severity of error-level messages with this option.

To suppress messages, specify one or more integer numbers that correspond to the Cray Fortran compiler messages you want 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.

## 3.18 `-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 or 255.

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

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

By default, lines are 72 characters wide for fixed-format sources and 132 characters wide for free-form sources.

## 3.19 `-O` *`opt`* `[,`*`opt`*`]` ...

The `-O` *`opt`* option specifies optimization features. You can specify more than one `-O` option, 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.

**Note:** The `-eo` option or the `ftnlx` command displays all the optimization options the compiler uses at compile time.

The `-O 0`, `-O 1`, `-O 2`, and `-O 3` options allow you to specify a general level of optimization that includes vectorization, scalar optimization, inlining, and streaming (X1 only). 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, inlining, and streaming (X1 only). 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. For more information about directives, see *Optimizing Applications on Cray X1 Series Systems* and *Optimizing Applications on Cray X2 Systems*.

Figure 2, page 36 shows the relationships between some of the `-O` *`opt`* values.

| | scalar0 | scalar1 | scalar2 | scalar3 | vector0 | vector1 | vector2 | vector3 | task0 | task1 | stream0 | stream1 | stream2 | stream3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low compile cost | X | | | | X | | | | X | X | X | | | |
| Moderate compile cost | | X | X | | | X | X | | | | | | | |
| Potentially high compile cost | | | | X | | | | X | | | | X | X | X |
| No numerical differences from serial execution (no vector/stream reductions) | | | | | X | | | | | | X | | | |
| Potential numerical differences from serial execution (vector/stream reductions) | | | | | | X | X | X | | | | X | X | X |
| Potential numerical differences from unoptimized execution (operator reassociation) | | X | X | X | | | | | | | | | | |
| No optimizations that may create exceptions | X | X | | | X | X | | | X | X | X | X | X | X |
| Optimizations that may create exceptions | | | X | X | | | X | X | | | | | | |
| Implies at least scalar1 | | | | | | X | | | | X | | X | X | X |
| Implies at least scalar2 | | | | | | | X | X | | | | | X | X |
| Loop nest restructuring | | X | X | X | | | X | X | | | | X | X | X |
| Vectorize array syntax statements | | | | | X | X | X | X | | | | | | |
| Vectorize/stream only inner loops | | | | | | X | | | | | | X | | |
| OpenMP disabled | | | | | | | | | X | | | | | |

Figure 2. Optimization Values

**Note:** The four columns in the table above (stream0, stream1, stream2, and stream3) apply only to the X1 series systems.

### 3.19.1 `-O` *n*

The `-O`*n* option performs general optimization at these levels: 0 (none), 1 (conservative), 2 (moderate, default), and 3 (aggressive).

- The `-O 0` option inhibits optimization including inlining. This option's characteristics include low compile time, small compile size, and no global scalar optimization.

  Most array syntax statements are vectorized, but all other vectorizations are disabled.

- The `-O 1` option specifies conservative optimization. This option's characteristics include 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 `!$OMP` directives are recognized. The `-O 1` option enables automatic multistreaming of array syntax and entire loop nests (X1 only).

- The `-O 2` option specifies moderate optimization. This option's characteristics include 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.

- The `-O 3` option specifies aggressive optimization. This option's characteristics include a potentially larger compile size, longer compile time, 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 or multistreaming (X1 only) reductions.

### 3.19.2 `-O aggress`, `-O noaggress`

The `-O aggress` option causes the compiler to treat a program unit (for example, a subroutine or a function) as a single optimization region. Doing so can improve the optimization of large program units by raising the limits for internal tables, which increases opportunities for optimization. This option increases compile time and size.

The default is `-O noaggress`.

### 3.19.3 `-O cache`*n*

The `-O cache`*n* option specifies the following levels of automatic cache management. The default on Cray X1 series systems is `-O cache0`. The default on Cray X2 systems is `-O cache2`.

- `-O cache0` specifies no automatic cache management; all memory references are allocated to cache in an exclusive state. Cache directives are still honored. Characteristics include low compile time.

  The `-O cache0` option is compatible with all scalar, vector, and (X1 only) stream optimization levels.

- `-O cache1` specifies conservative automatic cache management. Characteristics include moderate compile time. Data are placed in the cache when the possibility of cache reuse exists and the predicted cache footprint of the datum in isolation is small enough to experience the reuse.

  The `-O cache1` option requires at least `-O vector1`.

- `-O cache2` specifies moderately aggressive automatic cache management. Characteristics include moderate compile time. Data are placed in the cache when the possibility of cache reuse exists and the predicted state of the cache model is such that the datuml will experience the reuse.

  The `-O cache2` option requires at least `-O vector1`.

- `-O cache3` specifies aggressive automatic cache management. Characteristics include potentially high compile time. Data are placed in the cache when the possibility of cache reuse exists and the allocation of the datum to the cache is predicted to increase the number of cache hits.

  The `-O cache3` option requires at least `-O vector1`.

### 3.19.4 -O command

The X1 and X2 implementations of this option are described below in separate sections.

(The following section applies to the X1 series only.)

The command mode option (`-O command`) allows you to create commands for Cray X1 series systems to supplement commands developed by Cray. Command mode is not suitable for user applications or use with the `aprun` command.

The commands created with the command mode option cannot multistream, but will run serially on a single-streaming processor (SSP) within a support node. These commands will execute immediately without assistance from `psched`.

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

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

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

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

Programs linked with the `-O ssp` option and `-O command` must have been previously compiled with the `-O command` option. That is, do not link object files built with the command mode option with object files that did not use the option.

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

```
% ftn -O command -O vector0 -o fierce fierce.ftn
```

**Note:** The `-h command` option is another name for this option.

(The following section applies to the X2 only.)

The command mode option (`-O command`) allows you to create commands for Cray X2 systems to supplement commands developed by Cray. Commands can be run on application nodes using option `-n1` to specify a single process. Executing commands on multiple processes is not supported.

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

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

```
% ftn -O command -o fierce fierce.ftn
```

**Note:** The `-h command` option is another name for this option.

### 3.19.5 `-O fp`*n*

The `-O fp` option allows you to control the level of floating-point optimizations. The *n* argument controls the level of allowable optimization; 0 gives the compiler minimum freedom to optimize floating-point operations, while 3 gives it maximum freedom. The higher the level, the less the floating-point operations conform to the IEEE standard.

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

Generally, this is the behavior and usage for each `-O fp` level:

- `-O fp0` causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode (`-O fp2`). When this level is specified, many identity optimizations are disabled, executable code is slower than higher floating-point optimization levels, floating point reductions are disabled, and a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow.

  The `-O fp0` option should only be used when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance.

- `-O fp1` performs various, generally safe, IEEE non-conforming optimizations, such as folding `a == a` to `true`, where `a` is a floating point object. At this level, floating-point reassociation[1] is greatly limited, which may affect the performance of your code.

  The `-O fp1` options should only be used when your code pushes the limits of IEEE accuracy, or requires substantial IEEE standard conformance.

- `-O fp2` includes optimizations of `-O fp1`. This is the default.

- `-O fp3` includes optimizations of `-O fp1` and `-O fp2`.

  The `-O fp3` option should be used when performance is more critical than the level of IEEE standard conformance provided by `-O fp2`.

1 An example of reassociation is when `a+b+c` is rearranged to `b+a+c`, where `a`, `b`, and `c` are floating point variables.

Table 2 compares the various optimization levels of the `-O fp` option (levels 2 and 3 are usually the same). The table lists some of the optimizations performed; the compiler may perform other optimizations not listed. If multiple `-h fp` options are used, the compiler will use only the rightmost option and will issue a message indicating such.

Table 2. Floating-point Optimization Levels

| **Optimization Type** | **0** | **1** | **2 (default)** | **3** |
|---|---|---|---|---|
| Inline selected mathematical library functions | N/A | N/A | N/A | Accuracy is slightly reduced. |
| Complex divisions | Accurate and slower | Accurate and slower | Less accurate (less precision) and faster. | Less accurate (less precision) and faster. |
| Exponentiation rewrite | None | None | Maximum performance[2] | Maximum performance[2, 3] |
| Strength reduction | Fast | Fast | Aggressive | Aggressive |
| Rewrite division as reciprocal equivalent [4] | None | None | Yes | Yes |
| Floating point reductions | Slow | Fast | Fast | Fast |
| Safety | Maximum | Moderate | Moderate | Low |

2 Rewriting values raised to a constant power into an algebraically equivalent series of multiplications and/or square roots.

3 Rewriting exponentiations ($a^b$) not previously optimized into the algebraically equivalent form `exp(b * ln(a))`.

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

### 3.19.6 `-O fusion`*n*

The `-O fusion`*n* option globally controls loop fusion and changes the assertiveness of the `FUSION` directive. Loop fusion can improve the performance of loops, though in rare cases it may degrade performance.

The *n* argument allows you to turn loop fusion on or off and determine where fusion should occur. It also affects the assertiveness of the `FUSION` directive. Use one of these values for *n*:

`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` (default)

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

For more information about loop fusion, see *Optimizing Applications on Cray X1 Series Systems* and *Optimizing Applications on Cray X2 Systems.*

### 3.19.7 `-Ogcp`*n*

The `-Ogcp`*n* option enables / disables global constant propagation, where the value of *n* toggles the optimization on (1) or off (0). This optimization is off by default. Global constant propagation is an interprocedural optimization that replaces statically initialized variables with constants. For this optimization to work, the entire executable program must be presented to the compiler at once, which requires a large amount of memory and can significantly increase compile time. If the entire executable is not presented at once, the optimization fails. Messages are issued that indicate dead ends in the call graph.

This option can be used in conjunction with the `-Oipafrom=` option. For example:

```
% ftn -Oipafrom=ipa.f -Ogcp1 t.f
```

When using the `-Oipafrom=` command line option as shown above, the compiler will only look in `ipa.f` for routine definitions to use during interprocedural analysis. To also consider `t.f` for interprocedural analysis, enter the following command:

```
% ftn -Oipafrom=t.f:ipa.f -Ogcp1 t.f
```

**Note:** Only routines in `t.f` will actually get linked into the executable. For a routine to be linked into an executable, it must be input to the compile step.

**Warning:** Duplicate definitions of a routine in the input to the compiler and in the input to `-Oipafrom=` must be identical or the behavior of the generated code is unpredictable.

### 3.19.8 `-O gen_private_callee` (X1 only)

The `-O gen_private_callee` option is used when compiling source files containing subprograms which will be called from streamed regions, whether those streamed regions are created by Cray streaming directives (CSDs), or by the use of the SSP_PRIVATE directive to cause autostreaming.

See Chapter 6, page 143 for information about CSDs or to Section 5.3.2, page 118 for information about the SSP_PRIVATE directive.

**Note:** The `-h gen_private_callee` option is another name for this option.

### 3.19.9 `-O infinitevl`, `-O noinfinitevl`

The `-O infinitevl` option assumes that the safe vector length is infinite for IVDEP directives without the SAFEVL clause. The `-O noinfinitevl` option assumes the safe vector length is the maximum vector length supported by the target for IVDEP directives without the SAFEVL or INFINITEVL clause.

See Section 5.2.6, page 100 for more information about the INFINITEVL and SAFEVL clause.

The default is `-O infinitevl`.

### 3.19.10 `-O ipa`*n* and `-O ipafrom=`*source*`[:source]` ...

Inlining is the process of replacing a user procedure call with the procedure definition itself. This saves subprogram call overhead and may allow better optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for parallelization.

The `-O ipa`*n* option specifies automatic inlining. Automatic inlining allows the compiler to automatically select, depending on the inlining level *n*, which functions to inline. Each *n* specifies a different set of heuristics. When `-O ipa`*n* is used alone, the candidates for expansion are all those functions that are present in the input file to the compile step. If `-O ipa`*n* is used in conjunction with `-O ipafrom=`, the candidates for expansion are those functions present in *source*. For an explanation of each lining level, see Table 3, page 47.

The compiler supports the following inlining modes through the indicated options:

- Automatic inlining allows the compiler to automatically select, depending on the selected inlining level, which procedures to inline.
- Explicit inlining allows you to explicitly indicate which procedures the compiler should attempt to inline.
- Combined inlining allows you to specifiy potential targets for inline expansion, while applying the selected level of inlining heuristics.

Cloning is the attempt to duplicate a procedure under certain conditions and replace dummy arguments with associated constant actual arguments throughout the cloned procedure. The compiler attempts to clone a procedure when a call site contains actual arguments that are scalar integer and/or scalar logical constants. When the constants are exposed to the optimizer, it can generate more efficient code.

Automatic cloning is enabled at `-Oipa4` and higher.

The compiler will first attempt to inline a call site. If inlining the call site fails, the compiler will attempt to clone the procedure for the specific call site.

When a clone is made, dummy arguments are replaced with associated constant values throughout the routine. The following example shows cloning in action:

```
PROGRAM TEST

CALL SAM(3, .TRUE.)  ! Call site with constants

END

SUBROUTINE SAM(I, L)
INTEGER I
LOGICAL L

IF (L) THEN
   PRINT *, I
ENDIF
END
```

Compiling the previous program with the `-O ipa4` option, the compiler produces the following program:

```
PROGRAM TEST

CALL SAM@1(3, .TRUE.)   ! This is a call to a clone of SAM.

END

! Original Subroutine
SUBROUTINE SAM(I, L)
INTEGER I
LOGICAL L

IF (L) THEN
   PRINT *, I
ENDIF
END

! Cloned subroutine
SUBROUTINE SAM@1(I, L)
INTEGER I
LOGICAL L
```

```
IF (.TRUE.) THEN          ! The optimizer will eliminate this IF test
   PRINT *, 3
ENDIF
END
```

#### 3.19.10.1 Automatic Inlining

The `-O ipa`*n* option allows the compiler to automatically decide which procedures to consider for inlining. Procedures that are potential targets for inline expansion include all the procedures within the input file to the compilation. Table 3 explains what is inlined at each level.

Table 3. Automatic Inlining Specifications

| Inlining level | Description |
|---|---|
| `0` | All inlining is disabled. All inlining compiler directives are ignored. |
| `1` | Directive inlining. Inlining is attempted for call sites and routines that are under the control of an inlining compiler directive. See Chapter 5, page 87 for more information about inlining directives. |
| `2` | Call nest inlining. Inline a call nest to an arbitrary depth as long as the nest does not exceed some compiler-determined threshold. A call nest can be a leaf routine. The expansion of the call nest must yield straight-line code (code containing no external calls) for any expansion to occur. |
| `3` | Constant actual argument inlining. This includes levels 1 and 2, plus any call site that contains a constant actual argument. This is the default inlining level. |
| `4` | Tiny routine inlining. This includes levels 1, 2, and 3, plus the inlining of very small routines regardless of where those routines fall in the call graph. The lower limit threshold is an internal compiler parameter.<br><br>Routine cloning is attempted if inlining fails at a given call site. |
| `5` | Aggressive inlining. Inlining is attempted for every call site encountered. Cray does not recommend using this level.<br><br>Routine cloning is attempted if inlining fails at a given call site. |

### 3.19.10.2 Explicit Inlining

The `-O ipafrom=`*`source`*`[:source] ...` option allows you to explicitly indicate the procedures to consider for inline expansion. The *source* arguments identify each file or directory that contains the routines to consider for inlining. Whenever a call is encountered in the input program that matches a routine in *source*, inlining is attempted for that call site.

**Note:** Blanks are not allowed on either side of the equal sign.

All inlining directives are recognized with explicit inlining. For information about inlining directives, see Chapter 5, page 87.

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

Use one or more of the objects described in Table 4 in the *source* argument.

Table 4. File Types

| | |
|---|---|
| Fortran source files | The routines in Fortran source files are candidates for inline expansion. and must contain error-free code.<br><br>Source files that are acceptable for inlining are files that have one of the following extensions<br><br>• `.f`<br>• `.F`<br>• `.f90`<br>• `.F90`<br>• `.ftn`<br>• `.FTN` |
| Module files | When compiling with `-em` and `-Omodinline` 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. |

| | |
|---|---|
| | You cannot use the Fortran source of a module procedure as input to the `-O ipafrom=` option. |
| *dir* | A directory that contains any of the file types described in this table. |

#### 3.19.10.3 Combined Inlining

Combined inlining is invoked by specifying the `-O ipa`*n* 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 ipa`*n* option.

### 3.19.11 `-O inlinelib`

The `-O inlinelib` option causes the compiler to attempt inlining of those Cray scientific library routines that are available for inlining. At present this is a limited subset of the LibSci routines; more inlinable library routines will be added in future releases. For a report of what was inlined or not, see the `-O msgs,negmsgs` option.

This option is off by default.

### 3.19.12 `-O modinline`, `-O nomodinline`

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.

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 Section 3.3, page 17.

The default is -O modinline.

### 3.19.13 -O msgs, -O nomsgs

The -O msgs option causes the compiler to write optimization messages to stderr. These messages include VECTOR, SCALAR, INLINE, IPA , and STREAM (X1 only) messages.

When the -O msgs option is in effect, you may request that a listing be produced so that you can see the optimization messages in the listing. For information about obtaining listings, see Section 3.23, page 64.

The default is -O nomsgs.

### 3.19.14 -O msp (X1 only)

The -O msp option causes the compiler to generate code and to select the appropriate libraries to create an executable that runs on one or more *multistreaming processors* (MSPs). This is called *MSP mode*. Any code, including Cray distributed memory models, can use MSP mode.

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

```
ftn -O msp -c ...           !Produce MSP object files
ftn -O ssp -c ...           !Produce SSP object files
ftn sspA.o sspB.o msp.o ... !Link MSP and SSP object files
                            !to create an executable to run on MSPs
```

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

For more information about SSP and MSP mode, see Section 3.19.21, page 55 and *Optimizing Applications on Cray X1 Series Systems*.

This option is on by default.

**Note:** The -h msp option is another name for this option.

### 3.19.15 `-O negmsgs`, `-O nonegmsgs`

The `-O negmsgs` option causes the compiler to generate messages to `stderr` that indicate why optimizations such as vectorization, streaming (X1 only), 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.

The default is `-O nonegmsgs`.

### 3.19.16 `-O nointerchange`

The `-O nointerchange` option inhibits the compiler's attempts to interchange loops. Interchanging loops by having the compiler replace an inner loop with an outer loop can increase performance. The compiler performs this optimization by default.

Specifying the `-O nointerchange` option is equivalent to specifying a `NOINTERCHANGE` directive prior to every loop. To disable loop interchange on individual loops, use the `NOINTERCHANGE` directive. For more information about the `NOINTERCHANGE` directive, see Section 5.5.1, page 125.

### 3.19.17 `-O overindex`, `-O nooverindex`

The `-O nooverindex` option declares that there are no array subscripts which index a dimension of an array that are outside the declared bounds of that dimension. Short loop code generation occurs when the extent does not exceed the maximum vector length of the machine.

Specifying `-O overindex` declares that the program contains code that makes array references with subscripts that exceed the defined extents. This prevents the compiler from performing the short loop optimizations described in the preceding paragraph.

Overindexing is nonstandard, but it compiles correctly as long as data dependencies are not hidden from the compiler. This technique *collapses* loops; that is, it replaces a loop nest with a single loop. An example of this practice is as follows:

```
DIMENSION A(20, 20)
DO I = 1, N
   A(I, 1) = 0.0
END DO
```

Assuming that N equals 400 in the previous example, the compiler might generate more efficient code than a doubly nested loop. However, incorrect results can occur in this case if -O nooverindex is in effect.

You do not need to specify -O overindex if the overindexed array is a Cray pointee, has been equivalenced, or if the extent of the overindexed dimension is declared to be 1 or *. In addition, the -O overindex option is enabled automatically for the following extension code, where the number of subscripts in an array reference is less than the declared number:

```
DIMENSION A(20, 20)
DO I = 1, N
   A(I) = 0.0  ! 1-dimension reference;
               ! 2-dimension array
END DO
```

**Note:** The -O overindex option is used by the compiler for detection of short loops and subsequent code scheduling. This allows manual overindexing as described in this section, but it may have a negative performance effect because of fewer recognized short loops and more restrictive code scheduling. In addition, the compiler continues to assume, by default, a standard-conforming user program that does not overindex when doing dependency analysis for other loop nest optimizations.

The default is -O nooverindex.

### 3.19.18 -O pattern, -O nopattern

The -O pattern option enables pattern matching for library substitution. The pattern matching feature searches your code for specific code patterns and replaces them with calls to highly optimized routines.

The -O pattern option is enabled only for optimization levels -O 2, -O vector2 or higher; there is no way to force pattern matching for lower levels.

Specifying -O nopattern disables pattern matching and causes the compiler to ignore the PATTERN and NOPATTERN directives. For information about the PATTERN and NOPATTERN directives, see Section 5.2.8, page 102.

The default is -O pattern.

### 3.19.19 `-O scalar`*n*

The `-O scalar`*n* option specifies these levels of scalar optimization:

- `scalar0` disables scalar optimization. Characteristics include low compile time and size.

  The `-O scalar0` option is compatible with `-O task0` or `-O task1` and with `-O vector0`.

- `scalar1` specifies conservative scalar optimization. Characteristics include moderate compile time and size. Results can differ from the results obtained when `-O scalar0` is specified because of operator reassociation. No optimizations are performed that could create false exceptions.

  The `-O scalar1` option is compatible with `-O vector0` or `-O vector1`, with `-O task0` or `-O task1`, and with `-O stream0` (X1 only) or `-O stream1` (X1 only).

- `scalar2` specifies moderate scalar optimization. Characteristics include moderate compile time and size. Results can differ slightly from the results obtained when `-O scalar1` is specified because of possible changes in loop nest restructuring. Generally, no optimizations are done that could create false exceptions.

  The `-O scalar2` option is compatible with all vectorization, multistreaming, and tasking levels.

  This is the default scalar optimization level.

- `scalar3` specifies aggressive scalar optimization. Characteristics include potentially greater compile time and size. Results can differ from the results obtained when `-O scalar1` is specified because of possible changes in loop nest restructuring.

  The optimization techniques used can create false exceptions in rare instances. Analysis that determines whether a variable is used before it is defined is enabled at this level.

  The `-O scalar3` option is compatible with all tasking and vectorization levels.

### 3.19.20 `-O shortcircuit`*n*

The `-O shortcircuit`*n* option 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.

Example 1: Assume the following logical expression:

*operand1* `.AND.` *operand2*

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

Example 2: Assume the following logical expression:

*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. All .AND. and .OR. operators in these expressions are evaluated in this way. The left operand is evaluated, and if it determines the result of the operation, the right operand is not evaluated.
- -O shortcircuit3 specifies short circuiting of IF and ELSEIF logical conditions. It is an attempt to avoid making function calls. When this option is in effect, the left and right operands to .AND. and .OR. operators are examined to determine if one or the other contains function calls. If either operand has functions, short circuit evaluation is performed. The operand that has fewer calls is evaluated first, and if it determines the result of the operation, the remaining operand is not evaluated. If both operands have no calls, then no short circuiting is done. For the following example, the right operand of .OR. is evaluated first. If A==0 then ifunc() is not called:

  ```
  IF (ifunc() == 0 .OR. A==0) THEN
  ...
  ```

  -O shortcircuit3 is the default.

### 3.19.21 -O ssp (X1 only)

The -O ssp option causes the compiler to compile the source code and select the appropriate libraries to create an executable that runs on one *single-streaming processor* (SSP mode). Any code, including those using Cray distributed memory models, can use SSP mode. The executable is scheduled by psched and runs on one SSP on an application node.

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

```
ftn -O ssp -c ...  !Produce SSP object files
ftn -O ssp sspA.o sspB.o  ... !Link SSP object files
                              !to create an executable to run on a single SSP
```

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

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

For more information about SSP and MSP mode, see Section 3.19.14, page 50 and *Optimizing Applications on Cray X1 Series Systems*.

This option is off by default.

**Note:** The `-h ssp` option is another name for this option.

### 3.19.22 `-O streamn` (X1 only)

The `-O streamn` option controls the multistreaming when multistreaming is enabled. These levels can be set to no multistreaming optimization, at `-O stream0`, to aggressive multistreaming optimization at `-O stream3`. Generally, vectorized applications that execute on a one-processor system can expect to execute up to four times faster on a processor with multistreaming enabled.

At the default streaming level, `-O stream2`, the four processors SSP0, SSP1, SSP2, and SSP3 may be used by the code generated by the Fortran compiler. Automatic streaming can be turned off by using the `-O stream0` option. This does not mean that SSP1, SSP2, and SSP3 are not used during execution. These processors can still be used at times by the library routines called by the generated code. At times, the library routines may park (suspend) the SSP1, SSP2, and SSP3 processors. These SSPs are not available for other executables while code compiled with the `stream0` option enabled is executing.

The MSP optimization levels assume that certain scalar and vectorization optimization levels are also specified. If incompatible optimization levels are specified, the compiler adjusts the optimization levels used and issues a message. The various MSP optimization levels and their compatibilities with other optimizations are as follows:

- `-O stream0` inhibits automatic MSP optimizations. No MSP directives are recognized.

  The `-O stream0` option is compatible with all vectorization and scalar optimization levels.

- `-O stream1` is the same as `-O stream2`, except that stream consolidation is not done. Stream consolidation is a compiler optimization that attempts to minimize the synchronization cost of streaming.

- `-O stream2` specifies safe MSP optimization. The compiler recognizes MSP directives. The compiler automatically performs MSP optimizations on loop nests and appropriate BMM operations.

  The `-O stream2` option is compatible with `-O scalar2`, `-O scalar3`, `-O vector2`, and `-O vector3`.

  Default.

- `-O stream3` specifies aggressive MSP optimization on all code including appropriate BMM operations. The compiler recognizes MSP directives.

  The `-O stream3` option is compatible with `-O scalar2`, `-O scalar3`, `-O vector2`, and `-O vector3`.

For information about MSP directives, see Section 5.3, page 117. For information about optimizing with MSP, see *Optimizing Applications on Cray X1 Series Systems*. For more information about the effects the streaming option has on BMM operators, refer to the bmm man page.

### 3.19.23 `-O task0`, `-O task1`

The `-O task0` option causes the compiler to ignore OpenMP directives. Characteristics of this option include reduced compile time and size.

The `-O task0` option is compatible with all vectorization and scalar optimization levels.

The `-O task1` causes to compiler to recognize OpenMP directives.

The `-O task1` option is compatible with all vectorization and scalar optimization levels.

The default is `-O task1`.

### 3.19.24 `-O unroll`*n*

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

The *n* argument allows you to turn loop unrolling on or off and determine where unrolling should occur. It also affects the assertiveness of the `UNROLL` directive. Use one of these 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. That is, the compiler will unroll the loop if there is proof that the loop will benefit by unrolling.

`2` (default)

Attempt to unroll all loops (includes array syntax implied loops), except those marked with the `NOUNROLL` directive.

For more information about unrolling loops, see *Optimizing Applications on Cray X1 Series Systems*.

### 3.19.25 `-O vector`*n*

The `-O vector`*n* option specifies these levels of vectorization:

- `-O vector0` specifies very conservative vectorization. Characteristics include low compile time and small compile size.

  The `-O vector0` option is compatible with all scalar optimization levels and with `task0` or `task1`. Vector code is generated for most array syntax statements but not for user-coded loops.

- `-O vector1` specifies conservative vectorization. Characteristics include moderate compile time and size. Loop nests are restructured if scalar level > 0. Only inner loops are vectorized. No vectorizations that might create false exceptions are performed.

  The `-O vector1` option is compatible with `-O task0` or `-O task1` and with `-O scalar1`, `-O scalar2`, `-O scalar3`, or `-O stream1` (X1 only) .

- `-O vector2` specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured.

  The `-O vector2` option is compatible with `-O scalar2` or `-O scalar3` and with `-O task0`, `-O task1`, `-O stream0` (X1 only), `-O stream1` (X1 only), and `-O stream2` (X1 only).

  This is the default vectorization level.

- `-O vector3` 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.

  The `-O vector3` option is compatible with `-O scalar2`, `-O scalar3`, `-O stream2` (X1 only), and `-O stream3` (X1 only) and with all tasking levels.

### 3.19.26 `-O zeroinc`, `-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 default is `-O nozeroinc`, which means that you must prevent zero incrementing.

### 3.19.27 -O -h profile_generate

The profile_generate option lets you request that the source code be instrumented for profile information gathering. The compiler will insert calls and data gathering instructions to allow CPAT to gather information about the loops in a compilation unit. In order to actually get data out of this feature CPAT must be run on the resulting executable to link in the CPAT data gathering routines. If executable is not run through CPAT the inserted code will still execute, however, the gathered data will not be recorded. See the CPAT manuals for how to extract useful information for this feature.

### 3.19.28 -O -h profile_data=pgo_opt

The profile_data option instructs the compiler how to treat !PGO$ directives. There are two pgo_opt levels: sample and absolute. The default value is sample. Sample tells the compiler to treat the !PGO$ directive as information gathered from a sample program. This will keep the compiler from performing unsafe optimizations with the data. Absolute tells the compiler to treat the !PGO$ as representing the only data set that the program will ever see; this is intended for program units that either always are called with the same arguments or when it is known that the data set will not change from the experimental runs. The new directive !PGO$ loop_info is a special form of the directive !DIR$ loop_info; it tags the information as having come from profiling.

## 3.20 -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 load step is disabled and the binary file is written to the *out_file* specified as an argument to -o. For more information about the -c option, see Section 3.3, page 17.

## 3.21 -p *module_site*

The -p *module_site* option tells the compiler where to look for Fortran modules to satisfy USE statements.

**Note:** The compiler will automatically search for modules you stored in the directories specified by the `-J dir_name` option of the current compilation. You do not need to explicitly use the `-p` option to have the compiler do this. The compiler will specify a `-p` option with the *dir_name* path and place it on the end of the command line.

The *module_site* argument specifies the name of a binary file or directory to search for modules. The *module_site* specified can be an archive file, build file (`bld` file), or binary file (`.o`).

When searching files, the compiler searches files suffixed with `.o` (*file*`.o`) or library files suffixed with `.a` (*lib*`.a`) containing one or more modules. When searching a directory, the compiler searches files in the named directory that are suffixed with `.o` or `.a`, or if the `-e m` option is specified, the compiler searches `.mod` files. After searching the directory named in *module_site*, the compiler searches for modules in the current directory.

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. If you need to specify multiple binary files, library files, or directories, you must specify a `-p` option for each *module_site*. There is no limit on the number of `-p` options that you can specify. The compiler searches the binary files, library files, and directories in the order specified.

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

Example 1: 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`.

Example 2: The following example specifies binary file `murphy.o` and library file `molly.a`:

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

Example 3: In this example, assume that the following directory structure exists in your 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 your home directory, which contains the subdirectory `programs`:

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

Example 4: 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(l)
USE split
INTEGER l
l = 4
k = 5
CALL subb(l)
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, you can use the `-p` *`module_site`* option to specify one or more directories that contain module files rather than specifying every individual module file name.

## 3.22 `-Q` *`path`*

The `-Q` 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 `-S` 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
```

## 3.23 `-r` *`list_opt`*

The `-r` *`list_opt`* option generates a listing. The *list_opt* argument produces listings with commonly needed information.

If one or more input files are specified on the compiler command line, the listing is placed in *file*`.lst`.

If the `-C` option is specified with the `-r` *`list_opt`* option, the `-C` option is overridden and a warning message is generated.

The arguments for *list_opt* are shown below.

**Note:** Options `a`, `c`, `l`, `m`, `o`, `s`, and `x` invoke the `ftnlx` command. Option `d` provides a decompiled listing and is not CIF based. Option `T` retains the CIF. Options `b`, `e`, `p`, and `w` change the appearance of the listing produced by `ftnlx`.

| *list_opt* | Listing type |
|---|---|
| `-r a` | Includes all reports in the listing (including source, cross references, lint, loopmarks, common block, and options used during compilation). For more information about loopmarks, see *Optimizing Applications on Cray X1 Series Systems*. |
| `-r b` | Adds page breaks and headers to the listing report. |
| `-r c` | Listing includes a report of all `COMMON` blocks and all members of each common block. It also shows the program units that use the `COMMON` blocks. |
| `-r d` | Decompiles (translates) the intermediate representation of the compiler into listings that resemble the format of the source code. This is performed twice, resulting in two output files, at different points during the optimization process. You can use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes you can make to your Fortran source to improve its performance.<br><br>The compiler produces two decompilation listing files with these extensions per specified source file: `.opt` and `.cg`. The compiler generates the `.opt` file after applying most high level loop nest transformations to the code. The code structure of this listing most resembles your Fortran code and is readable by most users. In some cases, because of optimizations, the structure of the loops and conditionals will be significantly different than the structure in your source file. |

The `.cg` file contains a much lower level of decompilation. It is still displayed in a Fortran-like format, but is quite close to what will be produced as assembly output. This version displays the intermediate text after all multistreaming translation (X1 only), vector translation, and other optimizations have been performed. An intimate knowledge of the hardware architecture of the system is helpful to understanding this listing.

The `.opt` and `.cg` files are intended as a tool for performance analysis, and are not valid Fortran source code. The format and contents of the files can be expected to change from release to release.

The following examples (for the X2) show the listings generated when `-rd` is applied to this example:

**Note:** The column of numbers in the left-hand side of the `.opt` and `.cg` files refer to the line number in the Fortran source file.

```
!Source code, in file example.f:

subroutine example( a, b, c )
  real a(*), b(*), c(*)
  do i = 1,100
     a(i) = b(i) * c(i)
  enddo
end
```

Enter the following command:

```
% ftn -c -rd example.f
```

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

```
1.       subroutine example( a, b, c )
3.       $Induc01_N4 = 0
3. !dir$ ivdep
3.          do
4.             A(1 + $Induc01_N4) = C(1 + $Induc01_N4) * B(1 +
4.      .         $Induc01_N4)
5.             $Induc01_N4 = 1 + $Induc01_N4
3.             if ( $Induc01_N4 >= 100 ) exit
3.          enddo
6.       return
6.       end
```

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

```
1.       subroutine example( a, b, c )
3.       !  === Begin Short Vector Loop ===
4.       0[loc( A ):100:1] = 0[loc( B ):100:1] * 0[loc( C ):100:1]
3.       !  === End Short Vector Loop ===
6.       return
6.       end
```

**Note:** The entire subroutine is multistreamed.

`-r e` Expands included files in the source listing.

This option is off by default.

`-r l` Lists source code and includes lint style checking. The listing includes the `COMMON` block report (see the `-r c` option for more information about the `COMMON` block report).

`-r m` Produces a source listing with loopmark information. To provide a more complete report, this option automatically enables the `-O negmsg` option to show why loops were not optimized. If you do not require this information, use the `-O nonegmsg` option on the same command line.

Loopmark information will not be displayed if the `-d B` option has been specified.

`-r o` Show in the list file all options used by the compiler at compile time.

| | |
|---|---|
| `-r s` | Lists source code and messages. Error and warning messages are interspersed with the source lines. Optimization messages appear after each program unit. Produces 80-column output by default. |
| `-r T` | Retains *file*`.T` after processing rather than deleting it. This option may be specified in addition to any of the other options. For more information about *file*`.T`, see the `-C` option. |
| `-r w` | Produces 132-column output, which, when specified in conjunction with `-r s` or `-r x`, overrides the 80-column output that those options produce by default.<br><br>You can specify `-r w` in conjunction with either the `-r s` option or the `-r x` option. Specifying `-r w` in conjunction with any other `-r` listing option generates a warning message. |
| `-r x` | Generates a cross-reference listing. Produces 80-column output by default. |

## 3.24 `-R` *runchk*

The `-R` *runchk* option lets you specify any of a group of run-time checks for your program. To specify more than one type of checking, specify consecutive *runchk* arguments, such as: `-R ab`.

**Note:** 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 |
|---|---|
| `a` | Compares the number and types of arguments passed to a procedure with the number and types expected.<br><br>**Note:** When `-R a` is specified, some pattern matching may be lost because some of the library calls typically found in the generated code may not be present. This occurs when `-R a` is specified in conjunction with one of the following other options: `-O 2` (the default optimization level), `-O 3`, `-O ipa2`, `-O ipa3`, `-O ipa4` or `-O ipa5`. |

`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 about `NOBOUNDS`, see Section 5.6.1, page 130.

**Note:** 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 conformance checking of array operands in array expressions. Even without the `-R` option, such checking is performed during compilation when the dimensions of array operands can be determined.

`C` Passes a descriptor for the actual arguments as an extra argument to the called routine and sets a flag to signal the called routine that this descriptor is included.

`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 following directives are checked: `collapse`, `shortloop`, `shortloop128`, and the `loop_info` clauses *min_trips* and *max_trips*. Violation of a run-time check results in an immediate fatal error diagnostic.

`E` Creates a descriptor for the dummy arguments at each entry point and tests the flag from the caller to see if argument checking should be performed. If the flag is set, the argument checking is done.

`M` *msgnum*[`,`*msgnum*]...

Suppresses one or more specific run-time argument checking messages.

This suboption cannot be specified along with any other `-R` options. For example, if you want to specify `-Ra` and `-RM`, you must specify them as two separate options to the `ftn` command, as follows:

```
ftn -RM1640 -Ra otter.f.
```

You can use a comma to separate multiple message numbers. In the following example, run-time argument checking is enabled, but messages 1953 and 1946 are suppressed:

```
ftn -Ra -RM1953,1946 raccoon.f
```

n — Compares the number of arguments passed to a procedure with the number expected. Does not make comparisons with regard to argument data type (see `-R a`).

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`.

> **Note:** 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.

If argument checking is to be done for a particular call, the calling routine must have been compiled with either `-R a` or `-R C` and the called routine must have been compiled with either `-R a` or `-R E`. `-R a` is equivalent to `-R CE`. The separation of `-R a` into `-R C` and `-R E` allows some control over which calls are checked.

Libraries can be compiled with `-R E`. If the program that is calling the libraries is compiled with either `-R a` or `-R C`, library calls are checked. If the calling routines are not compiled with `-R a` or `-R C`, no checking occurs.

Slight overhead is added to each entry sequence compiled with `-R E` or `-R a` and to each call site compiled with `-R C` or `-R a`. If a call site passes the extra information to an entry that is compiled to perform checking, the checking itself costs a few thousand clock periods per call. This cost depends on the number of arguments at the call.

Some nonstandard code behaves differently when argument checking is used. Different behavior can include run-time aborts or changed results. The following example illustrates this:

```
CALL SUB1(10,15)
CALL SUB1(10)
END

SUBROUTINE SUB1(I,K)
PRINT *,I,K
END
```

Without argument checking, if the two calls in this example share the same stack space for arguments, subroutine SUB1 prints the values 10 and 15 for both calls. However, with argument checking enabled, an extra argument is added to the argument list, overwriting any previous information that was there. In this case, the second call to SUB1 prints 10, followed by an incorrect value.

If full argument checking is enabled by `-R a`, a message reporting the mismatch in the number of arguments is issued. This problem occurs only with nonstandard code in which the numbers of actual and dummy arguments do not match.

## 3.25 `-s` *size*

The `-s` *size* option allows you to modify 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 these 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 Section 3.25.2, page 74. |

`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`

**Note:** 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 Section 3.5, page 18.

`default64`

Adjust the data size of default types as follows:

- 64 bits: `REAL`, `INTEGER`, `LOGICAL`
- 128 bits: `COMPLEX`, `DOUBLE PRECISION`
- 256 bits: `DOUBLE COMPLEX`

If you used the `-s default64` at compile time, you must also specify this option when invoking the `ftn` command to call the loader.

**Note:** 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 `-eh` option. See Section 3.5, page 18.

`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`
- 128 bits: `COMPLEX` and `DOUBLE PRECISION`
- 256 bits: `DOUBLE COMPLEX`

`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 Section 3.25.2, page 74.

The default data size options (for example, `-s default64`) option does not affect the size of data that explicitly declare the size of the data (for example, `REAL(KIND=4) R`.

### 3.25.1 Different Default Data Size Options on the Command Line

You must 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.

### 3.25.2 Pointer Scaling Factor

You can specify that the compiler apply a scaling factor to integers used in pointer arithmetic involving Cray pointers so that the pointer is moved to the proper word or byte boundary. For example, the compiler views this code statement:

```
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 5. Scaling Factor in Pointer Arithmetic

| Scaling Option | 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)
```

## 3.26 `-S` *`asm_file`*

The `-S` *`asm_file`* option specifies the assembly language output file name. When `-S` *`asm_file`* is specified on the command line with either the `-e S` or `-b` *`bin_obj_file`* options, the `-e S` and `-b` *`bin_obj_file`* options are overridden.

## 3.27 `-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`).

## 3.28 -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. For more information about the -D option, see Section 3.6, page 26.

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*.FTN.
- The -e P or -e Z options have been specified.

For more information about source preprocessing, see Chapter 7, page 157.

## 3.29 -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.

## 3.30 -V

The -V option displays to the standard error file (stderr) the release version of the ftn command. Unlike all other command-line options, you can specify this option without specifying an input file name; that is, specifying ftn -V is valid.

## 3.31 -Wa"*assembler_opt*"

The -Wa"*assembler_opt*" option passes *assembler_opt* directly to the assembler. For example, -Wa"-h" passes the -h option directly 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.

## 3.32 -Wl"*loader_opt*"

The -Wl"*loader_opt*" option passes *loader_opt* directly to the loader. For example, specifying -Wl"-m" passes the argument -m directly to the loader's -m option. For more information about loader options, see the ld(1) man page.

**Note:** Cray recommends that you use the compiler to invoke the loader, because the compiler calls the loader with the appropriate default libraries. The appropriate default libraries may change from release to release.

## 3.33 -Wr"*lister_opt*"

The -Wr"*lister_opt*" option passes *lister_opt* directly to the ftnlx command. For example, specifying -Wr"-o cfile.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 you specify the -Wr"*lister_opt*" option, you must specify the -r *list_opt* option. For more information about options, see the ftnlx man page.

## 3.34 -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, OpenMP Fortran directives, and CSDs. For information about the OpenMP directives or CSDs see Chapter 8, page 167 or Chapter 6, page 143 respectively. |
| `csd` | All CSDs. See Chapter 6, page 143. |
| `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. |
| `conditional_omp` | All `C$` and `!$` conditional compilation lines. |

## 3.35 `-X` *npes*

The `-X` *npes* option specifies the number of processing elements (PEs) to use during execution. The value for *npes* ranges from 1 through 4096 inclusive.

**Note:** (X1 only) Programs compiled with the `-X` option can be executed without using the `aprun` command. If this command is used for these programs, you must specify to this command the same number of processors (*npes*) specified at compile time.

`N$PES` is a special symbol whose value is equal to the number of PEs available to your program. When the `-X` *npes* option is specified at compile time, the `N$PES` constant is replaced by integer value *npes*.

The `N$PES` constant can be used only in either of these situations:

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

One of the many uses for the N$PES symbol is illustrated in the following example, which declares the size of an array within a subroutine to be dependent upon the number of processors:

```
SUBROUTINE WORK
 DIMENSION A(N$PES)
```

Using the N$PES symbol in conjunction with the -X *npes* option allows the programmer to program the number of PEs into a program in places that do not accept run-time values. Specifying the number of PEs at compile time can also enhance compiler optimization.

## 3.36 -Y*phase*,*dirname*

The -Y*phase*,*dirname* option specifies a new directory (*dirname*) from which the designated *phase* should be executed. *phase* can be one or more of the values shown in Table 6.

Table 6. -Y*phase* Definitions

| *phase* | System phase | Command |
|---|---|---|
| 0 | Compiler | ftn |
| a | Assembler | as |
| l | Loader | ld |

## 3.37 -Z

The -Z option enables the compiler to recognize co-array syntax. *Co-arrays* are a syntactic extension to the Fortran language that offers a method for performing data passing. (Co-arrays are discussed in detail in Chapter 10.)

Data passing is an effective method for programming single-program-multiple-data (SPMD) parallel computations. Its chief advantages over message passing are lower latency and higher bandwidth for data transfers, both of which lead to improved scalability for parallel applications.

Compared to MPI and SHMEM, co-arrays provide enhanced readability and, thus, increased programmer productivity. As a language extension, the code can also be conditionally analyzed and optimized by the compiler.

## 3.38 --

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

## 3.39 *sourcefile*[*sourcefile.suffix* ...]

The *sourcefile*[*sourcefile.suffix* ...] option 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 loader will be invoked.

Preprocessor

Files having the F, F90, or FTN suffix invoke the preprocessor.

Compiler — Fortran source files having the following prefixes invoke the compiler:

- .f or .F, indicates a fixed source form file.
- .f90, .F90, .ftn, .FTN, indicates a free source form file.

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

Loader — Files with a .o extension (object files) invoke the loader. 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 loader.

You can specify object files produced by the Cray Fortran, C, C++, or assembler compilers. Object files are passed to the loader in the order in which they appear on the ftn command line. If the loader is disabled by the -b or -c option, no files are passed to the loader.

The loader allows other file types. See the -e m option in the ld man page for more information about these files.

# Environment Variables [4]

Environment variables are predefined shell variables, taken from the execution environment, that determine some of your 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 your program will perform on multiple processors. Using environment variables lets you tune the system for parallel processing without rebuilding libraries or other system software.

The variables allow you to control 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.

**Note:** 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 nondefault location, see your system support staff for path information.

## 4.1 Compiler and Library Environment Variables

The variables described in the following subsections allow you to control parallel processing at compile time.

### 4.1.1 CRAY_FTN_OPTIONS Environment Variable

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 right-most portion of the command line before the input files and binary files are listed. This allows you to set the environment variable 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 CRI_FTN_OPTIONS -G0
```

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

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

### 4.1.2 CRAY_PE_TARGET Environment Variable

The CRAY_PE_TARGET environment variable specifies the *target_system* for compilation. The command line option `-h cpu=`*`target_system`* takes precedence over the CRAY_PE_TARGET setting. The acceptable values for CRAY_PE_TARGET currently are `cray-x1`, `cray-x1e`, and `cray-x2`.

> **Note:** Currently, there are no differences in the code produced for the `cray-x1` and `cray-x1e` targets. This option was created to allow Cray to support future changes in optimization and code generation based on experience with the Cray X1E and future hardware platforms. It is possible that compilations with the `-h cpu=cray-x1e` option will not be compatible with Cray X1 machines in future releases.

### 4.1.3 FORMAT_TYPE_CHECKING Environment Variable

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: Table 17, page 202, Table 18, page 203, Table 19, page 203, and Table 20, page 203.

### 4.1.4 FORTRAN_MODULE_PATH Environment Variable

Like the Cray Fortran compiler `-p module_site` command line option, this environment variable allows you to specify 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'
```

### 4.1.5 LISTIO_PRECISION Environment Variable

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 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.

### 4.1.6 NLSPATH Environment Variable

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 catopen(3).

### 4.1.7 NPROC Environment Variable

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 make.

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 load 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.

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

### 4.1.8 TMPDIR Environment Variable

The TMPDIR environment variable specifies the directory containing the compiler temporary files. The location of the directory is defined by your administrator and cannot be changed.

### 4.1.9 ZERO_WIDTH_PRECISION Environment Variable

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.

## 4.2 OpenMP Environment Variable

OMP_THREAD_STACK_SIZE is a Cray specific OpenMP environment variable that affects programs at run time. It changes the size of the thread stack from the default size of 16 MB to the specified size. The size of the thread stack should be increased when private variables may utilize more than 16 MB of memory.

(X1 only) The requested thread stack space is allocated from the local heap when the threads are created. The amount of space used by each thread for thread stacks depend on whether you are using MSP or SSP mode. In MSP mode, the memory used is 5 times the specified thread stack size because each SSP is assigned one thread stack and one thread stack is used as the MSP common stack. For SSP mode, the memory used is one times the specified thread stack size.

(X1 only) Since memory is allocated from the local heap, you may want to consider how increasing the size of the thread stacks will affect available space in the local heap. To adjust the size of the local heap, see the X1_HEAP_SIZE and X1_LOCAL_HEAP_SIZE environment variables in the memory(7) man page.

(X2 only) The heaps on X2 do not have to be sized statically as they have to be on the X1 series systems; their sizes are adjusted as needed.

This is the format for the OMP_THREAD_STACK_SIZE environment variable:

```
OMP_THREAD_STACK_SIZE n
```

where *n* is a hex, octal or decimal integer specifying the amount of memory, in bytes, to allocate for a thread's stack.

For more information about OpenMP API, see Chapter 8, page 167.

## 4.3 Run Time Environment Variables

Run time environment variables allow you to adjust the following elements of your run time environment:

- Stack and heap sizes, see the `memory`(7) man page for more information.
- Default options for automatic `aprun`, see the `CRAY_AUTO_APRUN_OPTIONS` environment variable in the `aprun`(1) man page.
- (X1 only) Dynamic `COMMON` block, see the `X1_DYNAMIC_COMMON_SIZE` environment variable in the `ld`(1) man page.
- The field width `w` of `Fw.d` when `w` is zero on output, refer to the `ZERO_WIDTH_PRECISION` environment variable in Section 4.1.9, page 85.

# Cray Fortran Directives [5]

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

This chapter describes the Cray Fortran compiler directives. If you specify a directive while running on a system that does not support that particular directive, the compiler generates a message and continues with the compilation.

**Note:** The Cray Fortran compiler also supports the OpenMP Fortran API directives. See Chapter 8, page 167 for more information.

Section 5.1, page 90 describes how to use the directives and the effects they have on programs.

Table 7 categorizes the Cray Fortran compiler directives according to purpose and directs you to the pages containing more details.

For more information about optimization, see *Optimizing Applications on Cray X1 Series Systems*.

Table 7. Directives

| Purpose and Name | Description |
|---|---|
| Vectorization and tasking: | |
| `COPY_ASSUMED_SHAPE` | Copy arrays to temporary storage. For more information, see Section 5.2.4, page 98. |
| `HAND_TUNED` | Assert that the loop has been hand-tuned for maximum performance and restrict automatic compiler optimizations. For more information, see Section 5.2.5, page 100. |
| `IVDEP` | Ignore loop vector-dependencies that a loop might have. For more information, see Section 5.2.6, page 100. |
| `NEXTSCALAR` | Disable loop vectorization. For more information, see Section 5.2.7, page 101. |
| `PATTERN`, `NOPATTERN` | Replace or do not replace recognized code patterns with optimized library routines. For more information, see Section 5.2.8, page 102. |
| `PERMUTATION` | Declare that an integer array has no repeating values. For more information, see Section 5.2.9, page 102. |

| Purpose and Name | Description |
|---|---|
| PIPELINE | Attempt to force or inhibit software-based vector pipelining. For more information, see Section 5.2.18, page 115. |
| PREFERVECTOR | Vectorize nested loops. For more information, see Section 5.2.10, page 103. |
| PROBABILITY | Suggest the probability of a branch being executed. For more information, see Section 5.2.11, page 104. |
| SAFE_ADDRESS | Speculatively execute memory references within a loop. For more information, see Section 5.2.12, page 105. |
| SAFE_CONDITIONAL | Speculatively execute memory references and arithmetic operations within a loop. For more information, see Section 5.2.13, page 106. |
| SHORTLOOP, SHORTLOOP128 | Eliminate testing of conditional statements that terminate a loop for short loops. For more information, see Section 5.2.14, page 107. |
| LOOP_INFO | Provide loop count and cache allocation information to the optimizer to produce faster code sequences. This directive can be used to replace SHORTLOOP, SHORTLOOP128, NO_CACHE_ALLOC, or CACHE_SHARED. For more information, see Section 5.2.15, page 108. |
| UNROLL, NOUNROLL | Unroll or do not unroll loops to improve performance. For more information, see Section 5.2.16, page 112. |
| VECTOR, NOVECTOR | Vectorize or do not vectorize loops and array statements. For more information, see Section 5.2.17, page 115. |
| VFUNCTION | Declare the existence of a vectorized external function. For more information, see Section 5.2.19, page 116. |
| Multistreaming Processor (MSP) optimization (X1 only): | |
| PREFERSTREAM | Optimize the loop following the PREFERSTREAM directive, for cases where the compiler could perform MSP optimizations on more than one loop in a loop nest. For more information, see Section 5.3.1, page 118. |
| SSP_PRIVATE | Optimize loops containing procedural calls. See Section 5.3.2, page 118. |
| STREAM, NOSTREAM | Optimize or do not optimize loops and arrays. For more information, see Section 5.3.3, page 120. |

| Purpose and Name | Description |
|---|---|
| Inlining: | |
| CLONE, NOCLONE | Attempt cloning or do not attempt cloning at call sites. For more information, see Section 5.4.1, page 121. |
| INLINE, NOINLINE | Attempt to inline or do not attempt to inline call sites. For more information, see Section 5.4.2, page 122. |
| INLINENEVER, INLINEALWAYS | Never or always inline the specified procedures. For more information, see Section 5.4.3, page 122. |
| MODINLINE, NOMODINLINE | Enable or disable inlineable templates for the designated procedures. For more information, see Section 5.4.4, page 123. |
| Scalar optimization: | |
| INTERCHANGE, NOINTERCHANGE | Interchange or do not interchange the order of the loops. For more information, see Section 5.5.1, page 125. |
| NOSIDEEFFECTS | Tell the compiler that the data in the registers will not change when calling the specified subprogram. For more information, see Section 5.5.3, page 128. |
| SUPPRESS | Suppress scalar optimization of specified variables. For more information, see Section 5.5.4, page 129. |
| Local use of compiler features: | |
| BOUNDS, NOBOUNDS | Check or do not check the bounds of array references. For more information, see Section 5.6.1, page 130. |
| FREE, FIXED | Specify that the source uses a free or fixed format. For more information, see Section 5.6.2, page 132. |
| Storage: | |
| BLOCKABLE | Specify that it is legal to cache block subsequent loops. For more information, see Section 5.7.1, page 133. |
| BLOCKINGSIZE, NOBLOCKING | Assert that the loop following the directive is or is not involved in cache blocking. For more information, see Section 5.7.2, page 133. |
| STACK | Allocate variables on the stack. For more information, see Section 5.7.3, page 135. |
| Miscellaneous: | |
| CONCURRENT | Convey user-known array dependencies to the compiler. For more information, see Section 5.8.1, page 136. |

| Purpose and Name | Description |
|---|---|
| FUSION, NOFUSION | Allow you to fine-tune the selection of which DO loops the compiler should attempt to fuse. For more information, see Section 5.8.2, page 137. |
| ID | Insert an identifier string into the .o file. For more information, see Section 5.8.3, page 137. |
| IGNORE_TKR | Ignore the type, kind, and rank (*TKR*) of specified dummy arguments of a procedure interface. For more information, see Section 5.8.4, page 139. |
| NAME | Define a name that uses characters that are outside of the Fortran character set. See Section 5.8.5, page 140. |
| CACHE_EXCLUSIVE | Asserts that all vector loads with the specified symbols as the base are to be made using cache-exclusive instructions. See Section 5.2.1, page 97. |
| NO_CACHE_ALLOC | Suggest data objects that should not be placed into the cache. See Section 5.2.3, page 98. |
| CACHE_SHARED | Asserts that all vector loads with the specified symbols as the base are to be made using cache-shared instructions. For more information, see Section 5.2.2, page 97. |
| WEAK | Define a procedure reference as weak. See Section 5.8.7, page 141. |

## 5.1 Using Directives

This section describes how to use the directives and the effects they have on programs.

### 5.1.1 Directive Lines

A directive line begins with the characters CDIR$ or !DIR$. How you specify directives depends on the source form you are using, as follows:

- If you are 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 you are 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.

If you want to specify more than one directive on a line, separate each directive with a comma. Some directives require that you specify 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$).

### 5.1.2 Range and Placement of Directives

The range and placement of directives are as follows:

- The FIXED and FREE directives can appear anywhere in your source code. All other directives must appear within a program unit.
- These directives must reside in the declarative portion of a program unit and apply only to that program unit:
  - CACHE_SHARED
  - CACHE_EXCLUSIVE
  - COPY_ASSUMED_SHAPE
  - COERCE_KIND
  - IGNORE_RANK
  - IGNORE_TKR
  - INLINEALWAYS, INLINENEVER
  - NAME
  - NO_CACHE_ALLOC
  - NOSIDEEFFECTS
  - STACK
  - SSP_PRIVATE (X1 only)
  - SYMMETRIC
  - SYSTEM_MODULE
  - VFUNCTION
  - WEAK
- The following directives toggle a compiler feature on or off at the point at which the directive appears in the code. These directives are in effect until the opposite directive appears, until the directive is reset, or until the end of the program unit, at which time the command line settings become the default for the remainder of the compilation.
  - BOUNDS, NOBOUNDS

  - CLONE, NOCLONE
  - INLINE, NOINLINE
  - INTERCHANGE, NOINTERCHANGE
  - PATTERN, NOPATTERN
  - STREAM, NOSTREAM
  - VECTOR, NOVECTOR
- The SUPPRESS directive applies at the point at which it appears.
- The ID directive does not apply to any particular range of code. It adds information to the *file*.o generated from the input program.
- The following directives apply only to the next loop or block of code encountered lexically:
  - BLOCKABLE
  - BLOCKINGSIZE, NOBLOCKING
  - CONCURRENT
  - HAND_TUNED
  - INTERCHANGE, NOINTERCHANGE
  - IVDEP
  - NEXTSCALAR
  - PERMUTATION
  - PIPELINE, NOPIPELINE
  - PREFERSTREAM
  - PREFERVECTOR
  - PROBABILITY
  - SAFE_ADDRESS
  - SAFE_CONDITIONAL

  - SHORTLOOP, SHORTLOOP128
  - LOOP_INFO
  - UNROLL, NOUNROLL
- The MODINLINE and NOMODINLINE 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.

### 5.1.3 Interaction of Directives with the -x Command Line Option

The -x option on the ftn command accepts one or more directives as arguments. When your input is compiled, the compiler ignores directives named as arguments to the -x option. If you specify -x all, all directives are ignored. If you specify -x dir, all directives preceded by !DIR$ or CDIR$ are ignored.

For more information about the -x option, see Section 3.34, page 77.

### 5.1.4 Command Line Options and Directives

Some features activated by directives can also be specified on the ftn command line. A directive applies to parts of programs in which it appears, but a command line option applies to the entire compilation.

Vectorization, scalar optimization, streaming (X1 only), and tasking can be controlled through both command line options and directives. If a compiler optimization feature is disabled by default or is disabled by an argument to the `-O` option to the `ftn` command, the associated `!`*`prefix`*`$` directives are ignored. The following list shows Cray Fortran compiler optimization features, related command line options, and related directives:

- Specifying the `-O 0` option on the command line disables all optimization. All scalar optimization, vectorization, multistreaming (X1 only), and tasking directives are ignored.
- Specifying the `-O ipa0` option on the command line disables inlining and causes the compiler to ignore all inlining directives.
- Specifying the `-O scalar0` option disables scalar optimization and causes the compiler to ignore all scalar optimization and all vectorization directives.
- Specifying the `-O stream0` option disables MSP optimization and causes the compiler to ignore all MSP directives (X1 only).
- Specifying the `-O task0` option disables tasking and causes the compiler to ignore tasking directives.
- Specifying the `-O vector0` option causes the compiler to ignore all vectorization directives. Specifying the `NOVECTOR` directive in a program unit causes the compiler to ignore subsequent directives in that program unit that may specify vectorization.

The following sections describe directive syntax and the effects of directives on Cray Fortran compiler programs.

## 5.2 Vectorization Directives

This section describes the following directives used to control vectorization and tasking:

- CACHE_EXCLUSIVE
- CACHE_SHARED
- NO_CACHE_ALLOC
- COPY_ASSUMED_SHAPE
- HAND_TUNED
- IVDEP
- NEXTSCALAR
- PATTERN, NOPATTERN
- PERMUTATION
- PREFERVECTOR
- PROBABILITY
- SAFE_ADDRESS
- SAFE_CONDITIONAL
- SHORTLOOP, SHORTLOOP128
- LOOP_INFO
- UNROLL, NOUNROLL
- VECTOR, NOVECTOR
- PIPELINE, NOPIPELINE
- VFUNCTION

The `-O 0`, `-O scalar0`, `-O task0`, and `-O vector0` options on the `ftn` command override these directives.

### 5.2.1 Use Cache-exclusive Instructions for Vector Loads: CACHE_EXCLUSIVE

The CACHE_EXCLUSIVE directive asserts that all vector loads with the specified symbols as the base are to be made using cache-exclusive instructions. This is an advisory directive; if the compiler honors it, vector load misses cause the cache line to be allocated in an exclusive state in anticipation of a subsequent store. This directive is ignored for stores. Scalar loads and stores are also unaffected.

The primary use of this directive is to override automatic cache management decisions (see Section 3.19.3, page 38).

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

The syntax of the CACHE_EXCLUSIVE directive is:

```
!DIR$ CACHE_EXCLUSIVE symbol [, symbol]
```

*symbol* A base symbol (an array or scalar structure, but not a member reference or array element).

Examples of valid CACHE_EXCLUSIVE symbols are A, B, C. Symbols such as A%B or C(10) cannot be used as CACHE_EXCLUSIVE symbols.

### 5.2.2 Use Cache-shared Instructions for Vector Loads: CACHE_SHARED

The CACHE_SHARED directive asserts that all vector loads with the specified symbols as the base are to be made using cache-shared instructions. This an advisory directive; if the compiler honors it, vector load misses cause the cache line to be allocated in a shared state, in anticipation of a subsequent load by a different MSP (X1 only). This directive is not meaningful and will be ignored for stores. Scalar loads and stores are also unaffected. The compiler may override the directive when it determines the directive is not beneficial.

The syntax of the CACHE_SHARED directive is:

```
!DIR$ CACHE_SHARED symbol [, symbol ...]
```

*symbol* A base symbol (an array or scalar structure, but not a member reference or array element).

Examples of valid CACHE_SHARED symbols are A, B, C. Symbols such as A%B or C(10) cannot be used as CACHE_SHARED symbols.

### 5.2.3 Avoid Placing Object into Cache: NO_CACHE_ALLOC

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

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

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

This is the form of the directive:

```
!DIR$ NO_CACHE_ALLOC BASE_NAME [, BASE_NAME] ...
```

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

### 5.2.4 Copy Arrays to Temporary Storage: COPY_ASSUMED_SHAPE

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 format of this directive is as follows:

```
!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. You can use the -R c command line option to verify whether the arrays are shape conformant.

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

Assumed-shape dummy array 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 whether 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.

### 5.2.5 Limit Optimizations: HAND_TUNED

This directive asserts that the code in the loop that follows the directive has been arranged by hand for maximum performance and the compiler should restrict some of the more aggressive automatic expression rewrites. The compiler will still fully optimize, vectorize, and multistream the loop within the constraints of the directive.

The syntax of this directive is as follows:

```
!DIR$ HAND_TUNED
```

**Warning:** Exercise caution when using this directive and evaluate code performance before and after using it. The use of this directive may severely impair performance.

### 5.2.6 Ignore Vector Dependencies: IVDEP

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 to the first DO loop or DO WHILE loop that follows the directive. The directive applies to only the first loop that appears after 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 you to reference referencing arrays in a compact manner. Array syntax allows you to use 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.

Whether or not IVDEP is used, conditions other than vector dependencies can inhibit vectorization. The format of this directive is as follows:

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

*vlen* Specifies 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.

If no vector length is specified on the Cray X1 series or X2 systems, 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 Fortran 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 Fortran compiler does not reorder loop statements. For information about vector dependencies, see *Optimizing Applications on Cray X1 Series Systems*.

### 5.2.7 Specify Scalar Processing: `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. The format of this directive is as follows:

```
!DIR$ NEXTSCALAR
```

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

**Note:** The `NEXTSCALAR` directive does not affect multistreaming. (X1 only)

### 5.2.8 Request Pattern Matching: PATTERN and NOPATTERN

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

```
!DIR$ PATTERN

!DIR$ NOPATTERN
```

When !DIR$ NOPATTERN has been encountered, pattern matching is suspended for the remainder of the program unit or until a !DIR$ PATTERN directive is encountered. When the -O nopattern command line option (default) is in effect, the PATTERN and NOPATTERN compiler directives are ignored. For more information about -O nopattern, see Section 3.19.18, page 52.

The PATTERN and NOPATTERN directives should be specified before the beginning of a pattern.

Example: 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
```

### 5.2.9 Declare an Array with No Repeated Values: PERMUTATION

The !DIR$ PERMUTATION directive declares 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). When this directive precedes a loop to be vectorized, it may cause more efficient code to be generated.

The format for this directive is as follows:

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

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

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.

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, as in the following example:

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

### 5.2.10 Designate Loop Nest for Vectorization: PREFERVECTOR

For cases in which the compiler could vectorize more than one loop, the PREFERVECTOR directive indicates that the loop following the directive should be vectorized.

This directive can be used if there is more than one loop in the nest that could be vectorized. The format of this directive is as follows:

```
!DIR$ PREFERVECTOR
```

In the following example, both loops can be vectorized, but the compiler generates vector code for the outer `DO I` loop. Note that the `DO I` loop is vectorized even though the inner `DO J` loop was specified with an `IVDEP` directive:

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

### 5.2.11 Conditional Density: `PROBABILITY`

This directive 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. The information specified by this directive is used by interprocedural analysis and the optimizer to produce faster code sequences.

This directive can appear anywhere executable code is legal, and the syntax of this directive takes one of three forms.

```
!DIR$ PROBABILITY const
!DIR$ PROBABILITY_ALMOST_ALWAYS
!DIR$ PROBABILITY_ALMOST_NEVER
```

Where *const* is an expression between `0.0` (never) and `1.0` (always) that evaluates to a floating point constant at compilation time.

The specified probability is a hint, rather than a statement of fact. The 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. For example:

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

This example states that the probability of entering the block of code with the assignment statement is 0.3, or 30%. In turn, this means that `a(i)` is expected to be greater than `b(i)` 30% of the time as well.

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.

### 5.2.12 Allow Speculative Execution of Memory References Within Loops: SAFE_ADDRESS

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

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

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

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

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

To see the messages you must use the -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.

This is the syntax of the SAFE_ADDRESS directive:

```
!DIR$ SAFE_ADDRESS
```

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

```
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
```

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.

### 5.2.13 Allow Speculative Execution of Memory References and Arithmetic Operations: SAFE_CONDITIONAL

The SAFE_CONDITIONAL directive expands upon the SAFE_ADDRESS directive. It implies SAFE_ADDRESS and further specifies that arithmetic operations are safe, as well as memory operations.

This directive applies to scalar, vector, and multistreamed loop nests. It can improve performance by allowing the hoisting of invariant expressions from conditional code and allowing prefetching of memory references.

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

**Caution:** Incorrect use of the directive may result in segmentation faults, bus errors, excessive page faulting, or arithmetic aborts. However, it should not result in incorrect answers. Incorrect usage may result in severe performance degradation or program aborts.

The syntax of this directive is as follows:

```
!DIR$ SAFE_CONDITIONAL
```

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 you know that the condition is true at least once, then it is safe to use the SAFE_CONDITIONAL directive and execute s1*s2 speculatively.

```
      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
```

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.

### 5.2.14 Designate Loops with Low Trip Counts: SHORTLOOP, SHORTLOOP128

The SHORTLOOP directive, used before a DO or DO WHILE loop with a low trip count, allows the compiler to generate code that improves program performance by eliminating run-time tests for determining whether a vectorized DO loop has been completed. The compiler will diagnose misuse at compile time (when able) or under option -Rd at run time.

The formats of these directives are as follows:

```
!DIR$ SHORTLOOP

!DIR$ SHORTLOOP128
```

You can specify either of the preceding formats, as follows:

- If you specify `!DIR$ SHORTLOOP`, the loop trip count must be in the range $1 \leq$ *trip_count* $\leq 64$. If *trip_count* equals 0 or exceeds 64, results are unpredictable.
- If you specify `!DIR$ SHORTLOOP128`, the loop trip count must be in the range $1 \leq$ *trip_count* $\leq 128$. If *trip_count* equals zero or exceeds 128, results are unpredictable.

`SHORTLOOP` is ignored in the following cases:

- If vectorization is disabled.
- If the code in question is an array syntax assignment statement.
- If the compiler can determine that the directive is invalid. If so, a diagnostic message is issued.

The meaning of `SHORTLOOP` and `SHORTLOOP128` can be modified by using the `-eL` command. If enabled, this option changes the lower bound to allow zero-trip loops. For more information, see Section 3.5, page 18.

### 5.2.15 Provide More Information for Loops: `LOOP_INFO`

The `LOOP_INFO` directive allows additional information to be specified about the behavior of a loop. This currently includes information about the run-time trip count and hints on cache allocation strategy. The compiler will diagnose misuse at compile time (when able) or under option `-Rd` at run time.

With respect to the trip count information, the `LOOP_INFO` directive is similar to the `SHORTLOOP` or `SHORTLOOP128` directive, but provides more information to the optimizer and can produce faster code sequences. `LOOP_INFO` is used before a `DO` or `WHILE` loop with a low or known trip count.

For cache allocation hints, the `LOOP_INFO` directive can be used to override default settings or to supersede earlier `NO_CACHE_ALLOC`, `CACHE_EXCLUSIVE`, or `CACHE_SHARED` directives.

The syntax of the LOOP_INFO directive is as follows:

```
!DIR$ LOOP_INFO [min_trips(c)] [est_trips(c)] [max_trips(c)]
                [cache_ex( symbol [, symbol ...] )]
                [cache_sh( symbol [, symbol ...] )]
                [cache_na( symbol [, symbol ...] )]
                [prefer_amo ][prefer_noamo ]
                [prefetch ][noprefetch ]
```

Where min_trips is the guaranteed minimum number of trips, est_trips is the estimated or average number of trips, and max_trips is the guaranteed maximum number of trips.

The SHORTLOOP and SHORTLOOP128 directives are equivalent, respectively, to:

```
! dir$ loop_info min_trips(1) max_trips(64)
! dir$ loop_info min_trips(1) max_trips(128)
```

The cache_ex, cache_sh, and cache_na options specify symbols that are to receive the exclusive, shared, and non-allocating cache hints, respectively. If no hints are specified and no NO_CACHE_ALLOC or CACHE_SHARED directives are present, the default is exclusive.

The cache hints are local and apply only to the specified loop nest. For more information about cache_na behavior, see Section 5.2.3, page 98 . For more information about cache_sh behavior, see Section 5.2.2, page 97. The cache_ex hint can be used to override locally any earlier NO_CACHE_ALLOC or CACHE_SHARED directive.

The prefer_amo clause of the loop_info directive only has meaning on architectures that have vector atomic memory operation capability in hardware including the Cray X2. On architectures that lack this hardware, such as the Cray X1 and Cray X1E, the clause is accepted but has no effect. The prefer_amo clause instructs, but does not require, the compiler to use vector atomic memory operations as aggressively as possible, including in those cases that the compiler would normally avoid because it expects the performance to be poor. For example:

```
      subroutine p_amo( ia, ib, n )
      integer (kind=8) ia(n), ib(n)
!  The compiler avoids vector AMOs in this case for most access patterns
      do i = 1,n
        ia(i) = ia(i) + 1
      enddo
!  Direct the compiler to use vector AMOs when possible
!dir$ loop_info prefer_amo
      do i = 1,n
        ib(i) = ib(i) + 1
      enddo
      end
```

For sample test case p_amo, the compiler does not use a vector atomic memory operation for the first loop, but it does use it for the second loop because of the prefer_amo compiler clause of the loop_info directive. A message similar to the following lines is issued when messages are enabled:

```
   ib(i) = ib(i) + 1
ftn-6385 ftn: VECTOR P_AMO, File = amo.f, Line = 10
  A vector atomic memory operation was used for this statement.
```

The prefer_noamo clause instructs, but does not require, the compiler to avoid all uses of vector atomic memory operations. The compiler may, at its discretion, continue to use vector atomic memory operations if there is no alternative solution to vectorizing the loop. The compiler automatically uses vector atomic memory operations if its assessment shows that the performance will improve. For example:

```
      subroutine a_amo( a, b, c, ia, ib, n )
      integer (kind=8) ia(n), ib(n)
      integer (kind=8) a(n), b(n), c(n)
!  Compiler automatically uses a vector AMO
      do i = 1,n
        a(ia(i)) = a(ia(i)) + c(i)
      enddo
!  Instruct the compiler to avoid using a vector AMO
!dir$ loop_info prefer_noamo
      do i = 1,n
        b(ib(i)) = b(ib(i)) + c(i)
      enddo
      end
```

For sample test case a_amo, the compiler uses a vector atomic memory operation for the 'update' construct in the first loop. In the second loop, the 'prefer_noamo' clause of the loop_info directive instructs the compiler to avoid using vector atomic memory operations. Messages demonstrating the effects of these directives similar to the following lines are ssued for the two loop bodies:

```
       a(ia(i)) = a(ia(i)) + c(i)
ftn-6385 ftn: VECTOR A_AMO, File = a_amo.f, Line = 6
  A vector atomic memory operation was used for this statement.
      do i = 1,n
ftn-6371 ftn: VECTOR A_AMO, File = a_amo.f, Line = 10
  A vectorized loop contains potential conflicts due to indirect addressing
  at line 11, causing less efficient code to be generated.
```

The hardware vector atomic memory operations for the Cray X2 include 64-bit integer bitwise and, bitwise or, bitwise exclusive or, and integer addition. The compiler recognizes these and other operations that can efficiently map onto the set of instructions.

The `prefetch` clause (X2 only) instructs the compiler to preload scalar data into the first-level cache to improve the frequency of cache hits and 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 feeding 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
```

### 5.2.16 Unroll Loops: `UNROLL` and `NOUNROLL`

Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

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

The formats of these directives are as follows:

```
!DIR$ UNROLL [ n ]

!DIR$ NOUNROLL
```

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

If you specify a value for *n*, the compiler unrolls the loop by that amount. If you do not specify *n*, 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 you specify `UNROLL0`, `UNROLL1`, or `NOUNROLL`. These directives are equivalent.

The `UNROLL` directive should be placed immediately before the `DO` statement of the loop that should be unrolled.

**Note:** The compiler cannot always safely unroll non-innermost loops due to data dependencies. In these cases, the directive is ignored (see Example 1).

The UNROLL directive can be used only on loops whose iteration counts 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, at each nest level, there is only one loop and only the innermost loop contains work.

The NOUNROLL directive inhibits loop unrolling.

**Note:** Loop unrolling occurs for both vector and scalar loops automatically. It is usually not necessary to use the unrolling directives. The UNROLL directive should be limited to non-inner loops such as Example 1 in which unroll-and-jam conditions can occur. Such loop unrolling is associated with compiler message 6005. Using the UNROLL directive for inner loops may be detrimental to performance and is not recommended. Typically, loop unrolling occurs in both vector and scalar loops without need of the UNROLL directive.

### Example 1: Unrolling 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
```

### Example 2: 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
```

### Example 3: 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
```

### 5.2.17 Enable and Disable Vectorization: VECTOR and NOVECTOR

The NOVECTOR directive suppresses compiler attempts to vectorize loops and array syntax statements. NOVECTOR takes effect at the beginning of the next loop and applies to the rest of the program unit unless it is superseded by a VECTOR directive. These directives are ignored if vectorization or scalar optimization have been disabled. The formats of these directives are as follows:

```
!DIR$ VECTOR

!DIR$ NOVECTOR
```

When !DIR$ NOVECTOR has been used within the same program unit, !DIR$ 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.

The VECTOR directive affects subsequent loops. The NOVECTOR directive also affects subsequent loops, but if it is specified within the body of a loop, it affects the loop in which it is contained and all subsequent loops.

### 5.2.18 Enable or Disable, Temporarily, Soft Vector-pipelining: PIPELINE and 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 will automatically attempt 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 will either not pipeline a loop that could be pipelined, or pipeline a loop without producing performance gains. In these cases, you can use the PIPELINE or NOPIPELINE directives to advise the compiler to pipeline or not pipeline the loop immediately following the directive.

The format of the pipelining directives is as follows:

```
!DIR$ PIPELINE

!DIR$ NOPIPELINE
```

Software vector pipelining is valid only for the innermost loop of a loop nest.

The PIPELINE and NOPIPELINE directives are advisory only. While you can use the NOPIPELINE directive to inhibit automatic pipelining, and you can use the PIPELINE directive to attempt to override the compiler's decision not to pipeline a loop, you cannot force the compiler 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`). For more information about the messages issued, see the *Optimizing Applications on Cray X1 Series Systems*.

### 5.2.19 Specify a Vectorizable Function: VFUNCTION

The VFUNCTION directive declares that a vector version of an external function exists. The VFUNCTION directive must precede any statement function definitions or executable statements in a program. VFUNCTION cannot be specified for internal or module procedures. VFUNCTION cannot be specified for functions within interface blocks.

This is the format of the VFUNCTION directive:

```
!DIR$ VFUNCTION function_name [,f ] ...
```

*f* Symbolic name of a vector external function. The maximum length is 29 characters because the % character is added at the beginning and end of the name as part of the calling sequence. For example, if the function is named FUNC, the CAL vector version is spelled %FUNC%. (The scalar version is FUNC%.)

The following rules and recommendations apply to any function *f* named as an argument in a VFUNCTION directive:

- *f* cannot be declared in an EXTERNAL statement, have its interface specified in an interface body, or be specified in a PROCEDURE declaration statement.
- *f* must be written in CAL and must use the call-by-register sequence.
- Arguments to *f* must be either vectorizable expressions or scalar expressions; array syntax and array expressions are not allowed.
- A call to *f* can pass a maximum of seven single-word items or one four-word item (complex (KIND=KIND(0.0D0))). No structures or character arguments can be passed. These can be mixed in any order with a maximum of seven words total.
- *f* should not change the value of its arguments or variables in common blocks or modules. Any changed value should be for variables that are distinct from the arguments.
- *f* should not reference variables in common blocks or modules that are also used by a program unit in the calling chain.
- A call to *f* cannot occur within a WHERE statement or WHERE block.
- *f* must not have side effects or perform I/O.

Arguments to *f* are sent to the V registers that have numbers that match the arguments' ordinal numbers in the argument list: X=VFUNC(*v1*,*v2*,*v3*,*v4*). (The scalar version uses the same convention with the S registers.)

If the argument list for *f* contains both scalar and vector arguments in a vector loop, the scalar arguments are broadcast into the appropriate vector registers. If all arguments are scalar or the function reference is not in a vector loop, *f* is called with all arguments passed in S registers.

## 5.3 Multistreaming Processor (MSP) Directives (X1 only)

The MSP directives work with the -O stream*n* command line option to determine whether parts of your program are optimized for the MSP. Therefore, one of the following options must be specified on the ftn command line in order for these directives to be recognized: -O stream1 or -O stream3. The default streaming option, -O stream2, also causes recognition of the directives. For more information about the -O stream*n* command line option, see Section 3.19.22, page 56.

The MSP directives are as follows:

- PREFERSTREAM
- SSP_PRIVATE
- STREAM, NOSTREAM

The following subsections describe the MSP optimization directives.

### 5.3.1 Specify Loop to be Optimized for MSP: PREFERSTREAM

For cases in which the compiler could perform MSP optimizations on more than one loop in a loop nest, the PREFERSTREAM directive indicates that the loop following the directive is the one to be optimized. The format of this directive is as follows:

```
!DIR$ PREFERSTREAM
```

This directive is ignored if `-O stream0` is in effect.

### 5.3.2 Optimize Loops Containing Procedural Calls: SSP_PRIVATE

The SSP_PRIVATE directive allows the compiler to stream loops that contain procedural calls. By default, the compiler does not stream procedural calls contained in a loop, because the call may have *side effects* that interfere with correct parallel execution. The SSP_PRIVATE directive asserts that the specified procedure is free of side effects that inhibit parallelism and that the specified procedure, and all procedures it calls, will run on one SSP.

An implied condition for streaming loops containing a call to a procedure specified with the SSP_PRIVATE directive is that the loop body must not contain any problems that prevent parallelism. The compiler can disregard an SSP_PRIVATE directive if it detects possible *loop-carried dependencies* that are not directly related to a call inside the loop.

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

When using the SSP_PRIVATE directive, you must ensure that the procedure called within the body of the loop follows these criteria:

- The procedure does not modify data in one iteration and reference this same data in another iteration of the streamed loop. This rule applies equally to arguments, common variables, and data declared by using a SAVE statement.
- The procedure does not reference data in one iteration that is defined in another iteration.
- If the procedure modifies an argument, common variable, or data declared in a SAVE statement, the iterations cannot modify data at the same storage location. unless these variables are scoped as PRIVATE. Following the streamed loop, the content of private variables are undefined.

  The SSP_PRIVATE directive does not force the master thread to execute the last iteration of the task loop.
- If the procedure uses shared data (for example, global data, actual arguments) that can be written to and read, you must protect it with a guard (such as the CSD CRITICAL directive or the lock command) or have the SSPs access the data disjointedly (where access does not overlap).
- The procedure calls only other procedures that are capable of being called privately.
- The procedure uses the appropriate synchronization mechanism when calling I/O.

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

The SSP_PRIVATE directive can only be used in the specification part, before any executable statements. The SSP_PRIVATE directive may be used multiple times within a procedure.

This is the form of the SSP_PRIVATE directive:

```
!DIR$ SSP_PRIVATE PROC_NAME[, PROC_NAME] ...
```

*PROC_NAME* specifies one or more procedure names called from within the loops that are candidates for streaming. Procedures specified in the procedure name list retain the SSP_PRIVATE attribute throughout the entire program unit. These procedures must be compiled with the `-O gen_private_callee` option.

The following example demonstrates use of the SSP_PRIVATE directive:

```
! Code in file1.ftn
subroutine example(X, Y, P, N, M)
   dimension X(N), Y(N), P(0:M)

   !dir$ ssp_private poly_eval

   do I = 1, N
      call poly_eval( Y(I), X(I), P, M )
   enddo
end

! Code in file2.ftn.
subroutine poly_eval( Y, X, P, M )
   dimension P(0:M)

   Y = P(M)
   do J = M-1, 0, -1
      Y = X*Y + P(J)
   enddo
end
```

This example compiles the code:

```
% ftn -c -O gen_private_callee file2.ftn
% ftn file1.ftn file2.o
```

Now we run the code:

```
% aprun a.out
```

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

### 5.3.3 Enable MSP Optimization: STREAM and NOSTREAM

The STREAM and NOSTREAM directives specify whether the compiler should perform MSP optimizations over a range of code. These optimizations are applied to loops and array syntax statements. The formats of these directives are as follows:

```
!DIR$ STREAM

!DIR$ NOSTREAM
```

One of these directives remains in effect until the opposite directive is encountered or until the end of the program unit. These directives are ignored if `-O stream0` is in effect.

## 5.4 Inlining Directives

The inlining directives allow you to specify whether the compiler should attempt to inline certain subprograms or procedures. These are the inlining directives:

- `clone`, `noclone`
- `inline`, `noinline`, `resetinline`
- `inlinealways`, `inlinenever`
- `modinline`, `nomodinline`

These directives work in conjunction with the following command line options:

- `-O ipa`*n* and `-O ipafrom`, described in Section 3.19.10, page 44.
- `-O modinline` and `-O nomodinline`, described in Section 3.19.12, page 49.

The following subsections describe the inlining directives.

### 5.4.1 Disable or Enable Cloning for a Block of Code: `CLONE` and `NOCLONE`

The `clone` and `noclone` directives control whether cloning is attempted over a range of code. If `!dir$ clone` is in effect, cloning is attempted at call sites. If `!dir$ noclone` is in effect, cloning is not attempted at call sites. The formats of these directives are as follows:

```
!dir$ clone

!dir$ noclone
```

One of these directives remains in effect until the opposite directive is encountered or until the end of the program unit. These directives are recognized when cloning is enabled on the command line (`-O clone1`). These directives are ignored if the `-O ipa0` option is in effect.

### 5.4.2 Disable or Enable Inlining for a Block of Code: INLINE, NOINLINE, and RESETINLINE

The inline, noinline, and resetinline directives control whether inlining is attempted over a range of code. If !dir$ inline is in effect, inlining is attempted at call sites. If !dir$ noinline is in effect, inlining is not attempted at call sites. After either directive is used, !dir$ resetinline can be used to return inlining to the default state. These are the formats of these directives:

```
!dir$ inline
!dir$ noinline
!dir$ resetinline
```

The inline and noinline directives remain in effect until the opposite directive is encountered, until the resetinline directive is encountered, or until the end of the program unit. These directives are ignored if -O ipa0 is in effect.

### 5.4.3 Specify Inlining for a Procedure: INLINEALWAYS and INLINENEVER

The inlinealways directive forces attempted inlining of specified procedures. The inlinenever directive suppresses inlining of specified procedures. The formats of these directives are as follows:

```
!dir$ inlinealways name [, name  ] ...

!dir$ inlinenever name [, name  ] ...
```

where *name* is the name of a procedure.

The following rules determine the scope of these directives:

- A !dir$ inlinenever directive suppresses inlining for *name.* That is, if !dir$ inlinenever b appears in routine b, no call to b, within the entire program, is inlined. If !dir$ inlinenever b appears in a routine other than b, no call to b from within that routine is inlined.
- A !dir$ inlinealways directive specifies that inlining should always be attempted for *name.* That is, if !dir$ inlinealways c appears in routine c, inlining is attempted for all calls to c, throughout the entire program. If !dir$ inlinealways c appears in a routine other than c, inlining is attempted for all calls to c from within that routine.

An error message is issued if inlinenever and inlinealways are specified for the same procedure in the same program unit.

Example: The following file is compiled with `-O ipa1`:

```
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
```

### 5.4.4 Create Inlinable Templates for Module Procedures: `MODINLINE` and `NOMODINLINE`

The `MODINLINE` and `NOMODINLINE` directives enable and disable the creation of inlinable templates for specific module procedures. The formats of these directives are as follows:

```
!DIR$ MODINLINE

!DIR$ NOMODINLINE
```

**Note:** 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.

To inline module procedures, the module being used associated must have been compiled with `-O modinline`.

Example:

```
     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`.

## 5.5 Scalar Optimization Directives

The following directives control aspects of scalar optimization:

- INTERCHANGE and NOINTERCHANGE
- NOSIDEEFFECTS
- SUPPRESS

The following subsections describe these directives.

### 5.5.1 Control Loop Interchange: INTERCHANGE and NOINTERCHANGE

The loop interchange control directives specify whether or not the order of the following two or more loops should be interchanged. These directives apply to the loops that they immediately precede.

The formats of these directives are as follows:

```
!DIR$ INTERCHANGE (do_variable1,do_variable2 [,do_variable3]...)

!DIR$ NOINTERCHANGE
```

*do_variable*

Specifies two or more *do_variable* names. The *do_variable* names can be specified in any order, and the compiler reorders the loops. The loops must be perfectly nested. If the loops are not perfectly nested, you may receive unexpected results.

The compiler reorders the loops such that the loop with *do_variable1* is outermost, then loop *do_variable2*, then loop *do_variable3*.

The NOINTERCHANGE directive inhibits loop interchange on the loop that immediately follows the directive.

Example: The following code has an INTERCHANGE directive:

```
!DIR$ INTERCHANGE (I,J,K)
      DO K = 1,NSIZE1
        DO J = 1,NSIZE1
          DO I = 1,NSIZE1
             X(I,J) = X(I,J) + Y(I,K) * Z(K,J)
          ENDDO
        ENDDO
      ENDDO
```

The following code results when the INTERCHANGE directive is used on the preceding code:

```
      DO I = 1,NSIZE1
        DO J = 1,NSIZE1
          DO K = 1,NSIZE1
             X(I,J) = X(I,J) + Y(I,K) * Z(K,J)
          ENDDO
        ENDDO
      ENDDO
```

### 5.5.2 Control Loop Collapse: COLLAPSE and NOCOLLAPSE

The loop collapse directives control collapse of the immediately following loop nest or elemental array syntax statement. When the COLLAPSE directive is applied to a DO-loop nest, the loop control variables of the participating loops must be listed in order of increasing access stride. NOCOLLAPSE disqualifies the immediately following DO-loop from collapsing with any other loop; before an elemental array syntax statement, it inhibits all collapse in said statement.

```
  subroutine S(A, n, n1, n2)
   real A(n, *)
!dir$ collapse (i, j)
   do i = 1, n1
     do j = 1, n2
            A(i,j) = A(i,j) + 42.0
     enddo
   enddo
   end
```

The above yields code equivalent to the following, which should not be coded directly because as program source, it violates the Fortran language standard.

```
   subroutine S(A, n, n1, n2)
   real A(n, *)
   do ij = 1, n1*n2
          A(ij, 1) = A(ij, 1) + 42.0
   enddo
   end
```

With array syntax, the collapse directive appears as follows:

```
   subroutine S( A, B )
   real, dimension(:,:) :: A, B
!dir$ collapse
   A = B       ! user promises uniform access stride.
   end
```

In each of the above examples, the directive enables the compiler to assume appropriate conformity between trip counts and array extends. The compiler will diagnose misuse at compile time (when able); or, under option `-Rd`, at run time.

`NOCOLLAPSE` prevents the compiler from collapsing a given loop with others or from performing any loop collapse within a specified array syntax statement. Collapse is almost always desirable, so this directive should be used sparingly.

```
   subroutine S(A, n)
   dimension A(n,n)
!dir$ nocollapse
   do i = 1, n          ! disallow collapse involving i-loop.
     do j = 1, n
        A(i,j) = 1.2
     enddo
   enddo
   end
```

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. Hence, 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. This typically occurs when a computation can flow from one column of a multidimensional array into the next, viewing the array as a flat sequence. Hence, array sections such as A(:,3:7) are generally suitable for collapse, while a section like A(1:n-1,:) lacks the needed access uniformity. Care must taken when applying the collapse directive to assumed shape dummy arguments and Fortran pointers because the underlying storage need not be contiguous.

### 5.5.3 Determine Register Storage: NOSIDEEFFECTS

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

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

The format of this directive is as follows:

```
!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.

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.

### 5.5.4 Suppress Scalar Optimization: SUPPRESS

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. The format of this directive is as follows:

```
!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.

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.

Example:

```
      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.

## 5.6 Local Use of Compiler Features

The following directives provide local control over specific compiler features.

- BOUNDS and NOBOUNDS
- FREE and FIXED

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. The following subsections describe these directives.

### 5.6.1 Check Array Bounds: BOUNDS and NOBOUNDS

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.

**Note:** Bounds checking behavior differs with the optimization level. 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. The formats of these directives are as follows:

```
!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.

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.

**Note:** 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. For example:

```
      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 when it detects an out-of-bounds subscript. 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 your 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.

**Note:** 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. For example:

```
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.
```

### 5.6.2 Specify Source Form: FREE and 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. The formats of these directives are as follows:

```
!DIR$ FREE

!DIR$ FIXED
```

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

You can change source form 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.

## 5.7 Storage Directives

The following directives specify aspects of storing common blocks, variables, or arrays:

- BLOCKABLE
- BLOCKINGSIZE and NOBLOCKING
- STACK

The following sections describe these directives.

### 5.7.1 Permit Cache Blocking: BLOCKABLE Directive

The BLOCKABLE directive specifies that it is legal to cache block the subsequent loops.

The format of this directive is as follows:

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

where *do_variable* specifies the *do_variable* names of two or more loops. The loops identified by the *do_variable* names must be adjacent and nested within each other, although they need not be perfectly nested.

This directive tells the compiler that these loops can be involved in a blocking situation with each other, even if the compiler would consider such a transformation illegal. The loops must also be interchangeable and unrollable. This directive does not instruct the compiler on which of these transformations to apply.

### 5.7.2 Declare Cache Blocking: BLOCKINGSIZE and NOBLOCKING Directives

The BLOCKINGSIZE and NOBLOCKING directives assert that the loop following the directive either is (or is not) involved in a cache blocking for the primary or secondary cache.

The formats of these directives are as follows:

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

!DIR$ NOBLOCKING
```

*n1*,*n2*
: An integer number that indicates the block size. If the loop is involved in a blocking, 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 it cannot guarantee this.

  For *n1*, specify a value such that *n1* .GE. 0. For *n2*, specify a value such that *n2* .LE. $2^{30}$.

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

Example:

```
      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
```

For the preceding code, 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 tile. If it did not, add a BLOCKINGSIZE(0) directive just before loop K to specify that the compiler should generate a loop such as the following:

```
      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
```

Note that an INTERCHANGE directive can be applied to the same loop nest as a BLOCKINGSIZE directive. The BLOCKINGSIZE directive applies to the loop it directly precedes; it moves with that loop when an interchange is applied.

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

### 5.7.3 Request Stack Storage: 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 Section 3.5, page 18.

The format of this directive is as follows:

```
!DIR$ STACK
```

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.
- If it is specified within the scope of an interface body, a message is issued.

## 5.8 Miscellaneous Directives

The following directives allow you to use several different compiler features:

- CONCURRENT
- FUSION and NOFUSION
- ID
- IGNORE_TKR
- NAME
- PREPROCESS
- WEAK

### 5.8.1 Specify Array Dependencies: CONCURRENT

The CONCURRENT directive conveys array dependency information to the compiler. This directive affects the loop that immediately follows it. The CONCURRENT directive is useful when vectorization or MSP (X1 only)optimization is specified by the command line. The format of this directive is as follows:

```
!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 integeral 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 MSP optimizations, streaming (X1 only), or vectorization is requested on the command line.

Example. 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)
```

### 5.8.2 Fuse Loops: FUSION and NOFUSION

The FUSION and NOFUSION directives allow you to fine-tune the selection of which DO loops the compiler should attempt to fuse. If there are only a few loops out of many that you want to fuse, then use the FUSION directive with the -O fusion1 option to confine loop fusion to these few loops. If there are only a few loops out of many that you do not want to fuse, use the NOFUSION directive with the -O fusion2 option to specify no fusion for these loops.

These are the formats of the directives:

```
!DIR$ FUSION

!DIR NOFUSION
```

The FUSION directive should be placed immediately before the DO statement of the loop that should be fused.

For more information about loop fusion and its benefits, see *Optimizing Applications on Cray X1 Series Systems* and *Optimizing Applications on Cray X2 Systems*.

### 5.8.3 Create Identification String: ID

The ID directive inserts a character string into the *file*.o produced for a Fortran source file. The format of this directive is as follows:

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

*character_ string*

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

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

### 5.8.4 Disregard Dummy Argument Type, Kind, and Rank: `IGNORE_TKR`

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 format for this directive is as follows:

```
!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 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.

Example: 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 8 indicates what is ignored for each dummy argument.

Table 8. 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 |

### 5.8.5 External Name Mapping: NAME

The NAME directive allows you to specify 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, in the following format:

```
!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.

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

Example:

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

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

### 5.8.6 Preprocess Include File: PREPROCESS

The PREPROCESS directive allows an include file to be preprocessed 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.

This is the format of the PREPROCESS directive:

```
!DIR$ PREPROCESS [expand_macros]
```

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.

### 5.8.7 Specify Weak Procedure Reference: WEAK

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. The WEAK directive can prevent the loader from adding the binary to your 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 loader 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 loaded is deferred. To cause the loader to add the binaries so the weak reference will work, you must have a *strong reference* (a normal reference) 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 WEAK directive has two forms:

```
!DIR$ WEAK procedure_name [, procedure_name] ...

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

The first form allows you to specify one or more weak objects. This form requires you to implement code that senses that the *procedure_name* procedure is loaded before calling it. The second form allows you to point a weak reference (*procedure_name*) to a stub procedure that exists in your code. This allows you to call 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*.

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

# Cray Streaming Directives (CSDs) (X1 only) [6]

The Cray Streaming Directives (CSDs) are nonadvisory directives that allow you to more closely control multistreaming for key loops. Nonadvisory means that the compiler must honor these directives. The intention of these directives is not to create an additional parallel programming style or demand large effort in code development. They are meant to assist the compiler in multistreaming your program. On its own, the compiler should perform multistreaming correctly in most cases. However, if multistreaming for key loops is not occurring as you desire, then use CSDs to override the compiler.

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

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

- CSD parallel regions. (Section 6.1, page 144)
- `PARALLEL` and `END PARALLEL`—Starts and ends the CSD parallel region. (Section 6.2, page 144)
- `DO` and `END DO`—Multistreams a `DO` loop. (Section 6.3, page 146)
- `PARALLEL DO` and `END PARALLEL DO`—Combine the CSD `parallel` and `do` directives into one directive pair. (Section 6.4, page 149)
- `SYNC`—Synchronizes all SSPs within an MSP. (Section 6.5, page 150)
- `CRITICAL` and `END CRITICAL`—Defines a critical section of code. (Section 6.6, page 150)
- `ORDERED` and `END ORDERED`—Specifies SSPs execute in order. (Section 6.7, page 151)
- `NOCSD`—Suppresses recognition of CSDs. (Section 6.8, page 152)

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

- Nested CSDs within Cray programming models (Section 6.9, page 153)
- CSD placement (Section 6.10, page 153)
- Protection of shared data (Section 6.11, page 154)
- Dynamic memory allocation for CSD parallel regions (Section 6.12, page 155)
- Compiler options affecting CSDs (Section 6.13, page 155)

**Note:** Sometimes the length of a CSD statement can be longer than the maximum allowable line length. To continue the statement, you can use an ampersand character as shown in this example:

```
!csd$ parallel do private (ii,jj,kk,
!csd$& ll,mm,nn)
```

## 6.1 CSD Parallel Regions

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

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

The CSD parallel region can contain loops and nonloop constructs, but only loops are multistreamed. Parallel execution of nonloop constructs, such as initializing variables for the targeted loop, are performed redundantly on all SSPs. Procedures called from the region will be multistreamed, but you must guarantee that the procedure does not cause any *side effects*. Parallel execution of the procedure is independent and redundant on all SSPs, except for code blocks containing stand-alone CSDs. See Section 6.10, page 153.

## 6.2 Start and End Multistreaming: PARALLEL and END PARALLEL

The PARALLEL and END PARALLEL directives define the CSD parallel region, tell the compiler to multistream the region, and optionally specify private data objects. All other CSDs must be used within the region. You cannot place the PARALLEL or END PARALLEL directive in the middle of a construct.

This is the form of the parallel directives:

```
!CSD$ PARALLEL [PRIVATE(list)] [ORDERED]
    structured-block
!CSD$ END PARALLEL
```

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

By default the variables used only for loop indexing, implied-do indices, and FOR ALL indices are assumed to be private. Other variables, unless specified in the PRIVATE clause, are assumed to be shared.

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

The ORDERED clause is needed if there are within the CSD parallel region, but not within CSD DO loops, any calls to procedures containing stand-alone CSD ORDERED directives. The clause is not needed if, within the CSD parallel region, only CSD DO loops contain calls to functions with stand-alone CSD ORDERED directives. If the clause is used and there are no called procedures containing a CSD ORDERED directive, the results produced by the code will be correct, but performance of that code will be slightly degraded. If the ORDERED clause is missing and there is a called procedure containing a CSD ORDERED directive, your results will be incorrect.

The following example shows when the ORDERED clause is needed:

```
!CSD$ PARALLEL ORDERED
    call par_sub  ! par_sub contains a stand-alone ORDERED directive.

!CSD DO
   ...             !No calls to procedures containing stand-alone ORDERED directives
!CSD END DO
!CSD$ END PARALLEL
```

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

**Note:** At the start of the PARALLEL directive, all SSPs are enabled; when the END PARALLEL directive is encountered, all SSPs are disabled.

This example shows how to use the PARALLEL directive:

```
!CSD$ PARALLEL PRIVATE(jx)
x = 2 * PI   !This line is computed on all SSPs
do I = 1,NN
   jx = y(i) * z(i)**x  !jx is private to each SSP
   ...
end do
!CSD$ END PARALLEL
```

## 6.3 Do Loops: DO and END DO

The compiler distributes among the SSPs the iterations of DO loops encapsulated by the CSD DO and END DO directives. Iterations of DO loops not contained by the CSD DO directives are not distributed among the SSPs, but are all executed redundantly by all SSPs.

See Section 6.10, page 153 for placement restrictions of the CSD DO directive.

This is the form of the CSD DO directive:

```
!CSD$ DO [ORDERED] [SCHEDULE(STATIC [, chunk_size])] [ORDERED]
   Do loop block
[!CSD$ END DO [NOWAIT]]
```

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

The iteration distribution is calculated by you or the compiler. You or the compiler will divide the number of iterations into groups or chunks. The compiler will then statically assign the chunks to the 4 SSPs in a round-robin fashion in iteration order. An SSP could have one or more chunks. The number of iterations per chunk is called the chunk size, which is specified by the *chunk_size* argument. The *chunk_size* argument specifies the maximum number of iterations a chunk can have.

You can use these tips to calculate the chunk size:

- Balance the parallel work load across all 4 SSPs (the number of SSPs in an MSP) by dividing the number of iterations by 4. If you have a remainder, add one to the chunk size. Using 4 chunks gives you the best performance, because multiple chunks per SSP increases the overhead caused by the CSD `DO` directive. That is, the fewer number of chunks per SSP (minimum 1), the better the performance.
- The workload distribution among the SSPs will be imbalanced if the chunk size is greater than one fourth of the total number of iterations.
- If the chunk size is greater than the total number of iterations, the first SSP (SSP0) will do all the work.

The compiler calculates the iteration distribution (*`chunk_size`*) if the `SCHEDULE` clause or *`chunk_size`* argument is not specified. The value used is dependent on the conditions shown in Table 9.

Table 9. Compiler-calculated Chunk Size

| Calculated chunk size | Condition |
|---|---|
| 1 | When a CSD `SYNC`, `CRITICAL`, or `ORDERED` directive or a procedural call appears in the loop. |
| Iterations / 4 | The number of iterations are divided as evenly as possible into 4 chunks if these CSDs are not present in the CSD parallel region: `SYNC`, `CRITICAL`, or `ORDERED` directive or a procedural call. This maximum chunk size is 64. |

The `ORDERED` clause is needed if the `DO` loop encapsulated by the CSD `DO` directive calls any procedure containing a stand-alone CSD `ORDERED` directive. If the clause is used and there are no called procedures containing a stand-alone CSD `ORDERED` directive, the results produced by the code encapsulated by the directive will be correct, but performance of that code will be slightly degraded. If the `ORDERED` clause is missing and there is a called procedure containing a stand-alone CSD `ORDERED` directive, the results produced by the code encapsulated by the directive will be incorrect.

The following example shows when the ORDERED clause is needed:

```
!CSD$ PARALLEL
!CSD$ DO ORDERED
   do i = 1, n
      call do_sub(i) !do_sub contains ORDERED directive
   end do
!CSD$ END DO
!CSD$ END PARALLEL
```

The end of the DO loop or the presence of the optional CSD END DO directive marks the end of the streamed CSD DO region. An implicit barrier synchronization occurs at the end of the DO region, unless the NOWAIT clause is also specified. The implicit barrier protects a SSP from prematurely accessing shared data. The NOWAIT clause assumes that you are guaranteeing that consumption-before-production cannot occur.

The following examples illustrate compiler and user-calculated chunk sizes. The compiler calculates the chunk size as 1 for this example, because of the subprogram call (consequently, the first SSP performs iterations 1, 5, 9, ....; the second SSP performs 2, 6, 10, ...; etc.):

```
!CSD$ DO
   DO I = 1, NUM_SAMPLES
      CALL PROCESS_SAMPLE(SAMPLE(I))
   END DO
!CSD$ END DO
```

For this example, because there are no SYNC, CRITICAL, or ORDERED directives or subprogram calls, the compiler calculates the chunk size as MIN(64, (ARRAY_SIZE + 3) / 4):

```
!CSD$ DO
   DO I = 1, ARRAY_SIZE
      PRODUCT(I) = OPERAND1(I) * OPERAND2(I)
   END DO
!CSD$ END DO
```

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

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

```
!CSD$ DO SCHEDULE(STATIC, 128)
   DO I = 1, ARRAY_SIZE
      PRODUCT(I) = OPERAND1(I) * OPERAND2(I)
   END DO
!CSD$ END DO
```

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

## 6.4 Parallel Do Loops: PARALLEL DO and END PARALLEL DO

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

```
!CSD$ PARALLEL [PRIVATE(list)]
!CSD$ DO [SCHEDULE(STATIC [, chunk])] [ORDERED]
   Do_loop_block
!CSD$ END DO
!CSD$ END PARALLEL
```

The differences between the PARALLEL DO and its counterparts include the lack of the NOWAIT clause, because it is not needed.

This is the form of the PARALLEL DO directive:

```
!CSD$ PARALLEL DO [PRIVATE(list)] [SCHEDULE(STATIC
[, chunk_size])]
   Do loop block
!CSD$ END PARALLEL DO
```

For a description of the syntax of the PARALLEL DO directive, refer to the PARALLEL and DO directives at Section 6.2, page 144 and Section 6.3, page 146.

## 6.5 Synchronize SSPs: SYNC

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

The SYNC directive can appear anywhere within the CSD parallel region, even within the CSD DO and PARALLEL DO directives. If the SYNC directive appears within a CSD parallel region but outside of an enclosed CSD DO directive, then it performs an MSYNC on all four SSPs.

This example shows how to use the SYNC directive:

```
!CSD$ PARALLEL DO PRIVATE(J)
   DO I = 1, 4
      DO J = 1, 100000
         X(J, I) = ...          ! Produce X
      END DO

      . . .
      !CSD$ SYNC

      DO J + 1, 100000
         ... = X(J, 5-I) * ... ! Consume X
      END DO
   END DO NOWAIT
!CSD$ END PARALLEL
```

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

## 6.6 Specify Critical Regions: CRITICAL and END CRITICAL

The CRITICAL and END CRITICAL directives specify a critical region where only one SSP at a time will execute the enclosed region.

This is the form of the CRITICAL directive:

```
!CSD$ CRITICAL
   Block of code
!CSD$ END CRITICAL
```

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

```
SUM = 0   !Shared variable

!CSD$ PARALLEL PRIVATE(PRIVATE_SUM)
   PRIVATE_SUM = 0

   !CSD$ DO
      DO I = 1, A_SIZE
         PRIVATE_SUM = PRIVATE_SUM + A(I)
      END DO
   !CSD$ END DO NOWAIT

   !CSD$ CRITICAL
      SUM = SUM + PRIVATE_SUM
   !CSD$ END CRITICAL
!CSD$ END PARALLEL
```

## 6.7 Define Order of SSP Execution: ORDERED and END ORDERED

The CSD ORDERED and END ORDERED directives allow you to multistream loops with particular dependencies by ensuring the execution order of the SSPs and that only one SSP at a time executes the code. That is, first SSP0 completes execution of the block of code surrounded by the ordered directive; next SSP1 completes execution of that block of code etc.

If a stand-alone CSD ORDERED directive is placed in a procedure that is called from a CSD parallel region, the CSD PARALLEL, PARALLEL DO, or DO directives that most closely encapsulates the call needs to specify the ORDERED clause to ensure correct results. See the appropriate CSD for more information.

This is the format of the ORDERED directive:

```
!CSD$ ORDERED
   Block of code
!CSD$ END ORDERED
```

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

```
!CSD$ PARALLEL DO SCHEDULE(STATIC, 1)
   DO I = 3, A_SIZE
      !CSD$ ORDERED
         A(I) = A(I-1) + A(I-2)
      !CSD$ END ORDERED

      ...    ! other processing
   END DO
!CSD$ END PARALLEL DO
```

If the execution time for the code indicated by the other processing comment is larger compared to the time to compute the assignment within the ORDERED directives, then the loop will mostly run concurrently on the 4 SSPs, even if the ORDERED directives are used.

## 6.8 Suppress CSDs: [NO]CSD

The NOCSD directive suppresses recognition of CSDs. It takes effect after the appearance of the directive and applies to the rest of the program unit unless it is superseded by a !DIR$ CSD statement. CSDs are also ignored if multistreaming optimization is disabled by the -O stream0 option.

If the !DIR$ CSD statement follows a !DIR$ NOCSD statement within the same program unit, the compiler resumes recognition of CSDs.

These are the formats of the directives:

```
!DIR$ CSD

!DIR$ NOCSD
```

## 6.9 Nested CSDs within Cray Parallel Programming Models

CSDs can be used within all Cray programming models on Cray X1 series systems with the CSDs at the deepest level. These are the nesting levels:

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

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

## 6.10 CSD Placement

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

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

```
!CSD$ PARALLEL
  ...
!CSD$ DO
   Do loop block...
!CSD$ END DO

!CSD$ END PARALLEL
```

The stand-alone CSDs are `SYNC`, `CRITICAL`, and `ORDERED`. If stand-alone CSDs are placed in a procedure and the procedure is not called from a parallel region, the code will execute as if no CSD requests were present.

## 6.11 Protection of Shared Data

Updates to shared data by procedures called from a CSD parallel region must be protected against simultaneous access by SSPs used for the CSD parallel region. Shared data includes statically allocated data objects (such as data defined in a COMMON block or static files), dynamically allocated data objects pointed to by more than one SSP, and subprogram formal arguments where corresponding actual arguments are shared. Protecting your shared data includes using the CRITICAL directive or the DO loop indices.

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

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

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

```
INTEGER A(SIZE1, SIZE2)
INTEGER INTER_SUM(SIZE2)
INTEGER SUM

!CSD$ PARALLEL DO PRIVATE(INTER_SUM)
   DO I = 1, SIZE2
      INTER_SUM(I) = 0

      DO J = 1, SIZE1
         INTER_SUM(I) = INTER_SUM(I) + A(J, I)
      END DO
   END DO
!CSD$ END PARALLEL

SUM = 0

DO I = 1, SIZE2
   SUM = SUM + INTER_SUM(I)
END DO
```

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

## 6.12 Dynamic Memory Allocation for CSD Parallel Regions

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

Calls to the ALLOCATE and DEALLOCATE intrinsic procedures from within CSD parallel regions must be made by only one SSP at a time. In general, this requires the calls be made from CSD critical regions. This requirement may be relaxed in a future release.

Dynamic memory for private data objects specified by the PRIVATE list of the PARALLEL directive must be allocated and deallocated within the CSD parallel region. Dynamic memory cannot be allocated for private objects before entry into the CSD parallel region and then made private.

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

If the shared or private data object does not have the SAVE attribute, its memory will be automatically deallocated at the end of the procedure containing the CSD parallel region. For private objects, this automatic deallocation may cause an error because deallocation occurs outside of the parallel region. Therefore, you must ensure that memory allocated to private objects are deallocated before exiting the CSD parallel region.

## 6.13 Compiler Options Affecting CSDs

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

To disable CSDs, compile with `-O stream0`, `-x all`, or `-x csd` option.

# Source Preprocessing [7]

Source preprocessing can help you port a program from one platform to another by allowing you to specify source text that is platform specific.

For a source file to be preprocessed automatically, it must have an uppercase extension, either `.F` (for a file in fixed source form), or `.F90` 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 Section 7.4, page 166.

## 7.1 General Rules

You can alter the source code through source preprocessing directives. These directives are fully explained in Section 7.2, page 158. 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`).
- You cannot include a source file that contains an `#if` directive 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.
- You cannot write form feed (FF) or vertical tab (VT) characters 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.

## 7.2 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.

### 7.2.1 `#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. For information about the source preprocessing command line options, see Section 7.4, page 166.

### 7.2.2 #define Directive

The #define directive lets you declare a variable and assign a value to the variable. It also allows you to define 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. For more information about the `#undef` directive, see Section 7.2.3, page 161.

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 with the `-F` option and then run:

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

The following program prints `Hello, Hello, world.` when compiled with the `-F` option and then run:

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

### 7.2.3 #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 undefined.

### 7.2.4 # (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.

### 7.2.5 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 you may or may not want 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, you can write the following directives:

```
#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.

### 7.2.5.1 `#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 directive forms 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).

### 7.2.5.2 #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. For more information about the -D option, see Section 7.4, page 166. This directive has the following format:

```
#ifdef identifier
```

The #ifdef directive is equivalent to either of the following two directives:

- #if defined *identifier*
- #if defined(*identifier*)

### 7.2.5.3 #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*)

### 7.2.5.4 #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.

### 7.2.5.5 #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
```

### 7.2.5.6 `#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
```

## 7.3 Predefined Macros

The Cray Fortran compiler source preprocessing supports a number of predefined macros. They are divided into groups as follows:

- Macros that are based on the host machine
- Macros that are based on UNICOS/mp and UNICOS/lc system targets

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 UNICOS/mp and UNICOS/lc systems as targets:

`__crayx1`

Defined as 1 on all Cray X1 series systems.

`__crayx1e`

Defined as 1 on all Cray X1E systems.

`__crayx2`

Defined as 1 on all Cray X2 systems.

`_UNICOSMP`

Defined as 1 on all Cray X1 series systems.

`cray`, `CRAY`, `_CRAY`

(X1 only) These macros are defined for UNICOS/mp systems as targets.

`_CRAYIEEE`

Defined as 1 on all Cray X1 series and X2 systems as targets.

`_MAXVL`

Defined as the hardware vector register length (64 for the Cray X1 and 128 for the Cray X2).

`_ADDR64`

Defined for UNICOS/mp and UNICOS/lc systems as targets. The target system must have 64-bit address registers.

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.

## 7.4 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 Section 3.6, page 26.
- The `-eP` option, which performs source preprocessing on *file*`.f[90]`, *file*`.F[90]`, *file*`.ftn`, or *file*`.FTN` but does not compile. The `-eP` option produces *file*`.i`. For more information about this option, see Section 3.5, page 18.
- The `-eZ` option, which performs source preprocessing and compilation on *file*`.f[90]`, *file*`.F[90]`, *file*`.ftn`, or *file*`.FTN`. The `-eZ` option produces *file*`.i`. For more information about this option, see Section 3.5, page 18.
- The `-F` option, which enables macro expansion throughout the source file. For more information about this option, see Section 3.8, page 26.
- The `-U` *identifier* `[,` *identifier*`]` ... option, which undefines variables used for source preprocessing. For more information about this option, see Section 3.28, page 76.

The `-D` *identifier* `[=`*value*`]`, `-F`, and `-U` *identifier* `[,` *identifier*`] ...` options are ignored unless one of the following is true:

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

# OpenMP Fortran API [8]

OpenMP Fortran is a parallel programming model that is portable across shared memory architectures from Cray and other vendors. The Cray Fortran compiler supports the *OpenMP Fortran Application Program Interface, version 2.5* standard. All OpenMP library procedures and directives, except for limitations in a few directive clauses, are supported.

All OpenMP directives and library procedures are documented by the OpenMP Fortran specification which is accessible at `http://www.openmp.org/drupal/node/view/8`.

For information about Cray specific OpenMP Fortran information like implementation differences, see the following sections:

- Cray Implementation Differences (Section 8.1, page 167)
- `OMP_THREAD_STACK_SIZE` Environment Variable (Section 8.2, page 169)
- OpenMP Optimizations (Section 8.3, page 170)
- Compiler Options that Affect OpenMP (Section 8.4, page 172)
- OpenMP Program Execution (Section 8.5, page 172)

## 8.1 Cray Implementation Differences

The OpenMP Fortran Application Program Interface specification defines areas of implementation that have vendor-specific behaviors. This section documents those areas and other areas not defined by the specification.

These OpenMP items have Cray specific behaviors in areas defined as implementation-dependent by the OpenMP specification:

- Implementation-dependent areas of parallel region constructs:
  - 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 run-time system can supply, the program will terminate.
  - 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.

- Implementation-dependent areas of DO and PARALLEL DO directives:
  - SCHEDULE(GUIDED,*chunk*)—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.
  - SCHEDULE(RUNTIME)—The schedule type and chunk size can be chosen at run time by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the schedule type and chunk size default to GUIDED and 1, respectively.
  - Default schedule—In the absence of the SCHEDULE clause, the default schedule is STATIC and the default chunk size is roughly the number of iterations divided by the number of threads.
- Implementation-dependent area of the THREADPRIVATE directive—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.
- Implementation-dependent area of the PRIVATE clause—If a variable is declared as PRIVATE, and 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.
- Implementation-dependent areas of the ATOMIC directive—The ATOMIC directive is replaced with a critical section that encloses the statement.
- Implementation-dependent areas of OpenMP library functions:
  - OMP_GET_NESTED—This procedure always returns .FALSE. because nested parallel regions are always serialized.
  - OMP_GET_NUM_THREAD—If the number of threads has not been explicitly set by the user, the default is the *depth* value defined through the aprun -d *depth* option. If this option is not set, the aprun command defaults depth to 1, which sets the number of threads to one, which value OMP_GET_NUM_THREAD returns.
  - OMP_SET_NUM_THREADS—If dynamic adjustment of the number of threads is disabled, the number_of_threads argument sets the number of threads for all subsequent parallel regions until this procedure is called again with a different value.
  - OMP_SET_DYNAMIC—The default for dynamic thread adjustment is on.

  - OMP_SET_NESTED—Calls to this function are ignored since nested parallel regions are always serialized.

- Implementation-dependent areas of OpenMP environment variables:

  - OMP_DYNAMIC—The default value is .TRUE.

  - OMP_SET_NESTED—This environment variable is ignored because nested parallel regions are always serialized and executed by a team of one thread.

  - OMP_NUM_THREADS—The default value is the value of *depth* as defined by the aprun -d *depth* option or 1 if the option is not specified.

    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; otherwise, it uses up to 16 threads on the Cray X1 series and X2 systems.

  - OMP_SCHEDULE—The default values for this environment variable are GUIDED for schedule and 1 for chunk size.

- Implementation-dependent areas of OpenMP library routines that have generic interfaces—If an OMP run-time library routine interface is defined to be generic by an implementation, use of arguments of kind other than those specified by the OMP_*_KIND constants is undefined.

These OpenMP features have Cray specific behaviors in areas not defined as implementation-dependent by the OpenMP specification:

- 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.

- The omp_get_wtime and omp_get_wtick procedures return REAL(KIND=8) values instead of DOUBLE PRECISION values.

## 8.2 OMP_THREAD_STACK_SIZE Environment Variable

OMP_THREAD_STACK_SIZE is a Cray specific OpenMP environment variable that affects programs at run time. It changes the size of the thread stack from the default size of 16 MB to the specified size. The size of the thread stack should be increased when private variables may utilize more than 16 MB of memory.

(X1 only) The requested thread stack space is allocated from the local heap when the threads are created. The amount of space used by each thread for thread stacks depend on whether you are using MSP or SSP mode. In MSP mode, the memory used is 5 times the specified thread stack size because each SSP is assigned one thread stack and one thread stack is used as the MSP common stack. For SSP mode, the memory used is one times the specified thread stack size.

(X1 only) Since memory is allocated from the local heap, you may want to consider how increasing the size of the thread stacks will affect available space in the local heap. To adjust the size of the local heap, see the `X1_HEAP_SIZE` and `X1_LOCAL_HEAP_SIZE` environment variables in the `memory`(7) man page.

(X2 only) The heaps on X2 do not have to be sized statically as on X1; their sizes are adjusted as needed.

This is the format for the `OMP_THREAD_STACK_SIZE` environment variable:

`OMP_THREAD_STACK_SIZE` *n*

where *n* is a hex, octal or decimal integer specifying the amount of memory, in bytes , to allocate for a thread's stack.

## 8.3 OpenMP 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. For more information about optimization, see *Optimizing Applications on Cray X1 Series Systems*(for the X1 series), and *Optimizing Applications on Cray X2 Systems* (for the X2 systems).

Example 1: Loop interchange. 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, you can parallelize the J loop or the I loop. You cannot parallelize the K loop 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, you can reorder the loops 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.

Example 2: Conditional parallelism. 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
```

## 8.4 Compiler Options that Affect OpenMP

These Cray Fortran compiler options enable or disable the OpenMP directives or determine the type of processing elements each thread runs on:

- Enable OpenMP directive recognition: `-O task1` (default)
- Disable OpenMP directive recognition: `-O 0`, `-O task0`, or `-x omp`
- (X1 only) Compile the code to allow the threads to run on MSPs or SSPs: `-O msp` (default), `-O ssp`

## 8.5 OpenMP Program Execution

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. This example shows how to reserve the physical processors:

```
aprun -d depth ompProgram
```

(X1 only) If the program is compiled for MSP mode, *depth* must be less than or equal to 4; for SSP mode less than or equal to 16. If *depth* is not specified, the `aprun` command defaults *depth* to 1.

(X2 only) If the program is compiled for X2 systems, *depth* must be less than or equal to 4, the size of an X2 SMP node.

If the `OMP_NUM_THREADS` environment variable is not set, the program behaves as if `OMP_NUM_THREADS` is set to the same value as *depth*.

The `aprun` options `-n` *processes* and `-N` *processes_per_node* are compatible with OpenMP but do not directly affect the execution of OpenMP programs.

# Cray Fortran Defined Externals [9]

This chapter describes global variables used by the Cray Fortran compiler targeting UNICOS/mp and UNICOS/lc systems.

## 9.1 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 loader option or through an environment variable.

The default error checking provides only limited error checking.

Use the loader 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 to run the executable. The loader 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: Table 17, page 202, Table 18, page 203, Table 19, page 203, and Table 20, page 203.

The environment variable `FORMAT_TYPE_CHECKING` is evaluated during execution. The environment variable will override a table chosen through the loader option. The environment variable provides an intermediate type of checking that is not provided by the loader option. The environment variable `FORMAT_TYPE_CHECKING` is described in section 4.1.3.

To select the least amount of checking, use one or more of the following `ftn` command line options.

- On UNICOS/mp systems with formatted READ, use:

  **`ftn  -W1,-equiv,_RCHK=_RNOCHK ...`**

- On UNICOS/mp systems with formatted WRITE, use:

  **`ftn -W1,-equiv,_WCHK=_WNOCHK *.f`**

- On UNICOS/mp systems with both formatted READ and WRITE, use:

  **`ftn -W1,-equiv,_WCHK=_WNOCHK -W1,-equiv,_RCHK=_RNOCHK *.f`**

- On UNICOS/lc systems with formatted READ, use:

  **`ftn -W1,--defsym,_RCHK=_RNOCHK *.f`**`(note the double dashes that precede defsym)`

- On UNICOS/lc systems with formatted WRITE, use:

  **`ftn -W1,--defsym,_WCHK=_WNOCHK *.f`**

- On UNICOS/lc 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 UNICOS/mp systems with formatted READ, use:

  ```
  ftn  -W1,-equiv,_RCHK=_RCHK77 *.f

  ftn  -W1,-equiv,_RCHK=_RCHK90 *.f
  ```

- On UNICOS/mp systems with formatted WRITE, use:

  ```
  ftn -W1,-equiv,_WCHK=_WCHK77 *.f

  ftn -W1,-equiv,_WCHK=_WCHK90 *.f
  ```

- On UNICOS/mp systems with both formatted READ and WRITE, use:

  ```
  ftn -W1,-equiv,_WCHK=_WCHK77 -W1,-equiv,_RCHK=_RCHK77 *.f

  ftn -W1,-equiv,_WCHK=_WCHK90 -W1,-equiv,_RCHK=_RCHK90 *.f
  ```

- On UNICOS/lc systems with formatted READ, use:

  ```
  ftn -W1,--defsym,_RCHK=_RCHK77 *.f

  ftn -W1,--defsym,_RCHK=_RCHK90 *.f
  ```

- On UNICOS/lc systems with formatted WRITE, use:

  ```
  ftn -W1,--defsym,_WCHK=_WCHK77 *.f

  ftn -W1,--defsym,_WCHK=_WCHK90 *.f
  ```

- On UNICOS/lc 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
  ```

# Part II: Cray Fortran and Fortran 2003 Differences

The Cray Fortran compiler is based on the Fortran 2003 standard. Part II documents only the differences between the Cray Fortran implementation and the Fortran standard. It is divided into the following chapters:

# Cray Fortran Language Extensions [10]

The Cray Fortran compiler supports several features beyond those specified by the standard. These features are referred to as extensions. The extensions described in this chapter include extensions widely implemented in other compilers and facilities designed to provide access to hardware features of the Cray X1 series and X2 systems. Also included are extensions that might become features in a future Fortran standard. The implementation of such features in the compiler might be modified as needed in the future to conform to the new standard. For information about obsolete features, see Obsolete Features (Chapter 11, page 229).

The listings provided by the compiler will identify language extensions when the `-e n` command line option is specified.

## 10.1 Characters, Lexical Tokens, and Source Form

### 10.1.1 Low-level Syntax

#### 10.1.1.1 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* | **is** | *currency_symbol* |
| | **or** | *at_sign* |
| *currency_symbol* | **is** | $ |
| *at_sign* | **is** | @ |

A name must begin with a letter and can consist of letters, digits, and underscores. The Cray Fortran compiler allows you to use the at sign (@) and dollar sign ($) in a name, but they cannot be the first character of a name.

Cray does not recommend using @ and $ in user names because they could cause conflicts with the names of internal variables or library routines.

#### 10.1.1.2 Switching Source Forms

The Cray Fortran compiler allows you to switch between fixed and free source forms within a file or include file by using the FIXED and FREE compiler directives.

#### 10.1.1.3 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.

#### 10.1.1.4 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.

## 10.2 Types

### 10.2.1 The Concept of Type

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.

#### 10.2.1.1 Alternate Form of LOGICAL Constants

The Cray Fortran compiler accepts `.T.` and `.F.` as alternate forms of `.true.` and `.false.`, respectively.

#### 10.2.1.2 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, 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*.

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.

You can use pointers 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. For more information about Cray character pointers, see Section 10.2.1.3, page 186.
- 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.

  **Note:** 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(3i) 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 X1 series and X2 systems, this prints the following:

```
Byte-addressable machine
```

**Note:** 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. You are 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(3i) 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.

#### 10.2.1.3 Cray Character Pointer Type

If a pointee is declared as 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.

#### 10.2.1.4 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 Section 10.4.1, page 191.

#### 10.2.1.5 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 you assign to the group. This name is treated as a type alias name.

#### 10.2.1.6 TYPEALIAS Statement

A TYPEALIAS statement allows you to define another name 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
```

You can use a type alias or the data type it aliases 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.

## 10.3 Data Object Declarations and Specifications

The Cray Fortran compiler accepts the following extensions to declarations.

### 10.3.1 Attribute Specification Statements

#### 10.3.1.1 BOZ Constants 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 |

### 10.3.1.2 Attribute Respecification

The Cray Fortran compiler permits an attribute to appear more than once in a given type declaration.

### 10.3.1.3 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.

### 10.3.2 IMPLICIT Statement

#### 10.3.2.1 `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.

### 10.3.3 Storage Association of Data Objects

#### 10.3.3.1 `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 you do not perform equivalencing in this manner, however, because alignment and padding differs across platforms, thus rendering your code less portable.

#### 10.3.3.2 `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.

## 10.4 Expressions and Assignment

### 10.4.1 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.

#### 10.4.1.1 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* |

#### 10.4.1.2 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 you redefine a masking operator, your definition overrides the intrinsic masking operator definition. See Table 11, page 194, for a list of the operators.

### 10.4.1.3 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 10. 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.) | | |
| // | 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 |

| Intrinsic operator | Type of $x_1$ | Type of $x_2$ | Type of result |
|---|---|---|---|
| | | 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.

#### 10.4.1.4 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 11. 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 12, page 195 shows which data types can be used together in bitwise logical operations.

Table 12. Data Types in Bitwise Logical Operations

| $x_1$ $x_2$[1] | **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[2] |
| Real | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Not valid | Not valid[2] |
| Boolean | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Not valid | Not valid[2] |
| Pointer | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Masking operation, Boolean result. | Not valid | Not valid[2] |
| Logical | Not valid[2] | Not valid[2] | Not valid[2] | Not valid[2] | Logical operation logical result | Not valid[2] |
| Character | Not valid[2] | Not valid[2] | Not valid[2] | Not valid[2] | Not valid | Not valid[2] |

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.

1 $x_1$ and $x_2$ represent operands for a logical or bitwise expression, using operators **.NOT.**, **.AND.**, **.OR.**, **.XOR.**, **.NEQV.**, and **.EQV.**.

2 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.

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
```

### 10.4.2 Assignment

#### 10.4.2.1 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.

## 10.5 Execution Control

### 10.5.1 STOP Code Extension

The STOP statement terminates the program whenever and wherever it is executed. The STOP statement is defined as follows:

| | | |
|---|---|---|
| *stop-stmt* | **is** | STOP [*stop_code*] |
| *stop-code* | **is** | *scalar_char_constant* |
| | **or** | *digit ...* |

The character constant or list of digits identifying the STOP statement is optional and is called a *stop-code*. When the *stop-code* is a string of digits, leading zeros are not significant; 10 and 010 are the same stop-code. The Cray Fortran compiler accepts 1 to 80 digits; the standard accepts up to 5 digits.

The stop code is accessible following program termination. The Cray Fortran compiler sends it to the standard error file (stderr). The following are examples of STOP statements:

```
STOP
STOP 'Error #823'
STOP 20
```

## 10.6 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.

### 10.6.1 File Connection

#### 10.6.1.1 OPEN Statement

The OPEN statement specifies the connection properties between the file and the unit, using keyword specifiers, which are described in this section. Table 13 indicates the Cray Fortran compiler extension in an OPEN statement.

Table 13. Values for Keyword Specifier Variables in an OPEN Statement

| Specifier | Possible values | Default value |
|---|---|---|
| FORM= | SYSTEM | Unformatted with no record marks |

The FORM= specifier has the following format:

FORM= *scalar-char-expr*

A file opened with SYSTEM is unformatted and has no record marks.

## 10.7 Error, End-of-record, and End-of-file Conditions

### 10.7.1 End-of-file Condition and the END-specifier

#### 10.7.1.1 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.

## 10.8 Input/Output Editing

### 10.8.1 Data Edit Descriptors

#### 10.8.1.1 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.

**Note:** UNICOS/mp and UNICOS/lc systems support 1- and 2-byte data types when the -eh compiler option is enabled. Cray discourages the use of this option because it can severely degrade performance. For more information about the -eh option, see Section 3.5, page 18.

#### 10.8.1.2 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 Section 4.1.9, page 85.

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 14. 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 |
| 16-byte (128-bit) IEEE | 46 | 36 | 4 |

#### 10.8.1.3 Logical Editing

The Cray Fortran compiler allows *w* to be zero or omitted on the L or G edit descriptors.

#### 10.8.1.4 Character Editing

The Cray Fortran compiler allows *w* to be zero or omitted on the G edit descriptor.

### 10.8.2 Control Edit Descriptors

#### 10.8.2.1 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)
```

### 10.8.3 List-directed Formatting

#### 10.8.3.1 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.

### 10.8.4 Namelist Formatting

#### 10.8.4.1 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'`

### 10.8.5 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.

Table 15 and Table 16, list the control and data edit descriptor extensions supported by the Cray Fortran compiler and provide a brief description of each.

Table 15. Summary of Control Edit Descriptors

| Descriptor | Description |
|---|---|
| $ or \ | Suppress carriage control |

Table 16. Summary of Data Edit Descriptors

| Descriptor | Description |
|---|---|
| Q | Return number of characters left in record |

For more information about the Q edit descriptor, see Section 10.8.2.1, page 199.

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.

Table 17. 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 |

Table 18, page 203 shows the restrictions for the various data types that are allowed when you set the FORMAT_TYPE_CHECKING environment variable to RELAXED. Not all data edit descriptors support all data sizes; for example, you cannot read/write a 16–byte real variable with an I edit descriptor.

Table 18. 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 |
| Character | NA | NA | NA | NA | NA | NA | I,O | NA | NA | NA | NA | NA | NA | I,O |

Table 19 shows the restrictions for the various data types that are allowed when you set the FORMAT_TYPE_CHECKING environment variable to STRICT77.

Table 19. 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 |

Table 20 shows the restrictions for the various data types that are allowed when you set the FORMAT_TYPE CHECKING environment variable to STRICT90 or STRICT95.

Table 20. STRICT90 and 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 |

## 10.9 Program Units

### 10.9.1 Main Program

#### 10.9.1.1 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*

### 10.9.2 Block Data Program Units

#### 10.9.2.1 Block Data Program Unit Extension

The Cray Fortran compiler permits named common blocks to appear in more than one block data program unit.

## 10.10 Procedures

### 10.10.1 Procedure Interface

#### 10.10.1.1 Interface Duplication

The Cray Fortran compiler allows you to specify an interface body for the program unit being compiled if the interface body matches the program unit definition.

### 10.10.2 Procedure Definition

#### 10.10.2.1 Recursive Function Extension

The Cray Fortran compiler allows direct recursion for functions that do not specify a `RESULT` clause on the `FUNCTION` statement.

#### 10.10.2.2 Empty `CONTAINS` Sections

The Cray Fortran compiler allows a `CONTAINS` statement with no internal or module procedure following. This is proposed for the 2008 Fortran standard.

## 10.11 Intrinsic Procedures and Modules

### 10.11.1 Standard Generic Intrinsic Procedures

#### 10.11.1.1 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 that are not obsolete are summarized in the following list. For more information about a particular procedure, see its man page.

| | |
|---|---|
| `ACOSD` | Arccosine, value in degrees |
| `ADD_CARRY@` | Add vectors with carry |
| `ADD_CARRY_S@` | Add scalars with carry |
| `AMO_AADD` | Atomic memory add |
| `AMO_AFADD` | Atomic memory add, return old |
| `AMO_AAX` | Atomic memory logicals |
| `AMO_AFAX` | Atomic memory logicals, return old |
| `AMO_ACSWAP` | Atomic compare and swap |
| `ASIND` | Arcsine, value in degrees |
| `ATAND` | Arctangent, value in degrees |
| `ATAND2` | Arctangent, value in degrees |
| `COSD` | Cosine, argument in degrees |
| `COT` | Cotangent |
| `DSHIFTL` | Double word left shift |
| `DSHIFTR` | Double word right shift |

| | |
|---|---|
| END_CRITICAL | End of a critical region |
| EXIT | Program termination |
| FREE | Free Cray pointee memory |
| GET_BORROW@ | Get vector borrow bits |
| 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 |
| LEADZ | Number of leading 0 bits |
| LOC | Address of argument |
| LOG2_IMAGES | Logarithm base 2 of number of images |
| M@CLR | Clears BML bit |
| M@LD | Bit matrix load |
| M@LDMX | Combined bit matrix load and multiply |
| M@MOR | Bit matrix inclusive or |
| M@MX | Bit matrix multiply |
| M@UL | Bit matrix unload |
| MALLOC | Allocate Cray pointee memory |
| MASK | Creates a bit mask in a word |
| NUMARG | Number of arguments in a call |

`NUM_IMAGES` Number of executing images

`POPCNT` Number of 1 bits in a word

`POPPAR` XOR reduction of bits in a word

`QPROD` Quad precision product

`REM_IMAGES` Mod (`num_images(), 2**log2_images()`)

`SET_BORROW@`

Set vector borrow bits

`SET_BORROW_S@`

Set scalar borrow bits

`SET_CARRY@` Set vector carry bits

`SET_CARRY_S@`

Set scalar carry bits

`SHIFTA` Arithmetic right shift

`SHIFTL` Left shift, zero fill

`SHIFTR` Right shift, zero fill

`SIND` Sin, argument in degrees

`SIZEOF` Size of argument in bytes

`SSPID@` SSP number within an MSP (`0..3`) (X1 only)

`START_CRITICAL`

Begin critical region

`STREAMING@` Indicates if streaming is allowed (X1 only)

`SUB_BORROW@`

Subtract vector with borrow

`SUB_BORROW_S@`

Subtract scalar with borrow

`SYNC_ALL` Synchronize all images

| | |
|---|---|
| SYNC_FILE | Synchronize file access among images |
| SYNC_IMAGES | Synchronize indicated images |
| SYNC_MEMORY | Memory barrier (same as GSYNC) |
| SYNC_TEAM | Synchronize a team of images |
| TAND | Tangent, argument in degrees |
| THIS_IMAGE | Image number of executing image |

All Cray Fortran intrinsic procedures are described in man pages that can be accessed online through the man(1) command.

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 intro_intrin(3i).

## 10.12 Exceptions and IEEE Arithmetic

### 10.12.1 The Exceptions

#### 10.12.1.1 IEEE Intrinsic Module Extensions

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 21. Cray Fortran IEEE Intrinsic Module Extensions

| Name | Effect of CALL IEEE_SET_STATUS (Name) |
|---|---|
| ieee_cri_silent_mode | • Clears all currently set exception flags<br>• Disables halting for all exceptions<br>• Disables setting of all exception flags<br>• Sets rounding mode to round_to_nearest |
| ieee_cri_nostop_mode | • Clears all currently set exception flags<br>• Disables halting for all exceptions<br>• Enables setting of all exception flags<br>• Sets rounding mode to round_to_nearest |
| ieee_cri_default_mode | • Clears all currently set exception flags<br>• Enables halting for overflow, divide_by_zero, and invalid<br>• Disables halting for underflow and inexact<br>• Enables setting of all exception flags<br>• Sets rounding mode to round_to_nearest |

## 10.13 Interoperability With C

### 10.13.1 Interoperability Between Fortran and C Entities

#### 10.13.1.1 `BIND(C)` Syntax

The *proc-language-binding-spec* specification allows Fortran programs to interoperate with C objects. The optional commas in `SUBROUTINE name(), BIND(C)` and `FUNCTION name(), BIND(C)` are Cray extensions to the Fortran standard.

## 10.14 Co-arrays

The Cray Fortran compiler implements co-arrays as a mechanism for data exchange in parallel programs.

Data passing has proven itself to be an effective method for programming single-program-multiple-data (SPMD) parallel computation. Its chief advantage over message passing is lower latency for data transfers, which leads to better scalability of parallel applications. *Co-arrays* are a syntactic extension to the Fortran Language that offers a method for programming data passing.

Data passing can also be accomplished by using the shared memory (SHMEM) library routines. Using SHMEM, the program transfers data from an object on one processing element to an object on another via subroutine calls. This technique is often referred to as one-sided communication.

Co-arrays provide an alternative syntax for specifying these transfers. With co-arrays, the concept of a processing element is replaced by the concept of an *image*. When data objects are declared as co-arrays, the corresponding co-arrays on different images can be referenced or defined in a fashin similar to the way in which arrays are referenced or defined in Fortran. This is done by adding additional dimensions, or *co-dimensions*, within brackets ([ ]) to an object's declarations and references. These extra dimensions express the image upon which the object resides. Since no subroutine calls are involved in data passing using co-arrays, this technique is referred to as zero-sided communication.

Co-arrays offer the following advantages over SHMEM:

- Co-arrays are syntax-based, so programs that use them can be analyzed and optimized by the compiler. This offers greater opportunity for hiding data transfer latency.
- Co-array syntax can eliminate the need to create and copy data to local temporary arrays.
- Co-arrays express data transfer naturally through the syntax of the language, making the code more readable and maintainable.
- The unique bracket syntax allows you to scan for and to identify communication in a program easily.

Consider the following SHMEM code fragment from a finite differencing algorithm:

```
CALL SHMEM_REAL_GET(T1, U, NROW, LEFT)
CALL SHMEM_REAL_GET(T2, U, NROW, RIGHT)
NU(1:NROW) = NU(1:NROW) + T1(1:NROW) + T2(1:NROW)
```

Co-arrays can be used to express this fragment simply as:

```
NU(1:NROW) = NU(1:NROW) + U(1:NROW)[LEFT] + U(1:NROW)[RIGHT]
```

Notice that the resulting code is more concise, easier to read, and that the copies to local temporary objects `T1` and `T2` are eliminated.

Co-arrays can interoperate with the other message passing and data passing models. This interoperability allows you to introduce co-arrays gradually into codes that presently use the Message Passing Interface (MPI) or SHMEM.

This chapter describes the syntax and semantics of the co-array extension to the Cray Fortran compiler.

The following technical papers may be of use to you when using co-arrays:

- R. W. Numrich and J. Reid, *Co-array Fortran for Parallel Programming*, vol. 17, Number 2 (ACM Fortran Forum, 1998), 1–31

  You can also access the document at this address:

  `ftp://matisa.cc.rl.ac.uk/pub/reports/nrRAL98060.ps.gz`

- R. W. Numrich, J. L. Steidel, B. H. Johnson, B. D. de Dinechin, G. W. Elsesser, G. S. Fischer, and T. A. MacDonald, *Definition of the F−− Extension to Fortran 90*, Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computers, Lectures on Computer Science Series, Number 1366, (Speinger-Verlag, 1998), 282–306

### 10.14.1 Execution Model and Images

Programs with Cray Fortran co-arrays use the *single-program-multiple-data (SPMD)* execution model. In the SPMD model, the program and all its data are replicated and executed asynchronously. Each replication of the program is an *image*. Each image is executed on a processing element.

Images are numbered consecutively starting with one.

**Note:** (X1 only) Indicating the processing element type an image runs on (multistreaming processor (MSP) or single streaming processor (SSP)), is determined at the command line of the Cray Fortran compiler. See Section 10.15.1, page 225.

The total number of images that are executing can be accessed through the `NUM_IMAGES` intrinsic function. An image can access its own image number through the `THIS_IMAGE` intrinsic function. Images can synchronize through the `SYNC_ALL` intrinsic subroutine.

### 10.14.2 Specifying Co-arrays

A *co-array* is a data object that is identically allocated on each image and, more significantly, can be directly referenced by any other image syntactically.

A co-array specification consists of the local object specification and the co-dimensions specification. The *local object* is the data object to be replicated on each image. The *co-dimensions* are the dimensions of the co-array, which are specified within brackets (`[ ]`) and appended to the specification for the local object.

Example 1. The following statements show co-array declarations:

```
REAL, DIMENSION(20)[8,*] :: A, C
REAL :: B(20)[8,*], D[*], E[0:*]
INTEGER :: IB(10)[*]
```

> **Note:** Generally, a co-dimension specification in brackets takes the same form as a dimension specification in parentheses. The exception is that for co-dimensions, the upper bound of the right-most co-dimension must be an asterisk (*). This is because co-array objects are replicated on all images, so co-size is always equal to NUM_IMAGES.

Elements of co-arrays on other images can be referenced by appending square brackets to the end of a reference to the local object. As the following shows, the brackets contain subscripts, one for each co-dimension:

```
A(5)[7,3] = IB(5)[3]
D[11] = E
A(:)[2,3] = C(1)[1,1]
```

The co-dimension specification of a co-array creates a mapping of subscripts to images. This mapping is identical to the mapping that parenthesized array dimensions create between subscripts and elements of an array. For example, the following table lists the image number for some references of the objects declared in Example 1:

| Reference | Image |
|---|---|
| IB(5)[3] | 3 |
| A(5)[7,3] | 31 |
| D[11] | 11 |
| E[11] | 12 |

The terms *local rank*, *local size*, and *local shape* refer to the rank, size and shape of the local object of a co-array. The terms *co-rank*, *co-size*, and *co-shape* refer to those properties implied by the co-dimensions of a co-array. For example, for co-array A declared in the preceding list, its local rank is 1; its local size is 20; its co-rank is 2; and its co-size is equal to NUM_IMAGES. The co-rank of a co-array cannot exceed 7.

The local object of a co-array can be of a derived type, but a co-array cannot be a component within a derived type. For example:

```
TYPE DTYPE1
   REAL :: X
   REAL :: Y
END TYPE DTYPE1

TYPE(DTYPE1) :: DT(100)[*] ! PERMITTED: CO-ARRAY OF DERIVED TYPE

TYPE DTYPE2
   REAL :: X
   REAL :: Y[*]           ! NOT PERMITTED:
                            ! CO-ARRAY IN DERIVED TYPE
END TYPE DTYPE2
```

Most objects can be the local object of a co-array, but the following list indicates restrictions on co-array specifications:

- Co-arrays with assumed-size local size are not supported. For example:

  ```
  REAL :: Y(*)[*]     ! NOT SUPPORTED: LOCAL OBJECT ASSUMED SIZE
  ```

- Co-arrays with deferred-shape local shape or co-shape are supported, but the co-array must be allocatable. Co-array pointers are not supported. For example:

  ```
  REAL, ALLOCATABLE :: WA(:)[:]   ! SUPPORTED: ALLOCATABLE
  REAL, POINTER     :: WP(:)[:]   ! NOT SUPPORTED: POINTER
  ```

- Co-arrays with assumed-shape local shape or co-shape are not supported. For example:

  ```
  SUBROUTINE S1( Z1, Z2 )
  REAL :: Z1(:)[*]    ! NOT SUPPORTED: ASSUMED-SHAPE LOCAL SHAPE
  REAL :: Z2(:)[:]    ! NOT SUPPORTED: ASSUMED-SHAPE CO-SHAPE
  ```

- Automatic co-arrays are not supported. For example:

  ```
  SUBROUTINE S2( A, N )
  REAL :: A(N)[*]            ! SUPPORTED: CO-ARRAY ACTUAL ARGUMENT
  REAL :: W(N)[*]            ! NOT SUPPORTED: AUTOMATIC LOCAL OBJECT
  ```

### 10.14.3 Referencing Co-arrays

Co-arrays can be referenced two ways: with brackets and without brackets.

When brackets are omitted, the object local to the invoking image is referenced; this is called a *local reference.* For example:

```
REAL, DIMENSION(100)[*] :: A, B, C, D, E

A(I) = B(I) + C(I)              ! LOCAL REFERENCES TO A, B, C
D = E                           ! LOCAL REFERENCES TO D, E
```

When brackets are specified, the object on the image specified by the subscripts within the brackets is referenced. This is called a *bracket reference.* For example:

```
A(I)[IP] = B(I) + C(I)          ! REFERENCE TO A ON IMAGE "IP";
                                ! LOCAL REFERENCES TO B, C

D(:) = E(:)[IP2]                ! REFERENCES TO E ON IMAGE "IP2"
                                ! LOCAL REFERENCES TO D
```

Components of derived type co-arrays are specified by appending the component specification after the brackets. For example:

```
TYPE DTYPE3
  REAL    :: X(100)
  INTEGER :: ICNT
END TYPE DTYPE3

TYPE (DTYPE3) :: DT3[*]

DT3%ICNT = DT3[IP]%ICNT   ! SUPPORTED: BRACKET IN DERIVED TYPE
DT3%X(J) = DT3[IP]%X(J)   ! COMPONENT REFERENCES
```

The co-subscripts of a co-array reference must translate to an image number between 1 and NUM_IMAGES, otherwise the behavior of the reference is undefined.

There is a restriction for co-array references. Specification of subscripts for co-dimensions generally follows the specification of subscripts within parentheses. However, support for triplet subscript notation within brackets is not supported. For example:

```
D(K)[1:N:2] = E(K)[1:N:2]       ! NOT SUPPORTED:
                                ! TRIPLET NOTATION IN []S
```

### 10.14.4 Initializing Co-arrays

Co-arrays can be initialized using the DATA statement, but only the initialization of the local object can be specified. Bracket references are not allowed in a DATA statement. For example:

```
REAL :: AI(100)[*]
DATA AI(3)      /1.0/   ! PERMITTED
DATA AI(3)[11]  /1.0/   ! NOT PERMITTED
```

When the program is executed, the co-array local objects on every image are initialized identically, as specified.

### 10.14.5 Using Co-arrays with Procedure Calls

If a procedure with a co-array dummy argument is called, the called procedure must have an explicit interface, and the actual argument must be a local reference to a co-array. If the actual argument has subscripts, their values should be the same across all images, otherwise the program behavior is undefined. For example:

```
INTERFACE
   SUBROUTINE S3( A, N )
   REAL :: A(N)[*]
END INTERFACE

REAL :: X(100,100), Y(100,100)[*]

CALL S3( X(1,K), 100 )  ! NOT PERMITTED:
                        ! LOCAL ACTUAL, CO-ARRAY DUMMY

CALL S3( Y(1,K), 100 )  ! PERMITTED: CO-ARRAY ACTUAL AND DUMMY;
                        ! UNDEFINED IF "K" NOT SAME VALUE ON
                        ! ALL IMAGES
```

Bracket references cannot appear as actual arguments in subroutine calls or function calls. For example:

```
CALL S3( Y(1,K)[IP], 100 )  ! NOT PERMITTED: BRACKET ACTUAL
```

Co-array bracket references can appear within an actual argument, but only as part of an expression that is passed as the actual argument. Parentheses can be used to turn a bracket reference into an expression. For example:

```
CALL S3( ( Y(1,K)[IP] ), 100 )  ! PERMITTED: ACTUAL IS EXPRESSION
```

The rules of resolving generic procedure references are the same as those in the Fortran standard.

The following restrictions affect co-arrays used in procedures:

- A function result is not permitted to be a co-array.
- A pure procedure is not permitted to contain any co-arrays.

### 10.14.6 Specifying Co-arrays in COMMON and EQUIVALENCE Statements

Co-arrays can be specified in COMMON statements. For example:

```
COMMON /CCC/ W1(100)[*], W2(100)[16,*] ! PERMITTED:
                                       ! CO-ARRAYS IN COMMON
```

The layout of the common block on any one image is as if all objects of the common block were declared without co-dimensions.

Data objects that are not co-array data objects can appear in the same common block as co-arrays.

Co-arrays can be specified in EQUIVALENCE statements, but bracket references cannot appear in EQUIVALENCE statements. For example:

```
REAL :: V1(100)[*], V2(100)[*], V3(100)

EQUIVALENCE ( V1(50), V2(1) )           ! PERMITTED: CO-ARRAYS
EQUIVALENCE ( V1(1)[16], V2(1)[1] )     ! NOT PERMITTED:
                                        ! SQUARE BRACKETS
```

Data objects that are not co-array data objects cannot be equivalenced to co-array data objects. For example:

```
EQUIVALENCE (V1(50), V3(1)) ! NOT PERMITTED: V3 NOT
                            ! CO-ARRAY OBJECT
```

### 10.14.7 Allocatable Co-arrays

A co-array can be allocatable. Co-dimensions are specified by appending brackets containing the co-dimension specification to the co-array local specification in an ALLOCATE statement. For example:

```
REAL, ALLOCATABLE :: A1(:)[:], A2(:)[:,:]

ALLOCATE ( A1(10)[*])          ! PERMITTED: ALLOCATABLE CO-ARRAY
ALLOCATE ( A2(24)[0:7,0:*] )
```

As with the specification of statically allocated co-arrays, the upper bound of the final co-dimension must be an asterisk (*) and the values of all other bounds must be identical across all images.

**Caution:** Execution of ALLOCATE and DEALLOCATE statements containing co-array objects causes an implicit barrier synchronization of all images. All images must participate in the execution of these statements, or deadlock can occur.

### 10.14.8 Pointer Components in Derived Type Co-arrays

A pointer cannot be declared as a co-array, but a co-array can be of a derived type containing a pointer component. This enables construction of irregularly sized data structures across images and indirect addressing of non-co-array data. For example:

```
TYPE DTYPE4
   INTEGER :: LEN
   REAL, POINTER :: AP(:)
END TYPE DTYPE4

TYPE(DTYPE4) :: D4[*]          ! PERMITTED: CO-ARRAY OF DERIVED
                               ! TYPE CONTAINING POINTER
```

To help prevent the possibility of pointers being assigned invalid data, co-array bracket references cannot appear in pointer assignment statements. For example:

```
REAL :: Q(100)

D4[IP]%AP => Q                   ! NOT PERMITTED: BRACKET IN
Q => D4[IP]%AP                   ! POINTER ASSIGNMENT
```

Pointer components of a co-array can be associated only with a local target, either through pointer assignment or allocation.

### 10.14.9 Allocatable Components in Derived Type Co-arrays

Co-array derived types are allowed to have allocatable components. This enables construction of irregularly sized data structures across images.

```
TYPE DTYPE4
   INTEGER :: LEN
   REAL, ALLOCATABLE :: AP(:)
END TYPE DTYPE4

TYPE(DTYPE4) :: D4[*]          ! PERMITTED: CO-ARRAY OF DERIVED
                               ! TYPE CONTAINING ALLOCATABLE COMPONENT
```

A bracket reference to a allocatable component in a derived type co-array returns the value from the object on the specified image. For example, the reference `D4[7]%AP(22)` returns the value of `D4%AP(22)` as evaluated on image 7.

Allocatable components are allocated independently on each image. The allocation must not include square brackets.

### 10.14.10 Intrinsic Procedures

These co-array intrinsics return information about images:

- `LOG2_IMAGES` returns the base 2 logarithm of the number of executing images, truncated to an integer
- `NUM_IMAGES` returns the total number of Co-array Fortran images executing
- `REM_IMAGES` returns `MOD(NUM_IMAGES(), 2**LOG2_IMAGES())`
- `THIS_IMAGE` returns the index of, or co-subscripts related to, the invoking image

Only `NUM_IMAGES`, `LOG2_IMAGES`, and `REM_IMAGES` can appear in specification statements. None of the intrinsics are permitted in initialization expressions.

These co-array intrinsic subroutines synchronize access to co-array data among the images:

- SYNC_ALL
- SYNC_TEAM
- SYNC_MEMORY
- START_CRITICAL and END_CRITICAL
- SYNC_FILE

The following sections contain more information about these intrinsic procedures.

### 10.14.11 Program Synchronization

Co-arrays provide synchronization procedures which allow you to ensure that access to co-array data in primary memory is coherent (reliable) across all images or a group of images called a team. That is, any image that modifies co-array data that is expected to be read by another image, or any image that reads data that is modified by another image must call the co-array synchronization intrinsic functions to ensure valid data is accessed by other images.

#### 10.14.11.1 SYNC_ALL

The SYNC_ALL intrinsic guarantees to all images executing a corresponding call to SYNC_ALL that the calling procedure has completed all preceding accesses to co-array data. The access must either be a direct read or write of the data or a procedure call that references the data. The SYNC_ALL intrinsic returns when all images have made a corresponding call to SYNC_ALL.

For example, consider the following subroutine:

```
SUBROUTINE TST(A,B,C,D,N,IP)
REAL :: A(N)[*], B(N)[*], C(N)[*], D(N)

A(:) = B(:)[IP]
CALL SUB1(C,N)
D(:) = 0.0

CALL SYNC_ALL()

END
```

When an image executes the SYNC_ALL call as in the preceding example, it guarantees to all images executing a corresponding SYNC_ALL call that its access to A and B are complete and that all accesses to C by SUB1 are complete. It does not guarantee that its accesses to D are complete, since D is not declared as a co-array. This is true even if the actual argument for D is a co-array.

Access behavior to the same data by different images without such corresponding synchronization calls is undefined.

This is the syntax of the SYNC_ALL intrinsic:

```
CALL SYNC_ALL([wait])
```

Calling SYNC_ALL without the *wait* argument is the same as calling SYNC_TEAM(*all*), where *all* has the value `(/ (I,I=1,num_images()) /)`. Calling SYNC_ALL(*wait*) is the same as calling SYNC_TEAM(*all*, *wait*). See Section 10.14.11.2, page 221 for more information about the *wait* argument.

Calling SYNC_ALL implies a call to SYNC_MEMORY function.

### 10.14.11.2 SYNC_TEAM

The SYNC_TEAM intrinsic function can be used to synchronize a subset (or team) of images.

The syntax is:

```
CALL SYNC_TEAM(team [,wait])
```

The *team* argument specifies the members of the team. It has the INTENT(IN) attribute and can be either an integer array of rank one or an integer scalar.

To create a team of two or more, pass an integer array containing the image numbers of all members of the team, including the image calling SYNC_TEAM. Valid values for each element are from 1 through NUM_IMAGES inclusive. The array must not contain duplicate image numbers.

This example synchronizes a team consisting of images 2, 4, and 6:

**Note:** The calling image must be either image 2, 4 or 6.

```
CALL SYNC_TEAM((/2,4,6/))
```

You can also pass an integer scalar to create a team of two where the image calling SYNC_TEAM is an implied member of the team and the scalar integer specifies the image number of the other team member.

For example, this code synchronizes a team consisting of the executing image and image 4:

```
CALL SYNC_TEAM(4)
```

The presence of the optional *wait* argument tells an image to wait only for a subset of the team members to make a corresponding call. The *wait* argument has the INTENT(IN) attribute and is either an integer array or integer scalar.

For the array case, it contains the numbers of the images to wait for. It should contain no duplicate entries. The scalar case is treated as if the argument were the array (/*wait*/).

For example, this code synchronizes a team consisting of images 2, 4, 6, and 8, while waiting for images 4 and 6:

```
CALL SYNC_TEAM((/2,4,6,8/, /4,6/))
```

All images participating in a SYNC_TEAM call must call with identical arguments. Otherwise the results are undefined.

#### 10.14.11.3 SYNC_MEMORY

The SYNC_MEMORY intrinsic guarantees to other images that the image calling the function has completed all preceding accesses to co-array data.

This is the syntax of the intrinsic:

```
CALL SYNC_MEMORY()
```

#### 10.14.11.4 START_CRITICAL and END_CRITICAL

The START_CRITICAL and END_CRITICAL intrinsic functions mark the beginning and end of a critical section. Only one image at a time may execute statements in a critical region. If an image executes a START_CRITICAL intrinsic while another image is in the critical region, it waits. Also, both intrinsics, like the SYNC_MEMORY intrinsic, ensure that the calling image has completed all preceding accesses to co-array data.

This is the syntax of the intrinsics:

```
CALL START_CRITICAL()
CALL END_CRITICAL()
```

### Example 4: Using `START CRITICAL` and `END CRITICAL`

**Source code:**

```
program critical
implicit none
real :: sum_local, median_local
integer:: mype

! Distribute work and calculate sum and median values for each image
! For the sake of simplicity, we just assign values to sum_local and
! median_local

mype = this_image()
select case(mype)
case (1)
  sum_local = 1000.
  median_local = 1234.
case(2)
  sum_local = 2000.
  median_local = 2345.
end select

! By putting these write statements in a critical region, you will get
! readable contiguous output on stdout. Without the critical region,
! lines of output from various images could be intermixed and unreadable.

  call start_critical()

     write (*,*) "********** Results for end of pass 1 on image ",mype," *********"
     write (*,*) "   sum = ",sum_local
     write (*,*) "median = ",median_local
     write (*,*) "------------------------------------------------------------"
     write (*,*)

  call end_critical()

end program
```

**Commands to compile and run program:**

```
% ftn -o caf_critical -Z caf_critical.ftn
% module load pbs
% qsub -I -l mppe=2
```

```
qsub: waiting for job <job id> to start
% aprun -n 2 /ptmp/user1/caf_critical
```

**Output:**

```
********** Results for end of pass 1 on image  2  *********
    sum =  2000.
 median =  2345.
 ------------------------------------------------------------

 ********** Results for end of pass 1 on image  1  *********
    sum =  1000.
 median =  1234.
 ------------------------------------------------------------
```

#### 10.14.11.5 SYNC_FILE

To synchronize file accesses among images, use the SYNC_FILE intrinsic function. The intrinsic flushes data to a file to ensure that all images have access to valid data. The intrinsic affects only the I/O unit connected an image. If the unit is not connected or does not exist, the intrinsic has no effect. If the unit is connected for sequential access, a call to SYNC_FILE causes all WRITE requests to advance input or output.

This is the syntax of SYNC_FILE:

```
CALL SYNC_FILE(unit)
```

The *unit* argument is a scalar integer with the INTENT(IN) attribute. The *unit* argument specifies a Fortran I/O unit.

### 10.14.12 I/O with Co-arrays

An image can perform input only on the portion of a co-array that is local to that image. An image can perform output on any portion of a co-array. For example:

```
REAL :: X(100)[*]
 ...
READ *, X(I)      ! PERMITTED: LOCAL CO_ARRAY REFERENCE
READ *, X(I)[IP]  ! NOT PERMITTED: BRACKET CO-ARRAY REFERENCE
PRINT *, X(I)[IP] ! PERMITTED: OUTPUT OF BRACKET CO-ARRAY REFERENCE
```

Each image has its own set of independent I/O units. A file can be opened on one image when it is already open on another, but only the BLANK, DELIM, PAD, ERR, and IOSTAT specifiers can have values that differ from those in effect on other images.

**Caution:** For a unit identified by an asterisk (*) in a READ or WRITE statement, there is a single position for all images. Only one image executes a statement for such a unit at any one time. The system introduces synchronization when necessary. Otherwise, each image positions each file independently. If the access order is important, the program must provide its own synchronization between images.

## 10.15 Compiling and Executing Programs Containing Co-arrays

There are various commands, tools, and products available in the programming environment to use for compiling and executing programs containing co-arrays.

### 10.15.1 ftn and aprun Options Affecting Co-arrays

The -Z compiler option on the ftn command line must be specified in order for co-array syntax to be recognized and translated. Otherwise, the co-array syntax generates ERROR messages.

Upon execution of an a.out file that has been compiled and loaded with the -Z 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.

You can 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.

(X1 only) Processing elements are either MSPs or SSPs. To run the images on SSPs, you must specify the -O ssp compiler option. To run on MSPs, you do not specify this option. For more information about SSP and MSP mode, see Section 3.19.21, page 55.

Bounds checking is performed by specifying the -Rb option on the ftn command line. This feature is not implemented for co-dimensions of co-arrays.

For more information about the ftn and aprun commands, see the ftn(1) and aprun(1) man pages.

### 10.15.2 Using the CrayTools Tool Set with Co-array Programs

The CrayTools tool set, which includes TotalView, and Cray performance analyzer tool (CrayPat), does not contain special support for co-arrays and does not support the bracket notation. In most cases, however, these tools can still be used effectively to analyze programs containing co-arrays.

The following sections discuss issues related to the interaction of these tools with programs containing co-arrays.

#### 10.15.2.1 Debugging Programs Containing Co-arrays (Deferred implementation)

The `totalview` debugger does not support the bracket notation. Co-arrays generally appear as their corresponding local object with co-dimensions stripped off.

Co-array data can be viewed and referenced by switching the `totalview Process` window to the processing element corresponding to the desired image and accessing the co-array with local references.

#### 10.15.2.2 Analyzing Co-array Program Performance

To the CrayTools performance tools, which include CrayPat, co-arrays generally appear as their corresponding local object with co-dimensions stripped off. For more information about CrayPat, see *Optimizing Applications on Cray X1 Series Systems*.

**Caution:** References to co-arrays on different images appear to the performance tools as local data references. This may skew the remote reference statistics of these tools.

### 10.15.3 Interoperating with Other Message Passing and Data Passing Models

Co-arrays can interoperate with all other message and data passing models: MPI and SHMEM. This allows you to introduce co-arrays into existing application codes incrementally.

These models are implemented through procedure calls, so the language interaction between co-arrays and these models is well defined. For more information about passing co-arrays to procedure calls, see Section 10.14.5, page 216.

**Caution:** MPI and SHMEM generally use processing element numbers, which start at zero, but the co-array model generally deals with image numbers, which start at one. For information about the mapping between processing elements and image numbers, see Section 10.15.1, page 225

Co-arrays are symmetric for the purposes of SHMEM programming. Pointers in co-arrays of derived type, however, may not necessarily point to symmetric data.

For more information about the other message passing and data passing models, see one of the following publications:

- *Message Passing Toolkit Release Overview*
- `intro_shmem`(3) command and man page.

### 10.15.4 Optimizing Programs with Co-arrays

Programs containing co-arrays benefit from all the usual steps you can take to improve run-time performance of code that runs on a single image.

Loops containing references to co-arrays can and should be vectorized. On UNICOS/mp systems such loops may also be multistreamed. On UNICOS/mp systems if a co-array vector memory reference references multiple images, you may receive a "No Forward Progress" exception. In this case, you should try vectorizing along a different dimension of the co-array or running the application in accelerated mode (`aprun -A`).

# Obsolete Features [11]

The Cray Fortran compiler supports legacy features to allow the continued use of existing codes. In general, these features should not be used in new codes. The obsolete features are divided into two groups. The first is the set of features identified in Annex B of the Fortran standard as deleted. These were part of the Fortran language but their usage is explicitly discouraged in new codes. The second group is the set of legacy extensions supported in the Cray compiler for which preferred alternatives now exist. The obsolete features and their preferred alternatives are listed in Table 22.

Table 22. Obsolete Features and Preferred Alternatives

| **Obsolete feature** | **Preferred alternative** |
|---|---|
| `IMPLICIT UNDEFINED` | `IMPLICIT NONE` |
| Type statements with `*`*n* | Type statements with `KIND=` parameters |
| `BYTE` data type | `INTEGER(KIND=1)` |
| `DOUBLE COMPLEX` statement | `COMPLEX` statement with `KIND` parameter |
| `STATIC` attribute and statement | `SAVE` attribute and statement |
| Slash data initialization | Standard initialization syntax |
| `DATA` statement features | Standard conforming `DATA` statements |
| Hollerith data | Character data |
| `PAUSE` statement | `READ` statement |
| `ASSIGN`, assigned `GOTO` statements and assigned format specifiers | Standard branching constructs |
| Two-branch `IF` statements | `IF` construct or statement |
| Real and double precision `DO` variables | Integer `DO` variables |
| Nested loop termination | Separate `END DO` statements |
| Branching into a block | Restructure code |
| `ENCODE` and `DECODE` statements | `WRITE` and `READ` with internal file |
| `BUFFER IN` and `BUFFER OUT` statements | Asynchronous I/O statements |
| Asterisk character constant delimiters | Use standard character delimiters |
| Negative-values `X` descriptor | `TL` descriptor |

| Obsolete feature | Preferred alternative |
|---|---|
| A descriptor used for noncharacter conventional data and R descriptor | Character type and other conventional matchings of data and descriptors |
| H edit descriptor | Character constants |
| Obsolete intrinsic procedures | For list and replacements, see Section 11.21, page 250 |
| Initialization using long strings | Replace the numeric target with a character item. Replace a Hollerith constant with a character constant |

## 11.1 IMPLICIT UNDEFINED

The Cray Fortran compiler accepts the `IMPLICIT UNDEFINED` statement. It is equivalent to the `IMPLICIT NONE` statement.

## 11.2 Type statement with *n

The Cray Fortran compiler defines the following additional forms of *type_declaration_stmt*:

| | | |
|---|---|---|
| *type_spec* | **is** | `INTEGER*` *length_value* |
| | **or** | `REAL*` *length_value* |
| | **or** | `DOUBLE PRECISION*` *length_value* |
| | **or** | `COMPLEX*` *length_value* |
| | **or** | `LOGICAL*` *length_value* |

- *length-value* is the size of the data object in bytes.

Data type declarations that include the data length are outmoded. The Cray Fortran compiler recognizes this usage in type statements, `IMPLICIT` statements, and `FUNCTION` statements, mapping these numbers onto kind values appropriate for the target machine.

## 11.3 BYTE Data Type

The `BYTE` statement and data type declares a 1–byte value. This data type is equivalent to the `INTEGER(KIND=1)` and `INTEGER*1` declarations.

## 11.4 DOUBLE COMPLEX Statement

The DOUBLE COMPLEX statement is used to declare an item to be of type double complex. The format for the DOUBLE COMPLEX statement is as follows:

```
DOUBLE COMPLEX [ , attribute-list :: ] entity-list
```

Items declared as DOUBLE COMPLEX contain two double precision entities.

When the -dp option is in effect, double complex entities are affected as follows:

- The nonstandard DOUBLE COMPLEX declaration is treated as a single-precision complex type.
- Double precision intrinsic procedures are changed to the corresponding single-precision intrinsic procedures.

The -ep or -dp specification is used for all source files compiled with a single invocation of the Cray Fortran compiler command. If a module is compiled separately from a program unit that uses the module, they both shall be compiled with the same -ep or -dp specification.

## 11.5 STATIC Attribute and Statement

The STATIC attribute and statement provides the same effect as the SAVE attribute and statement. Variables with the Cray Fortran STATIC attribute retain their value and their definition, association, and allocation status after the subprogram in which they are declared completes execution. Variables without this attribute cannot be depended on to retain its value and status, although the Cray Fortran compiler treats named common blocks as if they had this attribute. This attribute should always be specified for an object or the object's common named block, if it is necessary for the object to retain its value and status.

In Cray's implementation, the system retains the value of an object that is in a module whether or not the STATIC specifier is used.

Objects declared in recursive subprograms can be given the attribute. Such objects are shared by all instances of the subprogram.

Any object that is data initialized (in a DATA statement or a type declaration statement) has the STATIC attribute by default.

The following is a format for a type declaration statement with the attribute:

*type*, STATIC [, *attribute-list* ] :: *entity-decl-list*

| | | |
|---|---|---|
| *static-stmt* | **is** | STATIC [ [ :: ] *static-entity-list* ] |
| *static-entity* | **is** | *data-object-name* |
| | **or** | / *common-block-name* / |

A statement without an entity list is treated as though it contained the names of all items that could be saved in the scoping unit. The Cray Fortran compiler allows you to insert multiple statements without entity lists in a scoping unit.

If STATIC appears in a main program as an attribute or a statement, it has no effect.

The following objects must not be saved:

- A procedure
- A function result
- A dummy argument
- A named constant
- An automatic data object
- An object in a common block
- A namelist group

A variable in a common block cannot be saved individually; the entire named common block must be saved if you want any variables in it to be saved.

A named common block saved in one scoping unit of a program is saved throughout the program.

If a named common block is specified in a main program, it is available to any scoping unit of the program that specifies the named common block; it does not need to be saved.

The statement also confers the attribute. It is subject to the same rules and restrictions as the attribute.

The following example shows an entity-oriented declaration:

```
CHARACTER(LEN = 12), SAVE :: NAME
CHARACTER(LEN = 12), STATIC :: NAME
```

The following example shows an attribute-oriented declaration:

```
CHARACTER*12 NAME
STATIC NAME   !Use SAVE OR STATIC, but not both on the same name
```

The following example shows saving objects and named common blocks:

```
STATIC A, B, /BLOCKA/, C, /BLOCKB/
```

## 11.6 Slash Data Initialization

The Fortran type declaration statements provide a means for data initialization. For example, the following two methods are standard means for initializing integer data:

- Method 1:

  ```
  INTEGER :: I=3
  ```

- Method 2:

  ```
  INTEGER I
  DATA I /3/
  ```

The Cray Fortran compiler supports an additional method for each data type. The following example shows the additional, nonstandard method, used to define integer data:

- Method 3:

  ```
  INTEGER [::] I /3/
  ```

## 11.7 DATA Statement Features

The DATA statement has the following outmoded features:

- A constant need not exist for each element of a whole array named in a *data-stmt-object-list* if the array is the last item in the list.
- A Hollerith or character constant can initialize more than one element of an integer or single-precision real array if the array is specified without subscripts.

  Example 1: If the `-s default32` compiler option is used (default), an array is declared by `INTEGER A(2)`, the following DATA statements have the same effect:

```
DATA A /'12345678'/
DATA A /'1234','5678'/
```

  Example 2: If the `-s default64` compiler option is specified, an array is declared by `INTEGER A(2)`, the following DATA statements have the same effect:

```
DATA A /'1234567890123456'/
DATA A /'12345678','90123456'/
```

  An integer or single-precision real array can be defined in the same way in a DATA implied-DO statement.

## 11.8 Hollerith Data

Before the character data type was added to the Fortran 77 standard, Hollerith data provided a method of supplying character data.

### 11.8.1 Hollerith Constants

A Hollerith constant is expressed in one of three forms. The first of these is specified as a nonzero integer constant followed by the letter H, L, or R and as many characters as equal the value of the integer constant. The second form of Hollerith constant specification delimits the character sequence between a pair of apostrophes followed by the letter H, L, or R. The third form is like the second, except that quotation marks replace apostrophes. For example:

```
Character sequence:     ABC 12
Form 1:                 6HABC 12
Form 2:                 'ABC 12'H
Form 3:                 "ABC 12"H
```

Two adjacent apostrophes or quotation marks appearing between delimiting apostrophes or quotation marks are interpreted and counted by the compiler as a single apostrophe or quotation mark within the sequence. Thus, the sequence `DON'T USE "*"` would be specified with apostrophe delimiters as `'DON''T USE "*"'H`, and with quotation mark delimiters as `"DON'T USE ""*""H`.

Each character of a Hollerith constant is represented internally by an 8-bit code, with up to 32 such codes allowed. This limit corresponds to the size of the largest numeric type, `COMPLEX(KIND = 16)`. The ultimate size and makeup of the Hollerith data depends on the context. If the Hollerith constant is larger than the size of the type implied by context, the constant is truncated to the appropriate size. If the Hollerith constant is smaller than the size of the type implied by context, the constant is padded with a character dependent on the Hollerith indicator. When an H Hollerith indicator is used, the truncation and padding is done on the right end of the constant. The pad character is the blank character code (20).

Null codes can be produced in place of blank codes by substituting the letter L for the letter H in the Hollerith forms described above. The truncation and padding is also done on the right end of the constant, with the null character code (00) as the pad character.

Using the letter R instead of the letter H as the Hollerith indicator means truncation and padding is done on the left end of the constant with the null character code (00) used as the pad character.

All of the following Hollerith constants yield the same Hollerith constant and differ only in specifying the content and placement of the unused portion of the single 64-bit entity containing the constant:

| Hollerith | Internal byte, beginning on bit: | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| constant | 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
| 6HABCDEF | A | B | C | D | E | F | $20_{16}$ | $20_{16}$ |
| 'ABCDEF'H | A | B | C | D | E | F | $20_{16}$ | $20_{16}$ |
| "ABCDEF" H | A | B | C | D | E | F | $20_{16}$ | $20_{16}$ |
| 6LABCDEF | A | B | C | D | E | F | 00 | 00 |
| 'ABCDEF'L | A | B | C | D | E | F | 00 | 00 |
| "ABCDEF"L | A | B | C | D | E | F | 00 | 00 |
| 6RABCDEF | 00 | 00 | A | B | C | D | E | F |
| 'ABCDEF'R | 00 | 00 | A | B | C | D | E | F |
| "ABCDEF"R | 00 | 00 | A | B | C | D | E | F |

A Hollerith constant is limited to 32 characters except when specified in a CALL statement, a function argument list, or a DATA statement. An all-zero computer word follows the last word containing a Hollerith constant specified as an actual argument in an argument list.

A character constant of 32 or fewer characters is treated as if it were a Hollerith constant in situations where a character constant is not allowed by the standard but a Hollerith constant is allowed by the Cray Fortran compiler. If the character constant appears in a DATA statement value list, it can be longer than 32 characters.

### 11.8.2 Hollerith Values

A *Hollerith value* is a Hollerith constant or a variable that contains Hollerith data. A Hollerith value is limited to 32 characters.

A Hollerith value can be used in any operation in which a numeric constant can be used. It can also appear on the right-hand side of an assignment statement in which a numeric constant can be used. It is truncated or padded to be the correct size for the type implied by the context.

### 11.8.3 Hollerith Relational Expressions

Used with a relational operator, the Hollerith value $e_1$ is less than $e_2$ if its value precedes the value of $e_2$ in the collating sequence and is greater if its value follows the value of $e_2$ in the collating sequence.

The following examples are evaluated as true if the integer variable LOCK contains the Hollerith characters K, E, and Y in that order and left-justified with five trailing blank character codes:

```
3HKEY.EQ.LOCK
'KEY'.EQ.LOCK
LOCK.EQ.LOCK
'KEY1'.GT.LOCK
'KEY0'H.GT.LOCK
```

## 11.9 PAUSE Statement

Execution of a PAUSE statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate READ statement that awaits some input data.

The execution of the PAUSE statement suspends the execution of a program. This is now redundant, because a WRITE statement can be used to send a message to any device, and a READ statement can be used to wait for and receive a message from the same device.

The PAUSE statement is defined as follows:

| *pause-stmt* | **is** PAUSE [ *stop-code* ] |
|---|---|

The character constant or list of digits identifying the PAUSE statement is called the *stop-code* because it follows the same rules as those for the STOP statement's stop code. The stop code is accessible following program suspension. The Cray Fortran compiler sends the *stop-code* to the standard error file (stderr). The following are examples of PAUSE statements:

```
PAUSE
PAUSE 'Wait #823'
PAUSE 100
```

## 11.10 ASSIGN, Assigned GO TO Statements, and Assigned Format Specifiers

The ASSIGN statement assigns a statement label to an integer variable. During program execution, the variable can be assigned labels of branch target statements, providing a dynamic branching capability in a program. The unsatisfactory property of these statements is that the integer variable name can be used to hold both a label and an ordinary integer value, leading to errors that can be hard to discover and programs that can be difficult to read.

A frequent use of the ASSIGN statement and assigned GO TO statement is to simulate internal procedures, using the ASSIGN statement to record the return point after a reusable block of code has completed. The internal procedure mechanism of Fortran now provides this capability.

A second use of the ASSIGN statement is to simulate dynamic format specifications by assigning labels corresponding to different format statements to an integer variable and using this variable in I/O statements as a format specifier. This use can be accomplished in a clearer way by using character strings as format specifications. Thus, it is no longer necessary to use either the ASSIGN statement or the assigned GO TO statement.

Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined. Variables associated with the variable in an ASSIGN statement, however, become undefined as integers when the ASSIGN statement is executed. When an entity of double precision real type becomes defined, all totally associated entities of double precision real type become defined.

Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.

### 11.10.1 Form of the ASSIGN and Assigned GO TO Statements

Execution of an ASSIGN statement assigns a label to an integer variable. Subsequently, this value can be used by an assigned GO TO statement or by an I/O statement to reference a FORMAT statement. The ASSIGN statement is defined as follows:

| | | |
|---|---|---|
| *assign-stmt* | **is** | ASSIGN *label* TO *scalar-int-variable* |

The term *default integer type* in this section means that the integer variable shall occupy a full word in order to be able to hold the address of the statement label. Programs that contain an ASSIGN statement and are compiled with `-s default32` shall ensure that the *scalar-int-variable* is declared as `INTEGER(KIND=8)`. This ensures that it occupies a full word.

The variable shall be a named variable of default integer type. It shall not be an array element, an integer component of a structure, or an object of nondefault integer type.

The label shall be the label of a branch target statement or the label of a FORMAT statement in the same scoping unit as the ASSIGN statement.

When defined with an integer value, the integer variable cannot be used as a label.

When assigned a label, the integer variable cannot be used as anything other than a label.

When the integer variable is used in an assigned GO TO statement, it shall be assigned a label.

As the following example shows, the variable can be redefined during program execution with either another label or an integer value:

```
ASSIGN 100 TO K
```

Execution of the assigned GO TO statement causes a transfer of control to the branch target statement with the label that had previously been assigned to the integer variable.

The assigned GO TO statement is defined as follows:

| *assigned-goto-stmt* | **is** GO TO *scalar-int-variable* [ [ , ] (*label-list*) ] |
|---|---|

The variable shall be a named variable of default integer type. That is, it shall not be an array element, a component of a structure, or an object of nondefault integer type.

The variable shall be assigned the label of a branch target statement in the same scoping unit as the assigned GO TO statement.

If a label list appears, such as in the following examples, the variable shall have been assigned a label value that is in the list:

```
GO TO K
GO TO K (10, 20, 100)
```

The ASSIGN statement also allows the label of a FORMAT statement to be dynamically assigned to an integer variable, which can later be used as a format specifier in READ, WRITE, or PRINT statements. This hinders readability, permits inconsistent usage of the integer variable, and can be an obscure source of error.

This functionality is available through character variables, arrays, and constants.

### 11.10.2 Assigned Format Specifiers

When an I/O statement containing the integer variable as a format specifier is executed, the integer variable can be defined with the label of a FORMAT specifier.

## 11.11 Two-branch IF Statements

Outmoded IF statements are the two-branch arithmetic IF and the indirect logical IF.

### 11.11.1 Two-branch Arithmetic IF

A two-branch arithmetic IF statement transfers control to statement $s_1$ if expression *e* is evaluated as nonzero or to statement $s_2$ if *e* is zero. The arithmetic expression should be replaced with a relational expression, and the statement should be changed to an IF statement or an IF construct. This format is as follows:

IF ( *e* ) $s_1$, $s_2$

*e* Integer, real, or double precision expression

*s* Label of an executable statement in the same program unit

Example:

```
IF (I+J*K) 100,101
```

### 11.11.2 Indirect Logical IF

An indirect logical IF statement transfers control to statement $s_t$ if logical expression *le* is true and to statement $s_f$ if *le* is false. An IF construct or an IF statement should be used in place of this outmoded statement. This format is as follows:

IF ( *le* ) $s_t$, $s_f$

*le* Logical expression

$s_t$, $s_f$ Labels of executable statements in the same program unit

Example:

```
IF(X.GE.Y)148,9999
```

## 11.12 Real and Double Precision DO Variables

The Cray Fortran compiler allows real variables and values as the DO variable and limits in DO statements. The preferred alternative is to use integer values and compute the desired real value.

## 11.13 Nested Loop Termination

Older Cray Fortran compilers allowed nested DO loops to terminate on a single END DO statement if the END DO statement had a statement label. The END DO statement is included in the Fortran standard. The Fortran standard specifies that a separate END DO statement shall be used to terminate each DO loop, so allowing nested DO loops to end on a single, labeled END DO statement is an outmoded feature.

## 11.14 Branching into a Block

Although the standard does not permit branching into the code block for a DO construct from outside of that construct, the Cray Fortran compiler permits branching into the code block for a DO or DO WHILE construct. By default, the Cray Fortran compiler issues an error for this situation. Cray does not recommend branching into a DO construct, but if you specify the `ftn -eg` command, the code will compile.

## 11.15 ENCODE and DECODE Statements

A formatted I/O operation defines entities by transferring data between I/O list items and records of a file. The file can be on an external media or in internal storage.

The Fortran standard provides READ and WRITE statements for both formatted external and internal file I/O. This is the preferred method for formatted internal file I/O. It is the only method for list-directed internal file I/O.

The ENCODE and DECODE statements are an alternative to standard Fortran READ and WRITE statements for formatted internal file I/O.

An internal file in standard Fortran I/O shall be declared as character, while the internal file in ENCODE and DECODE statements can be any data type. A record in an internal file in standard Fortran I/O is either a scalar character variable or an array element of a character array. The record size in an internal file in an ENCODE or DECODE statement is independent of the storage size of the variable used as the internal file. If the internal file is a character array in standard Fortran I/O, multiple records can be read or written with internal file I/O. The alternative form does not provide the multiple record capability.

### 11.15.1 ENCODE Statement

The ENCODE statement provides a method of converting or encoding the internal representation of the entities in the output list to a character representation. The format of the ENCODE statement is as follows:

```
ENCODE ( n, f, dest ) [ elist ]
```

*n* — Number of characters to be processed. Nonzero integer expression not to exceed the maximum record length for formatted records. This is the record size for the internal file.

*f* — Format identifier. It cannot be an asterisk.

*dest* — Name of internal file. It can be a variable or array of any data type. It cannot be an array section, a zero-sized array, or a zero-sized character variable.

*elist* — Output list to be converted to character during the ENCODE statement.

The output list items are converted using format *f* to produce a sequence of *n* characters that are stored in the internal file *dest*. The *n* characters are packed 8 characters per word.

An ENCODE statement transfers one record of length *n* to the internal file *dest*. If format *f* attempts to write a second record, ENCODE processing repositions the current record position to the beginning of the internal file and begins writing at that position.

An error is issued when the ENCODE statement attempts to write more than *n* characters to the record of the internal file. If *dest* is a noncharacter entity and *n* is not a multiple of 8, the last word of the record is padded with blanks to a word boundary. If *dest* is a character entity, the last word of the record is not padded with blanks to a word boundary.

Example 1: The following example assumes a machine word length of 64 bits and uses the underscore character (_) as a blank:

```
      INTEGER ZD(5), ZE(3)
      ZD(1)='THIS____'
      ZD(2)='MUST____'
      ZD(3)='HAVE____'
      ZD(4)='FOUR____'
      ZD(5)='CHAR____'
1     FORMAT(5A4)
      ENCODE(20,1,ZE)ZD
      DO 10 I=1,3
        PRINT 2,'ZE(',I,')="',ZE(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A8,A)
      END
```

The output is as follows:

```
>ZE( 1)="THISMUST"
>ZE( 2)="HAVEFOUR"
>ZE( 3)="CHAR____"
```

### 11.15.2 DECODE Statement

The DECODE statement provides a method of converting or decoding from a character representation to the internal representation of the entities in the input list. The format of the DECODE statement is as follows:

```
DECODE ( n, f, source ) [ dlist ]
```

*n* Number of characters to be processed. Nonzero integer expression not to exceed the maximum record length for formatted records. This is the record size for the internal file.

*f* Format identifier. It cannot be an asterisk.

*source* Name of internal file. It can be a variable or array of any data type. It cannot be an array section or a zero-sized array or a zero-sized character variable.

*dlist* Input list to be converted from character during the DECODE statement.

The input list items are converted using format *f* from a sequence of *n* characters in the internal file *source* to an internal representation and stored in the input list entities. If the internal file *source* is noncharacter, the internal file is assumed to be a multiple of 8 characters.

Example 1: An example of a DECODE statement is as follows:

```
      INTEGER ZD(4), ZE(3)
      ZE(1)='WHILETHI'
      ZE(2)='S HAS  F'
      ZE(3)='IVE     '
3     FORMAT(4A5)
      DECODE(20,3,ZE)ZD
      DO 10 I=1,4
         PRINT 2,'ZD(',I,')="',ZD(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A8,A)
      END
```

The output is as follows:

```
>ZD( 1)="WHILE   "
>ZD( 2)="THIS    "
>ZD( 3)="HAS     "
>ZD( 4)="FIVE    "
```

## 11.16 BUFFER IN and BUFFER OUT Statements

You can use the BUFFER IN and BUFFER OUT statements to transfer data.

Data can be transferred while allowing the subsequent execution sequence to proceed concurrently. This is called *asynchronous I/O*. Asynchronous I/O may require the use of nondefault file formats or FFIO layers, as discussed in Chapter 15, page 295. BUFFER IN and BUFFER OUT operations may proceed concurrently on several units or files. If they do not proceed asynchronously, they will use synchronous I/O.

BUFFER IN is for reading, and BUFFER OUT is for writing. A BUFFER IN or BUFFER OUT operation includes only data from a single array or a single common block.

Either statement initiates a data transfer between a specified file or unit (at the current record) and memory. If the unit or file is completing an operation initiated by any earlier BUFFER IN or BUFFER OUT statement, the current BUFFER IN or BUFFER OUT statement suspends the execution sequence until the earlier operation is complete. When the unit's preceding operation terminates, execution of the BUFFER IN or BUFFER OUT statement completes as if no delay had occurred.

You can use the UNIT(3i) or LENGTH(3i) intrinsic procedures to delay the execution sequence until the BUFFER IN or BUFFER OUT operation is complete. These functions can also return information about the I/O operation at its termination.

The general format of the BUFFER IN and BUFFER OUT statements follows:

| | | |
|---|---|---|
| *buffer_in_stmt* | **is** | BUFFER IN (*id*, *mode*) (*start_loc*, *end_loc*) |
| *buffer_out_stmt* | **is** | BUFFER OUT (*id*, *mode*) (*start_loc*, *end_loc*) |
| *io_unit* | **is** | *external_file_unit* |
| | **or** | *file_name_expr* |
| *mode* | **is** | *scalar_integer_expr* |
| *start_loc* | **is** | *variable* |
| *end_loc* | **is** | *variable* |

In the preceding definition, the *variable* specified for *start_loc* and *end_loc* cannot be of a derived type if you are performing implicit data conversion. The data items between *start_loc* and *end_loc* must be of the same type.

The BUFFER IN and BUFFER OUT statements are defined as follows.

BUFFER IN (*io_unit*, *mode*) (*start_loc*, *end_loc*)

BUFFER OUT (*io_unit*, *mode*) (*start_loc*, *end_loc*)

*io_unit* An identifier that specifies a unit. The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

*mode* Mode identifier. This integer expression controls the record position following the data transfer. The mode identifier is ignored on files that do not contain records; only full record processing is available.

*start_loc*, *end_loc*

Symbolic names of the variables, arrays, or array elements that mark the beginning and ending locations of the `BUFFER IN` or `BUFFER OUT` operation. These names must be either elements of a single array (or equivalenced to an array) or members of the same common block. If *start_loc* or *end_loc* is of type character, then both must be of type character. If *start_loc* and *end_loc* are noncharacter, then the item length of each must be equal.

For example, if the internal length of the data type of *start_loc* is 64 bits, the internal length of the data type of *end_loc* must be 64 bits. To ensure that the size of *start_loc* and *end_loc* are the same, use the same data type for both.

The mode identifier, *mode*, controls the position of the record at unit *io_unit* after the data transfer is complete. The values of *mode* have the following effects:

- Specifying *mode* ≥ `0` causes full record processing. File and record positioning works as with conventional I/O. The record position following such a transfer is always between the current record (the record with which the transfer occurred) and the next record. Specifying `BUFFER OUT` with *mode* ≥ `0` ends a series of partial-record transfers.
- Specifying *mode* < `0` causes partial record processing. In `BUFFER IN`, the record is positioned to transfer its (*n* +1)th word if the *n*th word was the last transferred. In `BUFFER OUT`, the record is left positioned to receive additional words.

The amount of data to be transferred is specified in words without regard to types or formats. However, the data type of *end_loc* affects the exact ending location of a transfer. If *end_loc* is of a multiple-word data type, the location of the last word in its multiple-word form of representation marks the ending location of the data transfer.

`BUFFER OUT` with *`start_loc = end_loc +`* `1` and *mode* ≥ `0` causes a zero-word transfer and concludes the record being created. Except for terminating a partial record, *start_loc* following *end_loc* in a storage sequence causes a run-time error.

Example:

```
  PROGRAM XFR
  DIMENSION A(1000), B(2,10,100), C(500)
  ...
  BUFFER IN(32,0) (A(1),A(1000))
  ...
  DO 9 J=1,100
    B(1,1,J) = B(1,1,J) + B(2,1,J)
9 CONTINUE
  BUFFER IN(32,0) (C(1),C(500))
  BUFFER OUT(22,0) (A(1),A(1000))
  ...
  END
```

The first BUFFER IN statement in this example initiates a transfer of 1000 words from unit 32. If asynchronous I/O is available, processing unrelated to that transfer proceeds. When this is complete, a second BUFFER IN is encountered, which causes a delay in the execution sequence until the last of the 1000 words is received. A transfer of another 500 words is initiated from unit 32 as the execution sequence continues. BUFFER OUT begins a transfer of the first 1000 words to unit 22. In all cases *mode* = 0, indicating full record processing.

## 11.17 Asterisk Delimiters

The asterisk was allowed to delimit a literal character constant. It has been replaced by the apostrophe and quotation mark.

$*h_1 \ h_2 \ \ldots \ h_n*$

| | |
|---|---|
| * | Delimiter for a literal character string |
| *h* | Any ASCII character indicated by a C that is capable of internal representation |

Example:

```
*AN ASTERISK EDIT DESCRIPTOR*
```

## 11.18 Negative-valued X Descriptor

A negative value could be used with the X descriptor to indicate a move to the left. This has been replaced by the TL descriptor.

[-*b*]X

| | |
|---|---|
| *b* | Any nonzero, unsigned integer constant |
| X | Indicates a move of as many positions as indicated by *b* |

Example:

```
-55X   ! Moves current position 55 spaces left
```

## 11.19 A and R Descriptors for Noncharacter Types

The R*w* descriptor and the use of the A*w* descriptor for noncharacter data are available primarily for programs that were written before a true character type was available. Other uses include adding labels to binary files and the transfer of data whose type is not known in advance.

List items can be of type real, integer, complex, or logical. For character use, the binary form of the data is converted to or from ASCII codes. The numeric list item is assumed to contain ASCII characters when used with these edit descriptors.

Complex items use two storage units and require two A descriptors, for the first and second storage units respectively.

The A*w* descriptor works with noncharacter list items containing character data in essentially the same way as described in the Fortran standard. The R*w* descriptor works like A*w* with the following exceptions:

- Characters in an incompletely filled input list item are right-justified with the remainder of that list item containing binary zeros.
- Partial output of an output list item is from its rightmost character positions.

The following example shows the A*w* and R*w* edit descriptors for noncharacter data types:

```
      INTEGER IA
      LOGICAL LA
      REAL RA
      DOUBLE PRECISION DA
      COMPLEX CA
      CHARACTER*52 CHC
      CHC='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
      READ(CHC,3) IA, LA, RA, DA, CA
3     FORMAT(A4,A8,A10,A17,A7,A6)
      PRINT 4, IA, LA, RA, DA, CA
4     FORMAT(1x,3(A8,'-'),A16,'-',2A8)
      READ(CHC,5) IA, LA, RA
5     FORMAT(R2,R8,R9)
      PRINT 4, IA, LA, RA
      END
```

The output of this program would be as follows:

```
> ABCD    -EFGHIJKL-OPQRSTUV-XYZabcdefghijklm-nopqrst uvwxyz
> ooooooAB-CDEFGHIJ-LMNOPQRS-
```

The arrow (>) indicates leading blanks in the use of the A edit descriptor. The lowercase letter o is used to indicate where binary zeros have been written with the R edit descriptor.

The binary zeros are not printable characters, so the printed output simply contains the characters without the binary zeros.

## 11.20 H Edit Descriptor

This edit descriptor can be a source of error because the number of characters following the descriptor can be miscounted easily. The same functionality is available using the character constant edit descriptor, for which no count is required.

The following information pertains to the H edit descriptor:

Table 23. Summary of String Edit Descriptors

| Descriptor | Description |
|---|---|
| H | Transfer of text character to output record |
| *'text'* | Transfer of a character literal constant to output record |
| *"text"* | Transfer of a character literal constant to output record |

## 11.21 Obsolete Intrinsic Procedures

The Cray Fortran compiler supports many intrinsic procedures that have been used in legacy codes, but that are now obsolete. The following table indicates the obsolete procedures and the preferred alternatives. For more information about a particular procedure, see its man page.

Table 24. Obsolete Procedures and Alternatives

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| AND | IAND |
| BITEST | BTEST |
| BJTEST | BTEST |
| BKTEST | BTEST |
| CDABS | ABS |
| CDCOS | COS |
| CDEXP | EXP |
| CDLOG | LOG |
| CDSIN | SIN |
| CDSQRT | SQRT |
| CLOC | LOC or C_LOC |
| COMPL | NOT |
| COTAN | COT |
| CQABS | ABS |
| CQDEXP | EXP |
| CQSIN | SIN |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| CQSQRT | SQRT |
| CSMG | MERGE |
| CVMGM | MERGE |
| CVMGN | MERGE |
| CVMGP | MERGE |
| CVMGZ | MERGE |
| CVMGT | MERGE |
| DACOSD | ACOSD |
| DASIND | ASIND |
| DATAN2D | ATAN2D |
| DATAND | ATAND |
| DCMPLX | CMPLX |
| DCONJG | CONJG |
| DCOSD | COSD |
| DCOT | COT |
| DCOTAN | COT |
| DFLOAT | REAL |
| DFLOATI | REAL |
| DFLOATJ | REAL |
| DFLOATK | REAL |
| DIMAG | AIMAG |
| DREAL | REAL |
| DSIND | SIND |
| DTAND | TAND |
| EQV | NOT, IEOR |
| FCD | (none) |
| FLOATI | REAL |
| FLOATJ | REAL |
| FLOATK | REAL |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| FP_CLASS | IEEE_CLASS |
| IDATE | DATE_AND_TIME |
| IEEE_REAL | REAL |
| IIABS | ABS |
| IIAND | IAND |
| IIBCHNG | IBCHNG |
| IIBCLR | IBCLR |
| IIBITS | IBITS |
| IIBSET | IBSET |
| IIEOR | IEOR |
| IIDIM | DIM |
| IIDINT | INT |
| IIFIX | INT |
| IINT | INT |
| IIOR | IOR |
| IIQINT | INT |
| IISHA | SHIFTA |
| IISHC | ISHFT |
| IISHFT | ISHFTC |
| IISHFTC | ISHFTC |
| IISHL | ISHFT |
| IISIGN | SIGN |
| IMAG | AIMAG |
| IMOD | MOD |
| ININT | NINT |
| INT2 | INT |
| INT4 | INT |
| INT8 | INT |
| INOT | NOT |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| IQNINT | NINT |
| IRTC | SYSTEM_CLOCK |
| ISHA | SHIFTA |
| ISHC | ISHFTC |
| ISHL | IEEE_IS_NAN |
| JDATE | DATE_AND_TIME |
| JFIX | INT |
| JIABS | ABS |
| JIAND | IAND |
| JIBCHNG | IBCHNG |
| JIBCLR | IBCLR |
| JIBITS | IBITS |
| JIBSET | IBSET |
| JIEOR | IEOR |
| JIDIM | DIM |
| JIDINT | INT |
| JIFIX | INT |
| JINT | INT |
| JIOR | IOR |
| JIQINT | INT |
| JISHA | SHIFTA |
| JISHC | ISHFTC |
| JISHFT | ISHFT |
| JISHFTC | ISHFTC |
| JISHL | ISHFT |
| JISIGN | SIGN |
| JMOD | MOD |
| JNINT | NINT |
| JNOT | NOT |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| KIABS | ABS |
| KIAND | IAND |
| KIBCHNG | IBCHNG |
| KIBCLR | IBCLR |
| KIBITS | IBITS |
| KIBSET | IBSET |
| KIEOR | IEOR |
| KIDIM | DIM |
| KIDINT | INT |
| KINT | INT |
| KIOR | IOR |
| KIQINT | INT |
| KISHA | SHIFTA |
| KISHC | ISHFTC |
| KISHFT | ISHFT |
| KISHFTC | ISHFTC |
| KISHL | ISHFT |
| KISIGN | SIGN |
| KMOD | MOD |
| KNINT | NINT |
| KNOT | NOT |
| LENGTH | (none) |
| LONG | INT |
| LSHIFT | ISHFT or SHIFTL |
| MY_PE | THIS_IMAGE |
| MEMORY_BARRIER | SYNC_MEMORY |
| NEQV | IEOR |
| OR | IOR |
| QABS | ABS |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| QACOS | ACOS |
| QACOSD | ACOSD |
| QASIN | ASIN |
| QASIND | ASIND |
| QATAN | ATAN |
| QATAN2 | ATAN2 |
| DATAN2D | ATAN2D |
| QATAND | ATAND |
| QCMPLX | CMPLX |
| QCONJG | CONJG |
| QCOS | COS |
| QCOSD | COSD |
| QCOSH | COSH |
| QCOT | COT |
| QCOTAN | COT |
| QDIM | DIM |
| QEXP | EXP |
| QEXT | REAL |
| QFLOAT | REAL |
| QFLOATI | REAL |
| QFLOATJ | REAL |
| QFLOATJ | REAL |
| QFLOATK | REAL |
| QIMAG | AIMAG |
| QINT | AINT |
| QLOG | LOG |
| QLOG10 | LOG10 |
| QMAX1 | MAX |
| QMIN1 | MIN |

| Obsolete Intrinsic Procedure | Replacement |
|---|---|
| QMOD | MOD |
| QNINT | ANINT |
| QREAL | REAL |
| QSIGN | SIGN |
| QSIN | SIN |
| QSIND | SIND |
| QSINH | SINH |
| QSQRT | SQRT |
| QTAN | TAN |
| QTAND | TAND |
| QTANH | TANH |
| RAN | RANDOM_NUMBER |
| RANF | RANDOM_NUMBER |
| RANGET | RANDOM_SEED |
| RANSET | RANDOM_SEED |
| REMOTE_WRITE_BARRIER | SYNC_MEMORY |
| RSHIFT | ISHFT or SHIFTR |
| RTC | SYSTEM_CLOCK |
| SECNDS | CPU_TIME |
| SHIFT | ISHFTC |
| SHORT | INT |
| SNGLQ | REAL |
| TIME | DATE_AND_TIME |
| UNIT | WAIT statement |
| WRITE_MEMORY_BARRIER | SYNC_MEMORY |
| XOR | IEOR |

# Cray Fortran Deferred Implementation and Optional Features [12]

The PE 6.0 release of the Cray Fortran compiler supports most of the features specified by the Fortran standard. One supported feature must be turned on with an option. This chapter identifies the Fortran 2003 features that are not fully supported. It is expected that these remaining features will be implemented in future releases of the Cray Fortran compiler.

## 12.1 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.

## 12.2 Finalizers

Type bound `FINAL` routines are not supported for polymorphic objects, and code is not generated to invoke final routines of polymorphic objects.

## 12.3 Restrictions on Unlimited Polymorphic Variables

If the `-e h` option is specified to cause packed storage for short integers and logicals, unlimited polymorphic variables whose dynamic types are `integer(1)`, `integer(2)`, `logical(1)`, or `logical(2)` are not supported.

## 12.4 Enhanced Expressions in Initializations and Specifications

The Fortran 2003 standard greatly expands the list of Fortran intrinsic functions that may be referenced in initialization and specification expressions, used mainly to create constants in declarations. Support for using some of these intrinsics, including the trigonometric intrinsic functions, is included in the PE 6.0 release, but the full list is not yet implemented.

## 12.5 User-defined, Derived Type I/O

User-defined, derived type I/O routines are not supported.

## 12.6 ENCODING= in I/O Statements

The ENCODING= specifier in I/O statements is accepted by the compiler but has no effect in the PE 6.0 release.

## 12.7 Allocatable Assignment (Optionally Enabled)

The Fortran 2003 standard allows an allocatable variable in an intrinsic assignment statement (`variable = expression`) to have a shape different from the expression. If the shapes are different, the variable is automatically deallocated and reallocated with the shape of the expression. This feature is available in the PE 6.0 Cray Fortran compiler but is not enabled by default because of potential adverse effects on performance. The new behavior is enabled by the `-e w` command line option.

# Cray Fortran Implementation Specifics [13]

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.

## 13.1 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.

## 13.2 `INCLUDE` Line

There is no limit to the nesting level for `INCULDE` 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.

## 13.3 `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 `-e h` command line option is specified. Integer values are represented as two's complement binary values.

## 13.4 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, 8, and 16 corresponding to the 32, 64, and 128 bit IEEE representations.

## 13.5 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 -dp command line option is 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 -dp option is specified, the compiler is not standard conforming.

## 13.6 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 -e 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.

## 13.7 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.

## 13.8 Cray Pointers

Cray pointers are 64-bit objects.

## 13.9 ENUM Kind

An enumerator that specifies the BIND(C) attribute creates values with a kind type parameter of 4.

## 13.10 Storage Issues

This section describes how the Cray Fortran compiler uses storage, including how this compiler accommodates programs that use overindexing of blank common.

### 13.10.1 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.

### 13.10.2 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 your 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 your 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.

### 13.10.3 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 UNICOS/mp and UNICOS/lc libraries. It is recommended that you modify programs rather than expand blank common blocks, particularly when migrating from other environments.

Figure 3 shows the structure of a program under the UNICOS/mp and UNICOS/lc operating systems in relation to expanding a blank common block. In both figures, the user area includes code, data, and common blocks.

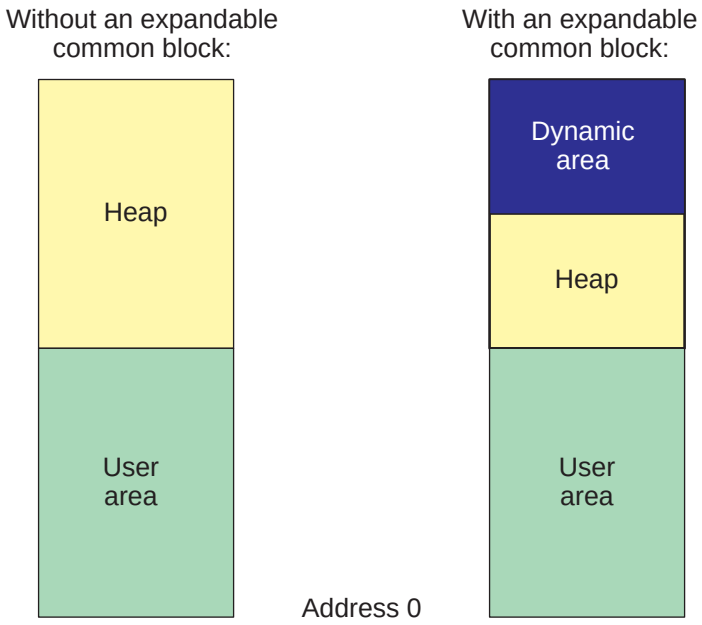


Figure 3. Memory Use

## 13.11 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.

## 13.12 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.

## 13.13 DEALLOCATE Error Status

If an error occurs during the execution of an DEALLOCATE statement with a stat= specifier, subsequent items in the deallocation list are not deallocated.

## 13.14 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.

## 13.15 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.

## 13.16 STOP Code Availability

If a STOP code is specified in a STOP statement, its value is output to the stderr file when the STOP statement is executed.

## 13.17 Stream File Record Structure and 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.

## 13.18 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.

## 13.19 `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 1024.

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.

## 13.20 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.

## 13.21 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.

## 13.22 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.

### 13.22.1 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 will become a quiet NaN if the hexadecimal string is not present or is not a valid NaN.

A '+' or '-' preceding the NaN on input will be 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.

### 13.22.2 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:

1. 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)
   ```

2. 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.

3. 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
```

## 13.23 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 Section 4.1.5, page 83).

## 13.24 Random Number Generator

A linear congruential generator is used to produce the output of the RANDOM_NUMBER intrinsic subroutine. The seed array contains two 32-bit integer values.

## 13.25 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 `kind = 8` count variable.

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.

## 13.26 IEEE Intrinsic Modules

The IEEE intrinsics modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES are supplied. Denormal numbers are not supported on Cray X1 or X2 hardware. The IEEE_SUPPORT_DENORMAL inquiry function returns `.false.` for all kinds of arguments.

At the start of program execution, the IEEE halting modes are set such that `overflow`, `divide_by_zero`, and `invalid` exceptions cause a trap, while traps are disabled for `underflow` and `inexact`.

# Part III: Cray Fortran Application Programmer's I/O Reference

Part III describes advanced Fortran input/output (I/O) techniques for use on Cray X1 series systems. It includes the following chapters:

- Using the Assign Environment (Chapter 14, page 271)
- Using FFIO (Chapter 15, page 295)
- FFIO Layer Reference (Chapter 16, page 311)
- Creating a `user` Layer (Chapter 17, page 337)
- Numeric File Conversion Routines (Chapter 18, page 363)
- Named Pipe Support (Chapter 19, page 377)

The reader should be familiar with the information presented in the following Cray man pages:

- The `assign`(1), `assign`(3f), and `ffassign`(3f) man pages
- The `intro_ffio`(1) man page, which describes the FFIO system and performance options available with the FFIO layers

For additional information about I/O, see *Optimizing Applications on Cray X1 Series Systems*.

# Using the Assign Environment [14]

Fortran programs require the ability to alter many details of a Fortran file connection. You may need to specify device residency, an alternative file name, a file space allocation scheme, file structure, or data conversion properties of a connected file. These details comprise the *assign environment*.

In addition, Cray X1 series and X2 systems support *flexible file I/O* (FFIO), which uses layered I/O to implement sophisticated I/O strategies. When used in the context of the assign environment, FFIO enables you to implement different I/O techniques and realize significant improvements in I/O performance without modifying source code.

This chapter describes the `assign`(1) command and the `assign`(3f) library routine, which together define the assign environment.

The FFIO system is described in Chapter 15, page 295.

The `ffassign`(3c) command provides an interface to assign processing from C/C++. See the `ffassign`(3c) man page for details about its use.

## 14.1 assign Basics

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.

1. Set the FILENV environment variable to the desired path:

   set FILENV *environment-file*

2. Run the assign command to define the current assign environment:

   assign *arguments assign-object*

   For example:

   assign -F cachea g:su

3. Run your program:

   ./a.out *arguments*

4. If you are 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(1) command passes information to Fortran open statements and to the ffopen(3c) routine to identify the following elements:

- A list of unit 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.

### 14.1.1 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 becomes 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 might apply to a given open request:

- `g:all` options apply to any open request.
- `g:su`, `g:du`, `g:sf`, `g:df`, and `g:ff` all apply to types of open requests. These equate to sequential unformatted, direct unformatted, sequential formatted, direct formatted, or `ffopen`, respectively.
- `u:`*`unit-number`* applies whenever *unit-number* is opened.
- `p:`*`pattern`* applies whenever 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`* applies whenever a file with the name *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.

### 14.1.2 The `assign` Command

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 `assign` command options follows. For details, see the `assign`(1) and `intro_ffio`(3f) man pages.

Here are the `assign` command 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.

Here are the `assign` command 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 you specify the `-C` option, you must also specify the `-F` option.

`-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`, `-S`, 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 `off`.

`-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:`*`file-name`* (for example, `f:file1`)
- `g:`*`io-type`*; *`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`)
- *file-name* (for example, `myfile`)

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.

### 14.1.3 Assign Library Routines

The `assign`(3f), `asnunit`(3f), `asnfile`(3f), and `asnrm`(3f) 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 the `assign` library routines are as follows:

```
call assign (cmd, ier)

call asnunit (iunit,astring,ier)

call asnfile (fname,astring,ier)

call asnrm (ier)
```

*cmd* Fortran character variable that contains a complete `assign` command in the format that is also 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 and a status of greater than 0 indicates a specific error status. Use the `explain` command to determine the meaning of the error status. For more information about the `explain` command, see the `explain`(1) man page.

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 `assign` and Fortran I/O

Assign processing lets you tune file connections. This sections describes several areas of `assign` command usage and provide examples of each use.

### 14.2.1 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 is 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 is 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.

### 14.2.2 File Structure Selection

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 far more flexible than selecting structures explicitly. FFIO allows nested file structures, buffer size specifications, and support for file structures that are not available through the `-s` option. You will also realize better I/O performance by using the `-F` option and FFIO layers.

For more information about the `-F` option and FFIO layers, see Chapter 15, page 295.

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 list summarizes how to select the different file structures with different options to the assign command:

| Structure | assign command |
|---|---|
| F77 blocked | assign -F f77 |
| text | assign -F text<br>assign -s text |
| unblocked | assign -F system<br>assign -s unblocked |
| COS blocked | assign -F cos<br>assign -s cos |

The following examples address file structure selection:

- 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')
```

- You can use the assign -s u command to specify the unblocked file structure for a sequential unformatted file. When this option is selected, the I/O is unbuffered. Each Fortran READ or WRITE statement results in a read(2) or write(2) system call such as the following:

```
CALL ASNFILE('fort.1','-s u',IER)
OPEN(1,FORM='UNFORMATTED',ACCESS='SEQUENTIAL')
```

- Use the following command to assign unit 10 a COS blocked structure:

```
assign -s cos u:10
```

The full set of options allowed with the `assign -s` command are as follows:

- `bin` (not recommended)
- `blocked`
- `cos`
- `sbin`
- `text`
- `unblocked`

Table 25 summarizes the Fortran access methods and options.

Table 25. 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`<br>`sbin`<br>`u`<br>`unblocked` |
| Direct unformatted | `unblocked` | `bin`<br>`sbin`<br>`u`<br>`unblocked` |
| Sequential formatted | `text` | `blocked`<br>`cos`<br>`sbin/text` |
| Direct formatted | `text` | `sbin/text` |

#### 14.2.2.1 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 that is 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 reposition a file with unblocked file structure with a BACKSPACE statement. You cannot reposition the file to a previous record when record boundaries do not exist.

BUFFER IN and BUFFER OUT statements can specify a file that has 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.

You can specify the unblocked data file structure by using the assign(1) command in several ways. All methods 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 declared or assumed for a Fortran file.

For more information about buffering, see Section 14.2.3, page 286.

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 UNICOS/mp and UNICOS/lc systems contain records that are the same length. The end-of-file record is recognized in sequential-access files.

### 14.2.2.2 assign -s sbin File Processing (not recommended)

You can use an assign -s sbin specification for a Fortran file that is 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.

**Note:** Cray discourages the use of `assign -s sbin` because of poor I/O performance. If you cannot use an FFIO layer, use `assign -s text` for formatted files and `assign -s unblocked` for unformatted files.

#### 14.2.2.3 `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.

#### 14.2.2.4 `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 that is 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.

#### 14.2.2.5 `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 will treat such an occurrence as a malformed record. This file structure can be specified for a file that is declared as formatted sequential access or formatted direct access. It is the default file structure for formatted sequential access files. It is also the default file structure for 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.

You cannot use `BUFFER IN` and `BUFFER OUT` statements with this structure. You can use a `BACKSPACE` statement to reposition a file with this structure.

#### 14.2.2.6 `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. You can 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.

You can specify this file structure with one of the following `assign`(1) 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 an EOR, or 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 $2^{24}$) |
| fwi | 55–63 | Forward index; the number of words to 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; $10_8$ for EOR, $16_8$ for EOF, and $17_8$ 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 $2^{20}$ to the block where the current file starts (as defined by the last EOF). |

| Field | Bits | Description |
|---|---|---|
| pri | 40–54 | Previous record index; offset modulo $2^{15}$ to the block where the current record starts. |
| fwi | 55–63 | Forward index; the number of words to next control word. |

### 14.2.3 Buffer Specifications

A buffer is a temporary storage location for data while the data is being transferred. A buffer is 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. This is another use of a cache.

- 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, you can increase the buffer size to a large value, using the following `assign` command:

  ```
  assign -b buffer-size u:buffer-count
  ```

- If file `foo` is a small file or is accessed infrequently, minimize the buffer size using the following `assign` command:

  ```
  assign -b 1 f:foo
  ```

#### 14.2.3.1 Default Buffer Sizes

The Fortran I/O library automatically selects default buffer sizes according to file access type as shown in Table 26. You can override the defaults by using the `assign`(1) command. The following subsections describe the default buffer sizes on various systems.

**Note:** One *block* is 4,096 bytes on UNICOS/mp and UNICOS/lc systems.

The default buffer sizes are as follows:

Table 26. 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:<br>• The record length in bytes + 1<br>• 16 blocks (65,536 bytes) |
| Direct unformatted | The larger of:<br>• The record length<br>• 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.

### 14.2.3.2 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`(1) 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.

#### 14.2.3.3 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`(1) 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 you plan to use `assign -s u` to bypass the system cache, all requests must be well formed.

#### 14.2.3.4 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`(1) 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.

### 14.2.4 Foreign File Format Specification

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
```

### 14.2.5 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.

### 14.2.6 Fortran 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`.

Figure 4 summarizes the available access methods and the default buffer sizes.

| | Blocked | | | Unblocked | | | |
|---|---|---|---|---|---|---|---|
| Access method<br>`assign` option | Blocked<br>`-F f77` | Blocked<br>`-s cos` | Text<br>`-s text` | Undef<br>`-s u` | Binary<br>`-s bin` | Unblocked<br>`-s unblocked` | Buffer size for default* |
| Formatted sequential I/O<br>`WRITE(9,20)`<br>`PRINT` | | Valid | Valid Default | | | | 16 |
| Formatted direct I/O<br>`WRITE(9,20,REC=)` | | | Valid Default | Valid | | Valid | min(recl+1, 8) bytes |
| Unformatted sequential I/O<br>`WRITE(9)` | Valid Default | Valid | | Valid | Valid | Valid | 128 |
| Unformatted direct I/O<br>`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

Figure 4. Access Methods and Default Buffer Sizes

## 14.3 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 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 you select local assign mode, the assign environment will be stored in memory. Thus, other processes can not adversely affect the assign environment used by the program.

The ASNCTL(3f) 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)
```

### Example 5: 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')
```

# Using FFIO [15]

This chapter provides an overview of the capabilities of the flexible file I/O (FFIO) system and describes how to use FFIO with common file structures to enhance code performance without changing source code.

Flexible file I/O, sometimes called layered I/O, is used to perform many I/O-related tasks. For details about each individual I/O layer, see Chapter 16, page 311.

## 15.1 Introduction to FFIO

The FFIO system is based on the concept that for all I/O a list of processing steps must be performed to transfer the user data between the user's memory and the desired I/O device. I/O can be the slowest part of a computational process, and the speed of I/O access methods varies depending on computational processes.

Figure 5 illustrates the typical flow of data from the user's variables to and from the I/O device.

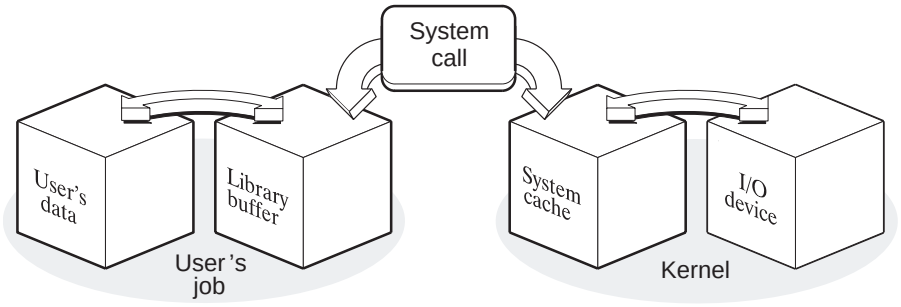


Figure 5. Typical Data Flow

It is useful to think of each of these boxes as a stopover for the data, and each transition between stopovers as a processing step. It is also important to realize that the actual I/O path can skip one or more steps in this process, depending on the I/O features used at a given point in a given program.

Each transition has benefits and costs. Different applications might use the total I/O system in different ways. For example, if I/O requests are large, the library buffer is unnecessary because the buffer is used primarily to avoid making system calls for every small request. You can achieve better I/O throughput with large I/O requests by not using library buffering.

If library buffering is not used, I/O requests should be large; otherwise, I/O performance will be degraded. On the other hand, if all I/O requests are small, the library buffer is essential to avoid making a costly system call for each I/O request.

It is useful to be able to modify the I/O process to prevent intermediate steps (such as buffering of data) for existing programs without requiring that the source code be changed. The `assign`(1) command lets you modify the total user I/O path by establishing an I/O environment.

The FFIO system lets you specify each stopover. You can specify a comma-separated list of one or more processing steps by using the `assign -F` command:

```
assign -F spec1,spec2,spec3...
```

Each *spec* in the list is a processing step that requests one I/O layer, or logical grouping of layers. The layer specifies the operations that are performed on the data as it is passed between the user and the I/O device. A *layer* refers to the specific type of processing being done. In some cases, the name corresponds directly to the name of one layer. In other cases, however, specifying one layer invokes the routines used to pass the data through multiple layers. See the `intro_ffio`(3f) man page for details about using the `-F` option to the `assign` command.

Processing steps are ordered as if the `-F` side (the left side) is the user and the system / device is the right side, as in the following example:

```
assign -F user,bufa,system
```

With this specification, a `WRITE` operation first performs the `user` operation on the data, then performs the `bufa` operation, and then sends the data to the system. In a `READ` operation, the process is performed from right to left. The data moves from the system to the user. The layers closest to the user are *higher-level layers*; those closer to the system are *lower-level layers*.

The FFIO system has an internal model of the world of data, which it maps to any given actual logical file type. Four of these concepts are basic to understanding the inner workings of the layers.

| Concept | Definition |
|---|---|
| Data | Data is a stream of bits. |
| Record marks | End-of-record (EOR) marks are boundaries between logical records. |
| File marks | End-of-file (EOF) marks are special types of record marks that exist in some file formats. |
| End-of-data (EOD) | An end-of-data (EOD) is a point immediately beyond the last data bit, EOR, or EOF in the file. |

All files are streams of 0 or more bits that may contain record and/or file marks.

Individual layers have varying rules about which of these things can appear and in which order they can appear in a file.

Fortran programmers and C programmers can use the capabilities described in this document. Fortran users can use the `assign`(1) command to specify these FFIO options. For C users, the FFIO layers are available only to programs that call the FFIO routines directly (`ffopen`(3c), `ffread`(3c), and `ffwrite`(3c)).

You can use FFIO with the Fortran I/O forms listed in the following table. For each form, the equivalent `assign` command is shown.

| **Fortran I/O Form** | **Equivalent `assign` Command** |
|---|---|
| Buffer I/O | `assign -F f77` |
| Unformatted sequential | `assign -F f77` |
| Unformatted direct access | `assign -F cache` |
| Formatted sequential | `assign -F text` |
| Namelist | `assign -F text` |
| List-directed | `assign -F text` |

## 15.2 Using Layered I/O

The specification list on the `assign -F` command comprises all of the processing steps that the I/O system performs. If `assign -F` is specified, any default processing is overridden. For example, unformatted sequential I/O is assigned a default structure of `f77` on UNICOS/mp and UNICOS/lc systems. The `-F f77` option provides the same structure. The FFIO system provides detailed control over I/O processing requests. However, to effectively use the `f77` option (or any FFIO option), you must understand the I/O processing details.

As a very simple example, suppose you were making large I/O requests and did not require buffering or blocking on your data. You could specify:

```
assign -F system
```

The `system` layer is a generic system interface that chooses an appropriate layer for your file. If the file is on a disk, it chooses the `syscall` layer, which maps each user I/O request directly to the corresponding system call. A Fortran `READ` statement is mapped to one or more `read`(2) system calls and a Fortran `WRITE` statement to one or more `write`(2) system calls.

If you want your file to be F77 blocked (the default blocking for Fortran unformatted I/O on UNICOS/mp and UNICOS/lc systems), you can specify:

```
assign -F f77
```

If you want your file to be COS blocked, you can specify:

```
assign -F cos
```

> **Note:** In all `assign -F` specifications, the `system` layer is the implied last layer. The above example is functionally identical to `assign -F cos,system`.

These two *spec*s request that each `WRITE` request first be blocked (blocking adds control words to the data in the file to delimit records). The `f77` layer then sends the blocked data to the `system` layer. The `system` layer passes the data to the device.

The process is reversed for `READ` requests. The `system` layer retrieves blocked data from the file. The blocked data is passed to the next higher layer, the `f77` layer, where it is deblocked. The deblocked data is then presented to the user.

### 15.2.1 I/O Layers

Several different layers are available for the *spec* argument. Each layer invokes one or more layers, which then handles the data it is given in an appropriate manner. For example, the `syscall` layer essentially passes each request to an appropriate system call. The `mr` layer tries to hold an entire file in a buffer that can change size as the size of the file changes; it also limits actual I/O to lower layers so that I/O occurs only at open, close, and overflow.

Table 27 defines the classes you can specify for the *spec* argument to the `assign -F` option. For detailed information about each layer, see Chapter 16, page 311.

Table 27. FFIO Layers

| Layer | Function |
|---|---|
| `bufa` | Asynchronous buffering layer |
| `cache` | Memory-cached I/O |
| `cachea` | Asynchronous memory-cached I/O |
| `cos` or `blocked` | COS blocking. This is the default for Fortran sequential unformatted I/O on UNICOS and UNICOS/mk systems. |
| `event` | I/O monitoring layer |
| `f77` | FORTRAN 77/UNIX Fortran record blocking. This is the default for Fortran sequential unformatted I/O on UNICOS/mp and UNICOS/lc systems and the common blocking format used by most FORTRAN 77 compilers on UNIX systems. |
| `fd` | File descriptor open |
| `global` | Distributed cache layer for MPI, SHMEM, OpenMP, and Co-array Fortran |
| `ibm` | IBM file formats |
| `mr` | Memory-resident file handlers |
| `null` | Syntactic convenience for users (does nothing) |
| `site` | User-defined site-specific layer |
| `syscall` | System call I/O |

| Layer | Function |
|---|---|
| system | Generic system interface |
| text | Newline separated record formats |
| user | User-defined layer |
| vms | VAX/VMS file formats |

### 15.2.2 Layered I/O Options

You can modify the behavior of each I/O layer. The following *spec* format shows how you can specify a *class* and one or more *opt* and *num* fields:

*class.opt1.opt2:num1:num2:num3*

For *class*, you can specify one of the layers listed in Table 27. Each layer has a different set of options and numeric parameter fields that can be specified. This is necessary because each layer performs different duties. The following rules apply to the *spec* argument:

- The *class* and *opt* fields are case-insensitive. For example, the following two *spec*s are identical:

  ```
  Ibm.VBs:100:200
  IBM.vbS:100:200
  ```

- The *opt* and *num* fields are usually optional, but sufficient separators must be specified as placeholders to eliminate ambiguity. For example, the following *spec*s are identical:

  ```
  cos..::40, cos.::40
  cos::40
  ```

  In this example, *opt1*, *opt2*, *num1*, and *num2* can assume default values.

- To specify more than one *spec*, use commas between *specs*. Within each *spec*, you can specify more than one *opt* and *num*. Use periods between *opt* fields, and use colons between *num* fields.

The following options all have the same effect. They all specify the vms layer and set the initial allocation to 100 blocks:

```
-F vms:100
-F vms.:100
-F vms..:100
```

The following option contains one *spec* for an `vms` layer that has an *opt* field of `scr` (which requests scratch file behavior):

```
-F vms.scr
```

The following option requests two *class*es with no *opt*s:

```
-F f77,vms
```

The following option contains two *spec*s and requests two layers: `cos` and `vms`. The `cos` layer has no options; the `vms` layer has options `scr` and `ovfl`, which specify that the file is a scratch file that is allowed to overflow and that the maximum allocation is 1000 sectors:

```
-F cos,vms.scr.ovfl::1000
```

When possible, the default settings of the layers are set so that optional fields are seldom needed.

## 15.3 FFIO and Common Formats

This section describes the use of FFIO with common file structures and the correlation between the common or default file structures and the FFIO usage that handles them.

### 15.3.1 Reading and Writing Text Files

Use the `fdcp` command to copy files while converting record blocking.

Most human-readable files are in *text format*; this format contains records comprised of ASCII characters with each record terminated by an ASCII line-feed character, which is the newline character in UNIX terminology. The FFIO specification that selects this file structure is `assign -F text`.

The FFIO package is seldom required to handle text files. In the following types of cases, however, using FFIO may be necessary:

- Optimizing text file access to reduce I/O wait time
- Handling multiple EOF records in text files
- Converting data files to and from other formats

I/O speed is important when optimizing text file access. Using `assign -F text` is expensive in terms of processor time, but it lets you use memory-resident files, which can reduce or eliminate I/O wait time.

The FFIO system also can process text files that have embedded EOF records. The `~e` string alone in a text record is used as an EOF record. Editors such as `sed`(1) and other standard utilities can process these files, but it is sometimes easier with the FFIO system.

The `text` layer is also useful in conjunction with the `fdcp`(1) command. The `text` layer provides a standard output format. Many forms of data that are not considered foreign are sometimes encountered in a heterogeneous computing environment. If a record format can be described with an FFIO specification, it can usually be converted to text format by using the following script:

```
OTHERSPEC=$1
INFILE=$2
OUTFILE=$3
assign -F ${OTHERSPEC} ${INFILE}
assign -F text ${OUTFILE}
fdcp ${INFILE} ${OUTFILE}
```

If the name of the script is `to.text`, you can invoke it as follows:

```
% to.text cos data_cos data_text
```

### 15.3.2 Reading and Writing Unblocked Files

The simplest data file format is the binary stream or *unblocked data*. It contains no record marks, file marks, or control words. This is usually the fastest way to move large amounts of data, because it involves a minimal amount of processor and system overhead.

The FFIO package provides several layers designed specifically to handle a binary stream of data. These layers are `syscall`, `mr`, `bufa`, `cache`, `cachea`, and `global`. These layers behave the same from the user's perspective; they only use different system resources. The unblocked binary stream is usually used for unformatted data transfer. It is not usually useful for text files or when record boundaries or backspace operations are required. The complete burden is placed on the application to know the format of the file and the structure and type of the data contained in it.

This lack of structure also allows flexibility; for example, a file declared with one of these layers can be manipulated as a direct-access file with any record length.

In this context, `fdcp` can be called to do the equivalent of the `cp`(1) command only if the input file is a binary stream and to remove blocking information only if the output file is a binary stream.

### 15.3.3 Reading and Writing Fixed-length Records

The most common use for fixed-length record files is for Fortran direct access. Both unformatted and formatted direct-access files use a form of fixed-length records. The simplest way to handle these files with the FFIO system is with binary stream layers, such as `system`, `syscall`, `cache`, `cachea`, `global`, and `mr`. These layers allow any requested pattern of access and also work with direct-access files. The `syscall` and `system` layers, however, are unbuffered and do not give optimal performance for small records.

The FFIO system also directly supports some fixed-length record formats.

### 15.3.4 Reading and Writing Blocked Files

The `f77` blocking format is the default file structure for all Fortran sequential unformatted files. The `f77` layer is provided to handle these files.

The `f77` layer is the default file structure on Cray X1 and X2 systems. If you specify another layer, such as `mr`, you may have to specify a `f77` layer to get `f77` blocking.

## 15.4 Enhancing Performance

FFIO can be used to enhance performance in a program without changing or recompiling the source code. This section describes some basic techniques used to optimize I/O performance. Additional optimization options are discussed in Chapter 16, page 311.

### 15.4.1 Buffer Size Considerations

In the FFIO system, buffering is the responsibility of the individual layers; therefore, you must understand the individual layers in order to control the use and size of buffers.

The `cos` layer has high payoff potential to the user who wants to extract top performance by manipulating buffer sizes. As the following example shows, the `cos` layer accepts a buffer size as the first numeric parameter:

```
assign -F cos:42 u:1
```

If the buffer is sufficiently large, the `cos` layer also lets you keep an entire file in the buffer and avoid almost all I/O operations.

### 15.4.2 Removing Blocking

I/O optimization usually consists of reducing overhead. One part of the overhead in doing I/O is the processor time spent in record blocking. For many files in many programs, this blocking is unnecessary. If this is the case, the FFIO system can be used to deselect record blocking and thus obtain appropriate performance advantages.

The following layers offer unblocked data transfer:

| Layer | Definition |
|---|---|
| `syscall` | System call I/O |
| `bufa` | Buffering layer |
| `cachea` | Asynchronous cache layer |
| `cache` | Memory-resident buffer cache |
| `global` | SHMEM and MPI cache layer |
| `mr` | Memory-resident (MR) I/O |

You can use any of these layers alone for any file that does not require the existence of record boundaries. This includes any applications that are written in C that require a byte stream file.

#### 15.4.2.1 The `syscall` Layer

The `syscall` layer offers a simple, direct system interface with a minimum of system and library overhead. If requests are larger than approximately 64 K, this method can be appropriate.

#### 15.4.2.2 The `bufa` and `cachea` Layers

The `bufa` and `cachea` layers permit efficient file processing. Both layers provide asynchronous buffering managed by the library, and the `cachea` layer allows recently accessed parts of a file to be cached in memory.

The number of buffers and the size of each buffer are tunable. In the `bufa`*:bs:nbufs* or `cachea`*:bs:nbufs* FFIO specifications, the *bs* argument specifies the size in 4096-byte blocks of each buffer. The default depends on the `st_blksize` field returned from a `stat`(2) system call of the file; if this return value is 0, the default is 8 for all files. The *nbufs* argument specifies the number of buffers to use. `bufa` defaults to 2 buffers, while `cachea` defaults to 512 buffers.

#### 15.4.2.3 The mr Layer

The mr layer lets you use main memory as an I/O device for many files. Used in combination with the other layers, cos blocked files, text files, and direct-access files can all reside in memory without recoding. This can result in excellent performance for a file, or part of a file, that can reside in memory.

The mr layer features both scr and save mode, and it directs overflow to the next lower layer automatically.

The assign -F command specifies the entire set of processing steps that are performed when I/O is requested. If a file is blocked, you must specify the appropriate layer for the handling of block and record control words as in the following examples:

```
assign -F f77,mr u:1
assign -F cos,mr fort.1
```

Section 15.5, page 307 contains several mr program examples.

#### 15.4.2.4 The global Layer (Deferred Implementation)

The global layer is a caching layer that distributes data across all multiple SHMEM or MPI processes. Open and close operations require participation by all processes that access the file; all other operations are performed independently by one or more processes. File positions can be private to a process or global to all processes.

You can specify both the cache size and the number of cache pages to use. Since this layer is used by parallel processes, the actual number of cache pages used is the number specified times the number of processes.

#### 15.4.2.5 The cache Layer

The cache layer permits efficient file processing for repeated access to one or more regions of a file. It is a library-managed buffer cache that contains a tunable number of pages of tunable size.

To specify the cache layer, use the following option:

```
assign -F cache[:[bs][:[nbufs]]]
```

The *bs* argument specifies the size in 4096-byte blocks of each cache page; the default is 16. The *nbufs* argument specifies the number of cache pages to use. The default is 4. You can achieve improved I/O performance by using one or more of the following strategies:

- Use a cache page size that is a multiple of the user's record size. This ensures that no user record straddles two cache pages. If this is not possible or desirable, it is best to allocate a few additional cache pages (*nbufs*).
- Use a number of cache pages that is greater than or equal to the number of file regions the code accesses at one time.

If the number of regions accessed within a file is known, the number of cache pages can be chosen first. To determine the cache page size, divide the amount of memory to be used by the number of cache pages. For example, suppose a program uses direct access to read 10 vectors from a file and then writes the sum to a different file:

```
integer VECTSIZE, NUMCHUNKS, CHUNKSIZE
parameter(VECTSIZE=1000*512)
parameter(NUMCHUNKS=100)
parameter(CHUNKSIZE=VECTSIZE/NUMCHUNKS)
read a(CHUNKSIZE), sum(CHUNKSIZE)
open(11,access='direct',recl=CHUNKSIZE*8)
call asnunit (2,'-s unblocked',ier)
open (2,form='unformatted')
do i = 1,NUMCHUNKS
  sum = 0.0
  do j = 1,10
    read(11,rec=(j-1)*NUMCHUNKS+i)a
    sum=sum+a
  enddo
  write(2) sum
enddo
end
```

If 4 MB of memory are allocated for buffers for unit 11, 10 cache pages should be used, each of the following size:

```
4MB/10 = 400000 bytes = 97 blocks
```

Make the buffer size an even multiple of the record length of 409600 bytes by rounding it up to 100 blocks (= 409600 bytes), then use the following `assign` command:

```
assign -F cache:100:10 u:11
```

## 15.5 Sample Programs

The following examples illustrate the use of the mr layers.

### Example 6: Unformatted direct mr with unblocked file

In the following example, batch job ex8 contains a program that uses unformatted direct-access I/O with an mr layer:

```
#QSUB -r ex8 -lT 10 -lQ 500000
#QSUB -eo -o ex8.out
date
set -x
cd $TMPDIR
cat > ex8.f <<EOF
      program example8
      dimension r(512)
      data r/512*2.0/
      open(1,form='unformatted',access='direct',recl=4096)
      do 100 i=1,100
        write(1,rec=i,iostat=ier)r
        if(ier.ne.0)then
           if(ier.eq.5034)then
              print *,"overflow to disk at record=",i
           else
              print *,"error on write=",ier
           end if
        end if
100   continue
      do 200 i=100,1,-1
        read(1,rec=i,iostat=ier)r
        if(ier.ne.0)then
           print *,"error on read=",ier
        end if
200   continue
      close(1)
      end
EOF
ftn  ex8.f -o ex8           # compile and compile
assign -R                   # reset assign parameters
assign -F mr.scr.ovfl::50: fort.1
                            # assign file fort.1 to be mr with a
                            # 50 block limit
./ex8                       # execute
```

The program writes the first 50 blocks of `fort.1` to the memory-resident layer. The next 50 blocks overflow the `mr` buffer and will be written to a disk. Because the `scr` option is specified, the file is not saved when `fort.1` is closed.

### Example 7: Unformatted sequential `mr` with blocked file

The following program uses an `mr` layer with unformatted sequential I/O:

```
     program example4a
     integer r(512)
     data r/512*1.0/
C    Reset assign environment, then assign file without FFIO
C    to be read back in by subsequent program.
     call assign('assign -R',ier1)
     call assign('assign -a /tmp/file1 -s unblocked f:fort.1',ier2)
     if(ier1.ne.0.or.ier2.ne.0)then
       print *,"assign error"
       goto200
     end if
     open(1,form='unformatted')
C    write out 100 records to disk file: /tmp/file1
     do 100 k=1,100
       write(1)r
100  continue
     close(1)
200  continue
     end
```

In the program unit `example4b` that follows, the `assign` command arguments contain the following options to use blocked file structure:

```
assign -R
assign -a /tmp/file1 -F f77,mr.save.ovfl u:3
```

`example4b` writes an unblocked file disk file, `/tmp/file1`. If you want to use a blocked file structure, the `assign` command arguments should contain the following instructions in program unit `example4a`:

```
assign -R
assign -a /tmp/file1 f:fort.1
```

```
      program example4b
      integer r(512)
C     Reset assign environment, then assign file
C     with an mr layer.
```

```
      call assign('assign -R',ier1)
      call assign('assign -a /tmp/file1
     &         -F mr.save.ovfl u:3',ier2)
      if(ier1.ne.0.or.ier2.ne.0)then
        print *,"assign error"
        goto300
      end if
C     open the previously written file '/tmp/file1',
C     load it into memory
      open(3,form='unformatted')
C     read 5 records
      do 200 k=1,5
        read(3)r1
200   continue
       rewind(3)
      close(3)
300   continue
      end
```

A sequential formatted file must always have a `text` specification before the residency layer specification so that the I/O library can determine the end of a record.

# FFIO Layer Reference [16]

This chapter provides details about each of the following FFIO layers:

| Layer | Definition |
|---|---|
| `bufa` | Library-managed asynchronous buffering |
| `cache` | Memory-cached layer |
| `cachea` | Asynchronous memory-cached layer |
| `cos` or `blocked` | COS blocking layer |
| `event` | I/O monitoring layer |
| `f77` | Common UNIX Fortran record blocking |
| `fd` | File descriptor open layer |
| `global` | Distributed I/O for MPI, SHMEM, OpenMP, and Co-array Fortran programs |
| `ibm` | IBM file formats |
| `mr` | Memory-resident file handlers |
| `null` | Syntactic convenience for users |
| `site` | User-defined site-specific layer |
| `syscall` | System call I/O |
| `system` | Generic system layer |
| `text` | Newline-separated record formats |
| `user` | User-defined layer |
| `vms` | VAX/VMS file formats |

Section 16.1 describes how to interpret the information presented in the remaining sections of this chapter. See the `intro_ffio`(3) man page for more details about FFIO layers.

## 16.1 Characteristics of Layers

In the descriptions of the layers that follow, the Data Manipulation tables use the following categories of characteristics:

| Characteristic | Description |
|---|---|
| Granularity | Indicates the smallest amount of data that the layer can handle. As of the Programming Environment 5.2 release, all layers use 8-bit (1-byte) granularity. |
| Data model | Indicates the data model. Three main data models are discussed in this section. The first type is the Record model, which has data with record boundaries and may have an end-of-file (EOF).<br><br>The second type is Stream (a stream of bits). None of these support the EOF.<br><br>The third type is the Filter, which does not have a data model of its own but derives it from the lower-level layers. Filters usually perform a data transformation (such as blank compression or expansion). |
| Truncate on write | Indicates whether the layer forces an implied EOD on every write operation (EOD implies truncation). |
| Implementation strategy | Describes the internal routines that are used to implement the layer.<br><br>The X-records type under Implementation Strategy (if used in the tables) refers to a record type in which the length of the record is prepended and appended to the record. For `f77` files, the record length is contained in 4 bytes at the beginning and the end of a record. |

In the descriptions of the layers, the Supported Operations tables use the following categories:

Operation

Lists the operations that apply to that particular layer. The following is a list of supported operations:

| | |
|---|---|
| `ffopen` | `ffclose` |
| `ffread` | `ffflush` |
| `ffreadc` | `ffweof` |
| `ffwrite` | `ffweod` |
| `ffwritec` | `ffseek` |
| `ffbksp` | |

Support — Uses three potential values: Yes, No, or Passed through. Passed through indicates that the layer does not directly support the operation but relies on the lower-level layers to support it.

Used — Lists two values: Yes or No. Yes indicates that the operation is required of the next lower-level layer. No indicates that the operation is never required of the lower-level layer. Some operations are not directly required but are passed through to the lower-layer if requested of this layer. These are noted in the comments.

Comments — Describes the function or support of the layer's function.

On many layers, you can also specify the numeric parameters by using a keyword.

## 16.2 The `bufa` Layer

The `bufa` layer provides library-managed asynchronous buffering. It is optimized to perform sequential I/O using adaptive I/O techniques, meaning the `bufa` layer transforms `READ` and `WRITE` requests into read-ahead and write-behind requests. This can minimize I/O wait time and reduce the number of low-level I/O requests for some files.

The syntax is as follows:

`bufa:[`*`num1`*`]:[`*`num2`*`]`

The keyword syntax is as follows:

`bufa[.bufsize=`*`num1`*`][.num_buffers=`*`num2`*`]`

The *num1* argument specifies the size, in 4096-byte blocks, of each buffer. The default buffer size depends on the device on which your file is located. The maximum allowed value on UNICOS/mp and UNICOS/lc systems 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

The *num2* argument specifies the number of buffers to be used. The default is 2.

Table 28. Data Manipulation: `bufa` Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bits | Stream | No |

Table 29. Supported Operations: `bufa` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | Yes | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | Yes | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | Yes | |
| `ffweof` | Passed through | | Yes | Only if explicitly requested |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | Only if supported by the underlying layer | Yes | Only if explicitly requested |
| `ffbksp` | No | | NA | |

## 16.3 The cache Layer

The cache layer improves nonsequential I/O by dividing files into cache page-sized sections, then keeping whichever pages are currently being accessed in main memory. This can significantly improve data reuse, with appropriately configured buffers, and can also reduce the number of low-level I/O requests for random access.

When used as the last layer above the system or syscall layer, the cache layer supports the assign -B option to enable or disable direct I/O.

This layer also offers efficient sequential access when a buffered, unblocked file is needed. The syntax is as follows:

cache[.*type*]:[*num1*]:[*num2*][*num3*]

The keyword syntax is as follows:

cache[.*type*][.page_size=*num1*][.num_pages=*num2*
[.bypass_size=*num3*]]

The *type* argument can be mem. mem directs that cache pages reside in main memory. *num1* specifies the size, in 4096–byte blocks, of each cache page buffer. The default is 16. The maximum allowed value is 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

*num2* specifies the number of cache pages. The default is 4. *num3* is the size in 4096-byte blocks at which the cache layer attempts to bypass cache layer buffering. If a user's I/O request is larger than *num3*, the request might not be copied to a cache page. The default is *num3=num1×num2*.

When a cache page must be preempted to allocate a page to the currently accessed part of a file, the least recently accessed page is chosen for preemption. Every access stores a time stamp with the accessed page so that the least recently accessed page can be found at any time.

Table 30. Data Manipulation: cache Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bit | Stream | No |
| 512 words | Stream | No |

Table 31. Supported Operations: cache Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| ffopen | Yes | | Yes | |
| ffread | Yes | | No | |
| ffreadc | Yes | | No | |
| ffwrite | Yes | | No | |
| ffwritec | Yes | | No | |
| ffclose | Yes | | Yes | |
| ffflush | Yes | | No | |
| ffweof | No | | No | |
| ffweod | Yes | | Yes | |
| ffseek | Yes | | Yes | Requires underlying interface to be a stream |
| ffbksp | No | | NA | |

## 16.4 The cachea Layer

The cachea layer is similar to the cache layer in that it improves data reuse and nonsequential I/O by dividing files into cache page-sized sections, then keeping whichever pages are currently being accessed in main memory. However, like the bufa layer, it also applies adaptive I/O techniques, transforming READ and WRITE operations into read-ahead and write-behinds. Furthermore, unlike the bufa layer, there can be multiple threads (I/O chains) of read-aheads and write-behinds, depending on how the file is being accessed.

As a result, this layer can provide high write performance by asynchronously writing out selective cache pages. It can also provide high read performance by detecting sequential read access, both forward and backward. When sequential access is detected and when read-ahead is chosen, file page reads are anticipated and issued asynchronously in the direction of file access.

When used as the last layer above the system or syscall layer, the cachea layer supports the assign -B option to enable or disable direct I/O.

The syntax is as follows:

cachea[*type*]:[*num1*]:[*num2*]:[*num3*]

The keyword syntax is as follows:

cachea[*type*][.page_size=*num1*][.num_pages=*num2*] [.max_lead=*num3*]

*type* Directs that cache pages reside in memory (mem).

*num1* Specifies the size, in 4096-byte blocks, of each cache page buffer. Default is 512. The maximum allowed value is 1,073,741,823. You may not, however, be able to use a value this large because this much memory may not be available.

*num2* Specifies the number of cache pages to be used. Default is 8.

*num3* Specifies the number of cache pages to asynchronously read ahead when sequential read access patterns are detected. The default is either (*num-2* - 1) or 8, whichever is smaller.

Table 32. Data Manipulation: cachea Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bit | Stream (mimics UNICOS/mp system calls) | No |

Table 33. Supported Operations: cachea Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| ffopen | Yes | | Yes | |
| ffread | Yes | | No | |
| ffreadc | Yes | | No | |
| ffwrite | Yes | | No | |
| ffwritec | Yes | | No | |
| ffclose | Yes | | Yes | |
| ffflush | Yes | | No | |
| ffweof | No | | No | |

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | | Yes | Requires that the underlying interface be a stream |
| `ffbksp` | No | | NA | |

## 16.5 The cos Blocked Layer

The `cos` layer performs COS blocking and deblocking on a stream of data. The general format of the `cos` specification follows:

`cos:[.`*`type`*`][.`*`num1`*`]`

The keyword syntax is as follows:

`cos[.`*`type`*`][.bufsize=`*`num1`*`]`

The *num1* argument specifies the working buffer size in 4096-byte blocks.

If not specified, the default buffer size is the larger of the following: the large I/O size, the preferred I/O block size (see the `stat`(2) man page for details), or 48 blocks. See the `intro_ffio`(3F) man page for more details.

When writing, full buffers are written in full record mode. Reads are always performed in partial read mode; therefore, you do not have to know the block size to read it (if the block size is larger than the buffer, partial mode reads ensure that no parts of blocks are skipped).

Table 34. Data Manipulation: `cos` Layer

| Granularity | Data model | Truncate on write | Implementation strategy |
|---|---|---|---|
| 8 bit | Records with multi-EOF capability | Yes | `cos` specific |

Table 35. Supported Operations: cos Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| ffopen | Yes | | Yes | |
| ffread | Yes | | Yes | |
| ffreadc | Yes | | No | |
| ffwrite | Yes | | Yes | |
| ffwritec | Yes | | No | |
| ffclose | Yes | | Yes | |
| ffflush | Yes | No-op | Yes | |
| ffweof | Yes | | No | |
| ffweod | Yes | | Yes | Truncation occurs only on close |
| ffseek | Yes | Minimal support (see following note) | Yes | |
| ffbksp | Yes | No records | No | |

**Note:** seek operations are supported only to allow for rewind (seek(fd,0,0)) and seek-to-end (seek(fd,0,2)).

## 16.6 The event Layer

The event layer enables you to monitor, on a per-file basis, the I/O activity that occurs in I/O layer immediately preceding it. It generates statistics as a text log file and reports information such as the number of times an event was called, the event wait time, the number of bytes requested, and so on. You can request the following types of statistics:

- A list of all event types
- Event types that occur at least once
- A single line summary of activities that shows information such as amount of data transferred and the data transfer rate.

Statistics are reported to `stderr` by default. The `FFIO_EVENT_LOGFILE` environment variable can be used to name a file to which statistics are written by the `event` layer. The default action is to overwrite the existing statistics file if it exists. You can append reports to the existing file by specifying a plus sign (+) before the file name, as in this example:

```
setenv FFIO_EVENT_LOGFILE +saveIO
```

This layer report counts all I/O (`read`, `write`, etc.) and I/O-related (`open`, `close`, `fcntl`, etc.) requests. These counts represent the number of calls made by the parent layer above the `event` layer to the child layer below it. (The terms "above" and "below" are somewhat arbitrary, with the "higher" layers being closer to the program or application and the "lower" layers being closer to the operating system.) Both the numbers and types of requests can change as you move down the FFIO chain, as FFIO layers will consolidate multiple I/O requests into fewer requests and convert requests from one type to another (i.e., from synchronous to asynchronous).

The `event` layer is enabled by default and is included in the executable file; you do not have to relink to study the I/O performance of your program. To obtain event statistics, rerun your program with the `event` layer specified on the `assign` command, as in this example:

```
assign -F bufa,cachea,event,system
```

In the above example, the log file will show the I/O activity in the `cachea` layer.

The syntax for the `event` layer is as follows:

`event[.`*`type`*`]`

There is no alternate keyword specification for this layer.

The *type* argument selects the level of performance information to be written to the log file; it can have one of the following values:

| Value | Definition |
|---|---|
| `nostat` | No information is reported. |
| `brief` | Generates a report on the amount of data transferred through the `event` layer. |
| `summary` (default) | Generates three reports:<br>• The `brief` report.<br>• A report on file information, including the file size.<br>• A list of all the I/O and I/O-related requests that passed through the `event` layer. |

## 16.7 The `f77` Layer

The `f77` layer handles blocking and deblocking of the `f77` record type, which is common to most UNIX Fortran implementations, for sequential unformatted files. The syntax for this layer is as follows:

`f77[.`*`type`*`]:[`*`num1`*`]:[`*`num2`*`]`

The keyword syntax is as follows:

`f77[.`*`type`*`][.recsize=`*`num1`*`][.bufsize=`*`num2`*`]`

*type* Specifies the record type and can take one of two values:

| | |
|---|---|
| `nonvax` | Control words in a format common to computers such as the MC68000 (big-endian); default. |
| `vax` | VAX format (byte-swapped) control words. |

The specification of `vax` or `nonvax` is not relevant to data conversion.

*num1* Maximum record size, in bytes. The default is 2 MB. The maximum value that can be specified is 4 MB.

*num2* Buffer size, in bytes. The default is 65 KB.

To achieve maximum performance, ensure that the working buffer size is large enough to hold any records that are written plus the control words (control words consist of two 4-byte fields; one at the beginning of the record and one at the end of the record). If a record plus control words are larger than the buffer, the layer must perform some inefficient operations to do the write. If the buffer is large enough, these operations can be avoided.

On reads, the buffer size is not as important, although larger sizes will usually perform better.

Table 36. Data Manipulation: `f77` Layer

| Granularity | Data model | Truncate on write | Implementation strategy |
|---|---|---|---|
| 8 bits | Record | Yes | x records |

Table 37. Supported Operations: `f77` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | Yes | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | Yes | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | No | |
| `ffweof` | Passed through | | Yes | Only if explicitly requested |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | `ffseek(fd,0,0)` equals rewind; `ffseek(fd,0,2)` seeks to end | Yes | |
| `ffbksp` | Yes | Only in lower-level layer | No | |

## 16.8 The fd Layer

The fd layer allows connection of a FFIO file to a system file descriptor. You must specify the fd layer, as follows:

fd:[*num1*]

The keyword specification is as follows:

fd[.file_descriptor=*num1*]

The *num1* argument must be a system file descriptor for an open file. The ffopen or ffopens request opens a FFIO file descriptor that is connected to the specified file descriptor. The file connection does not affect the file whose name is passed to ffopen.

When used as the last layer above the system or syscall layer, the fd layer supports the assign -B option to enable or disable direct I/O.

All other properties of this layer are the same as the system layer. See Section 16.14, page 332 for details.

## 16.9 The global Layer (Deferred Implementation)

The global layer is a caching layer that distributes data across all multiple SHMEM, MPI, OpenMP, or Co-array Fortran processes. Open and close operations require participation by all processes that access the file; all other operations are independently performed by one or more processes.

The syntax for this layer is as follows:

global[. *type*]:[*num1*]:[*num2*]

The keyword syntax is as follows:

global[. *type*][.page_size=*num1*][.num_pages=*num2*]

The *type* argument can be privpos (default), in which the file position is private to a process, or (deferred implementation) globpos, in which the file position is global to all processes.

The *num1* argument specifies the size in 4096-byte blocks of each cache page. The default is 16.*num2* specifies the number of cache pages to be used on each process. The default is 4. If there are *n* processes, then $n \times num2$ cache pages are used.

*num2* buffer pages are allocated on every process that shares access to a global file. File pages are direct-mapped onto processes such that page *n* of the file will always be cached on process (*n mod NPES*), where *NPES* is the total number of processes sharing access to the global file. Once the process is identified where caching of the file page will occur, a least-recently-used method is used to assign the file page to a cache page within the caching process.

Table 38. Data Manipulation: `global` Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bits | Stream | No |

Table 39. Supported Operations: `global` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | No | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | No | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | No | |
| `ffweof` | No | | No | |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | | Yes | Requires underlying interface to be a stream |
| `ffbksp` | No | | NA | |

## 16.10 The `ibm` Layer

The `ibm` layer handles record blocking for seven common record types on IBM operating systems. The general format of the specification follows:

`ibm.[`*`type`*`]:[`*`num1`*`]:[`*`num2`*`]`

The keyword syntax is as follows:

`ibm[.`*`type`*`][.recsize=`*`num1`*`][.mbs=`*`num2`*`]`

The supported *type* values are as follows:

| Value | Definition |
|---|---|
| `u` | IBM undefined record type |
| `f` | IBM fixed-length records |
| `fb` | IBM fixed-length blocked records |
| `v` | IBM variable-length records |
| `vb` | IBM variable-length blocked records |
| `vbs` | IBM variable-length blocked spanned records |

The `f` format is fixed-length record format. For fixed-length records, *num1* is the fixed record length (in bytes) for each logical record. Exactly one record is placed in each block.

The `fb` format records are the same as `f` format records except that you can place more than one record in each block. *num1* is the length of each logical record. *num2* must be an exact multiple of *num1*.

The `v` format records are variable-length records. *recsize* is the maximum number of bytes in a logical record. *num2* must exceed *num1* by at least 8 bytes. Exactly one logical record is placed in each block.

The `vb` format records are variable-length blocked records. This means that you can place more than one logical record in a block. *num1* and *num2* are the same as with `v` format.

The `vbs` format records have no limit on record size. Records are broken into segments, which are placed into one or more blocks. *num1* should not be specified. When reading, *num2* must be at least large enough to accommodate the largest physical block expected to be encountered.

The *num1* field is the maximum record size that may be read or written. The `vbs` record type ignores it.

The *num2* (maximum block size) field is the maximum block size that the layer uses on reads or writes.

Table 40. Values for Maximum Record Size on `ibm` Layer

| Field | Minimum | Maximum | Default | Comments |
|---|---|---|---|---|
| `u` | 1 | 32,760 | 32,760 | |
| `f` | 1 | 32,760 | None | Required |
| `fb` | 1 | 32,760 | None | Required |
| `v` | 5 | 32,756 | 32,752 | Default is *num2*, 8 if not specified. |
| `vb` | 5 | 32,756 | 32,752 | Default is *num2*, 8 if not specified. |
| `vbs` | 1 | None | None | No maximum record size |

Table 41. Values for Maximum Block Size in `ibm` Layer

| Field | Minimum | Maximum | Default | Comments |
|---|---|---|---|---|
| `u` | 1 | 32,760 | 32,760 | Should be equal to *num1* |
| `f` | 1 | 32,760 | *num1* | Must be equal to *num1* |
| `fb` | 1 | 32,760 | *num1* | Must be multiple of *num1* |
| `v` | 9 | 32,760 | 32,760 | Must be >= *num1* + 8 |
| `vb` | 9 | 32,760 | 32,760 | Must be >= *num1* + 8 |
| `vbs` | 9 | 32,760 | 32,760 | |

Table 42. Data Manipulation: `ibm` Layer

| Granularity | Data model | Truncate on write | Implementation strategy |
|---|---|---|---|
| 8 bits | Record | No for `f` and `fb` records. Yes for `v`, `vb`, and `vbs` records. | `f` records for `f` and `fb`. `v` records for `u`, `v`, `vb`, and `vbs`. |

Table 43. Supported Operations: `ibm` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | Yes | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | Yes | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | No | |
| `ffweof` | Passed through | | Yes | |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | `seek(fd, 0, 0)` only (equals rewind) | Yes | `seek(fd,0,0)` only |
| `ffbksp` | No | | No | |

## 16.11 The `mr` Layer

The memory-resident (`mr`) layer lets users declare that all or part of a file will reside in memory. This can improve performance for relatively small files that are heavily accessed or for larger files where the first part of the file is heavily accessed (for example, a file which contains a frequently updated directory at the beginning.) The `mr` layer tries to allocate a buffer large enough to hold the entire file.

**Note:** It is generally more advantageous to configure the layer preceding the `mr` layer to make the file buffer-resident, assuming that layer can support buffers of sufficient size.

The options are as follows:

`mr[.`*`type`*`[.`*`subtype`*`]]:`*`num1`*`:`*`num2`*`:`*`num3`*

The keyword syntax is as follows:

```
mr[.type[.subtype]][.start_size=num1][.max_size=num2]
[.inc_size=num3]
```

The *type* field specifies whether the file in memory is intended to be saved or is considered a scratch file. This argument accepts the following values:

| Value | Definition |
|---|---|
| `save` | Default. The file is loaded into memory when opened and written back to the next lower layer when closed. The `save` option also modifies the behavior of overflow processing. |
| `scr` | Scratch file. The file is not read into memory when opened and not written when closed. |

The *subtype* field specifies the action to take when the data can no longer fit in the allowable memory space. It accepts the following values:

| Value | Definition |
|---|---|
| `ovfl` | Default. Data which does not fit (overflows) the maximum specified memory allocation is written to the next lower layer, which is typically a disk file. An informative message is written to `stderr` on the first overflow. |
| `ovflnomsg` | Identical to `ovfl`, except that no message is issued when the data overflows the memory-resident buffer. |
| `novfl` | If data does not fit in memory, then subsequent `write`(1) operations fail. |

The *num1*, *num2*, and *num3* fields are nonnegative integer values that state the number of 4096-byte blocks to use in the following circumstances:

| Field | Definition |
|---|---|
| *num1* | The initial size of the memory allocation, specified in 4,096-byte blocks. The default is 0. |
| *num2* | The maximum size of the memory allocation, specified in 4,096-byte blocks. The default is either *num1* or 256 blocks (1 MB), whichever is larger. |
| *num3* | Increment size of the memory allocation, specified in 4,096-byte blocks. This value is used when allocation additional memory space. The default is 256 blocks (1 MB) or (*num2-num1*), whichever is smaller. |

The *num1* and *num3* fields represent best-effort values. They are intended for tuning purposes and usually do not cause failure if they are not satisfied precisely as specified. For example, if the available memory space is only 100 blocks and the chosen *num3* value is 200 blocks, growth is allowed to use the 100 available blocks rather than failing to grow, because the full 200 blocks requested for the increment are unavailable.

**Caution:** When using the `mr` layer, you must ensure that the size of the memory-resident portions of the files are limited to reasonable values. Unrestrained and unmanaged growth of such file portions can cause heap fragmentation, exhaustion of all available memory, and program abort. If this growth has consumed all available memory, the program may not abort gracefully, making such a condition difficult to diagnose.

Large memory-resident files may reduce I/O performance for sites that provide memory scheduling that favors small processes over large processes. Check with your system administrator if I/O performance is diminished.

Increment sizes which are too small can also contribute to heap fragmentation.

Memory allocation is done by using the `malloc`(3c) and `realloc`(3c) library routines. The file space in memory is always allocated contiguously.

When allocating new chunks of memory space, the *num3* argument is used in conjunction with `realloc` as a minimum first try for reallocation.

Table 44. Data Manipulation: mr Layer

| Primary function | Granularity | Data model | Truncate on write |
|---|---|---|---|
| Avoid I/O to the extent possible, by holding the file in memory. | 8 bit | Stream (mimics UNICOS/mp system calls) | No |

Table 45. Supported Operations: mr Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| ffopen | Yes | | Yes | Sometimes delayed until overflow |
| ffread | Yes | | Yes | Only on open |
| ffreadc | Yes | | No | |
| ffwrite | Yes | | Yes | Only on close, overflow |
| ffwritec | Yes | | No | |
| ffclose | Yes | | Yes | |
| ffflush | Yes | No-op | No | |
| ffweof | No | No representation | No | No representation |
| ffweod | Yes | | Yes | |
| ffseek | Yes | Full support (absolute, relative, and from end) | Yes | Used in open and close processing |
| ffbksp | No | No records | No | |

## 16.12 The null Layer

The null layer is a syntactic convenience for users; it has no effect. This layer is commonly used to simplify the writing of a shell script when a shell variable is used to specify a FFIO layer specification. For example, the following line is from a shell script with a file using the assign command and overlying blocking is expected (as specified by BLKTYP):

```
assign -F $BLKTYP,cos fort.1
```

If BLKTYP is undefined, the illegal specification list ,cos results. The existence of the null layer lets the programmer set BLKTYP to null as a default, and simplify the script, as in:

```
assign -F null,cos fort.1
```

This is identical to the following command:

```
assign -F cos fort.1
```

When used as the last layer above the system or syscall layer, the null layer supports the assign -B option to enable or disable direct I/O.

## 16.13 The syscall Layer

The syscall layer directly maps each request to an appropriate system call. The layer does not accept any options.

Table 46. Data Manipulation: syscall Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bits (1 byte) | Stream (UNICOS/mp system calls) | No |

Table 47. Supported Operations: `syscall` Layer

| Operation | Supported | Comments |
|---|---|---|
| `ffopen` | Yes | `open` |
| `ffread` | Yes | `read` |
| `ffreadc` | Yes | `read` plus code |
| `ffwrite` | Yes | `write` |
| `ffwritec` | Yes | `write` plus code |
| `ffclose` | Yes | `close` |
| `ffflush` | Yes | None |
| `ffweof` | No | None |
| `ffweod` | Yes | `trunc`(2) |
| `ffseek` | Yes | `lseek`(2) |
| `ffbksp` | No | |

Lower-level layers are not allowed.

## 16.14 The `system` Layer

The `system` layer is implicitly appended to all specification lists, if not explicitly added by the user (unless the `syscall` or `fd` layer is specified). It maps requests to appropriate system calls.

For a description of options, see the `syscall` layer. Lower-level layers are not allowed.

## 16.15 The `text` Layer

The `text` layer performs text blocking by terminating each record with a newline character. It can also recognize and represent the EOF mark. The `text` layer is used with character files and does not work with binary data. The general specification follows:

`text[.`*`type`*`]:[`*`num1`*`]:[`*`num2`*`]`

The keyword syntax is as follows:

`text[.`*`type`*`][.newline=`*`num1`*`][.bufsize=`*`num2`*`]`

The *type* field can have one of the following values:

| Value | Definition |
|---|---|
| `nl` | Newline-separated records. |
| `eof` | Newline-separated records with a special string such as `~e`. More than one EOF in a file is allowed. |

The *num1* field is the decimal value of a single character that represents the newline character. The default value is 10 (octal 012, ASCII line feed).

The *num2* field specifies the working buffer size (in decimal bytes). If any lower-level layers are record oriented, this is also the block size.

Table 48. Data Manipulation: `text` Layer

| Granularity | Data model | Truncate on write |
|---|---|---|
| 8 bits | Record | No |

Table 49. Supported Operations: `text` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | Yes | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | Yes | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | No | |
| `ffweof` | Passed through | | Yes | Only if explicitly requested |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | | Yes | |
| `ffbksp` | No | | No | |

## 16.16 The `user` and `site` Layers

The `user` and `site` layers let users and site administrators build user-defined or site-specific layers to meet special needs. The syntax follows:

`user[`*num1*`]:[`*num2*`]`

`site:[`*num1*`]:[`*num2*`]`

The open processing passes the *num1* and *num2* arguments to the layer and are interpreted by the layers.

See Chapter 17, page 337 for an example of how to create a `user` FFIO layer.

## 16.17 The `vms` Layer

The `vms` layer handles record blocking for three common record types on VAX/VMS operating systems. The general format of the specification follows:

`vms.[`*type*`.`*subtype*`]:[`*num1*`]:[`*num2*`]`

The following is the alternate keyword syntax for this layer:

`vms.[`*type*`.`*subtype*`][.recsize=`*num1*`][.mbs=`*num2*`]`

The following *type* values are supported:

| Value | Definition |
|---|---|
| `f` | VAX/VMS fixed-length records |
| `v` | VAX/VMS variable-length records |
| `s` | VAX/VMS variable-length segmented records |

In addition to the record type, you must specify a record subtype, which has one of the following four values:

| Value | Definition |
|---|---|
| `bb` | Format used for binary blocked transfers |
| `disk` | Same as binary blocked |
| `tr` | Transparent format, for files transferred as a bit stream to and from the VAX/VMS system |
| `tape` | VAX/VMS labeled tape |

The *num1* field is the maximum record size that may be read or written. It is ignored by the `s` record type.

Table 50. Values for Record Size: `vms` Layer

| Field | Minimum | Maximum | Default | Comments |
|---|---|---|---|---|
| `v.bb` | 1 | 32,767 | 32,767 | |
| `v.tape` | 1 | 9995 | 2043 | |
| `v.tr` | 1 | 32,767 | 2044 | |
| `s.bb` | 1 | None | None | No maximum record size |
| `s.tape` | 1 | None | None | No maximum record size |
| `s.tr` | 1 | None | None | No maximum record size |

The *num2* field is the maximum segment or block size that is allowed on input and is produced on output. For `vms.f.tr` and `vms.f.bb`, *num2* should be equal to the record size (*num1*). Because `vms.f.tape` places one or more records in each block, `vms.f.tape` *num2* must be greater than or equal to *num1*.

Table 51. Values for Maximum Block Size: `vms` Layer

| Field | Minimum | Maximum | Default | Comments |
|---|---|---|---|---|
| `v.bb` | 1 | 32,767 | 32,767 | |
| `v.tape` | 6 | 32,767 | 2,048 | |
| `v.tr` | 3 | 32,767 | 32,767 | N / A |
| `s.bb` | 5 | 32,767 | 2,046 | |
| `s.tape` | 7 | 32,767 | 2,048 | |
| `s.tr` | 5 | 32,767 | 2,046 | N / A |

For `vms.v.bb` and `vms.v.disk` records, *num2* is a limit on the maximum record size. For `vms.v.tape` records, it is the maximum size of a block on tape; more specifically, it is the maximum size of a record that will be written to the next lower layer. If that layer is `tape`, *num2* is the tape block size. If it is `cos`, it will be a COS record that represents a tape block. One or more records are placed in each block.

For segmented records, *num2* is a limit on the block size that will be produced. No limit on record size exists. For `vms.s.tr` and `vms.s.bb`, the block size is an upper limit on the size of a segment. For `vms.s.tape`, one or more segments are placed in a tape block. It functions as an upper limit on the size of a segment and a preferred tape block size.

Table 52. Data Manipulation: `vms` Layer

| Granularity | Data model | Truncate on write | Implementation strategy |
|---|---|---|---|
| 8 bits | Record | No for `f` records. Yes for `v` and `s` records. | `f` records for `f` formats. `v` records for `v` formats. |

Table 53. Supported Operations: `vms` Layer

| | Supported operations | | Required of next lower level? | |
|---|---|---|---|---|
| **Operation** | **Supported** | **Comments** | **Used** | **Comments** |
| `ffopen` | Yes | | Yes | |
| `ffread` | Yes | | Yes | |
| `ffreadc` | Yes | | No | |
| `ffwrite` | Yes | | Yes | |
| `ffwritec` | Yes | | No | |
| `ffclose` | Yes | | Yes | |
| `ffflush` | Yes | | No | |
| `ffweof` | Yes and passed through | Yes for `s` records; passed through for others | Yes | Only if explicitly requested |
| `ffweod` | Yes | | Yes | |
| `ffseek` | Yes | `seek(fd,0,0)` only (equals rewind) | Yes | `seek(fd,0,0)` only |
| `ffbksp` | No | | No | |

# Creating a `user` Layer [17]

This chapter explains some of the internals of the FFIO system and explains the ways in which you can put together a `user` or `site` layer.

## 17.1 Internal Functions

The FFIO system has an internal model of data that maps to any given actual logical file type based on the following concepts:

- Data is a stream of bits. Layers must declare their granularity by using the `fffcntl`(3c) call.
- Record marks are boundaries between logical records.
- End-of-file (EOF) marks are a special type of record that exists in some file structures.
- End-of-data (EOD) is a point immediately beyond the last data bit, EOR, or EOF in the file. You cannot read past or write after an EOD. In a case when a file is positioned after an EOD, a write operation (if valid) immediately moves the EOD to a point after the last data bit, end-of-record (EOR), or EOF produced by the write.

All files are streams that contain zero or more data bits that may contain record or file marks.

No inherent hierarchy or ordering is imposed on the file structures. Any number of data bits or EOR and EOF marks may appear in any order. The EOD, if present, is by definition last. Given the EOR, EOF, and EOD return statuses from read operations, only EOR may be returned along with data. When data bits are immediately followed by EOF, the record is terminated implicitly.

Individual layers can impose restrictions for specific file structures that are more restrictive than the preceding rules. For instance, in COS blocked files, an EOR always immediately precedes an EOF.

Successful mappings were used for all logical file types supported, except formats that have more than one type of partitioning for files (such as end-of-group or more than one level of EOF). For example, some file formats have level numbers in the partitions. FFIO maps level 017 to an EOF. No other handling is provided for these level numbers.

Internally, there are two main protocol components: the operations and the stat structure.

### 17.1.1 The Operations Structure

Many of the operations try to mimic the UNICOS/mp and UNICOS/lc system calls. In the man pages for `ffread`(3c), `ffwrite`(3c), and others, the calls can be made without the optional parameters and appear like the system calls. Internally, all parameters are required.

Table 54 provides a brief synopsis of the interface routines that are supported at the user level. Each of these `ff` entry points checks the parameters and issues the corresponding internal call. Each interface routine provides defaults and dummy arguments for those optional arguments that the user does not provide.

Each layer must have an internal entry point for all of these operations, although in some cases the entry point may simply issue an error or do nothing. For example, the `syscall` layer uses `_ff_noop` for the `ffflush` entry point because it has no buffer to flush, and it uses `_ff_err2` for the `ffweof` entry point because it has no representation for EOF. No optional parameters for calls to the internal entry points exist. All arguments are required.

Table 54 lists the variables for the internal entry points and the variable definitions. An internal entry point must be provided for all of these operations:

Table 54. C Program Entry Points

| Variable | Definition |
|---|---|
| *fd* | The FFIO pointer `(struct fdinfo *)fd`. |
| *file* | A `char*` file. |
| *flags* | File status flag for open, such as `O_RDONLY`. |
| *buf* | Bit pointer to the user data. |
| *nb* | Number of bytes. |
| *ret* | The status returned; >=0 is valid, <0 is error. |
| *stat* | A pointer to the status structure. |
| *fulp* | The value `FULL` or `PARTIAL` defined in `ffio.h` for full or partial-record mode. |
| *&ubc* | A pointer to the unused bit count; this ranges from 0 to 7 and represents the bits not used in the last byte of the operation. It is used for both input and output. |
| *pos* | A byte position in the file. |
| *opos* | The old position of the file, just like the `system` call. |
| *whence* | The same as the `syscall`. |
| *cmd* | The command request to the `fffcntl`(3c) call. |
| *arg* | A generic pointer to the `fffcntl` argument. |
| *mode* | Bit pattern denoting file's access permissions. |
| *argp* | A pointer to the input or output data. |
| *len* | The length of the space available at *argp*. It is used primarily on output to avoid overwriting the available memory. |

### 17.1.2 FFIO and the `stat` Structure

The `stat` structure contains four fields in the current implementation. They mimic the `iows` structure of the UNICOS/mp and UNICOS/lc `ASYNC syscalls` to the extent possible. All operations are expected to update the *stat* structure on each call. The `SETSTAT` and `ERETURN` macros are provided in the `ffio.h` file for this purpose.

The fields in the *stat* structure are as follows:

| Status field | Description |
|---|---|
| `stat.sw_flag` | 0 indicates outstanding; 1 indicates I/O complete. |
| `stat.sw_error` | 0 indicates no error; otherwise, the error number. |
| `stat.sw_count` | Number of bytes transferred in this request. This number is rounded up to the next integral value if a partial byte is transferred. |
| `stat.sw_stat` | This tells the status of the I/O operation. The `FFSTAT(stat)` macro accesses this field. The following values are valid:<br><br>`FFBOD`: At beginning-of-data (BOD).<br><br>`FFCNT`: Request terminated by count (either the count of bytes before EOF or EOD in the file or the count of the request).<br><br>`FFEOR`: Request termination by EOR or a full record mode read was processed.<br><br>`FFEOF`: EOF encountered.<br><br>`FFEOD`: EOD encountered.<br><br>`FFERR`: Error encountered. |

If `count` is satisfied simultaneously with EOR, the `FFEOR` is returned.

The EOF and EOD status values must never be returned with data. This means that if a byte-stream file is being traversed and the file contains 100 bytes and then an EOD, a read of 500 bytes will return with a stat value of FFCNT and a return byte count of 100. The next read operation returns FFEOD and a count of 0.

A FFEOF or FFEOD status is always returned with a 0-byte transfer count.

## 17.2 user Layer Example

This section gives a complete and working user layer. It traces I/O at a given level. All operations are passed through to the next lower-level layer, and a trace record is sent to the trace file.

The first step in generating a user layer is to create a table that contains the addresses for the routines that will fulfill the required functions described in Section 17.1.1, page 338 and Section 17.1.2, page 340. The format of the table can be found in struct xtr_s, which is found in the <ffio.h> file. No restriction is placed on the names of the routines, but the table must be called _usr_ffvect for it to be recognized as a user layer. In the example, the declaration of the table can be found with the code in the _usr_open routine.

To use this layer, you must take advantage of the weak external files in the library. The following script fragment is suggested for UNICOS/mp and UNICOS/lc systems:

```
# -D_LIB_INTERNAL is required to obtain the
#  declaration of struct fdinfo in <ffio.h>
#
cc -c -D_LIB_INTERNAL -hcalchars usr*.c
cat usr*.o > user.o
#
# Note that the -F option is selected that loads
# and links the entries despite not having any
# hard references.

cc -o user.o myprog.o
assign -F user,others... fort.1
./abs
```

```
static char USMID[] = "@(#)code/usrbksp.c       1.0    ";
/*   COPYRIGHT CRAY INC.
*    UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
*    THE COPYRIGHT LAWS OF THE UNITED STATES.
*/
#include <ffio.h>
#include "usrio.h"
/*
*    trace backspace requests
*/
int
_usr_bksp(struct fdinfo *fio, struct ffsw *stat)
     {
     struct fdinfo *llfio;
     int ret;

     llfio = fio->fioptr;
     _usr_enter(fio, TRC_BKSP);
     _usr_pr_2p(fio, stat);
     ret = XRCALL(llfio, backrtn) llfio, stat);
     _usr_exit(fio, ret, stat);
    return(0);
    }
```

```
static char USMID[] = "@(#)code.usrclose.c     1.0    ";
/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <malloc.h>
#include <ffio.h>
#include "usrio.h"
/*
 *  trace close requests
 */
int
_usr_close(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        struct trace_f *pinfo;
        int ret;
        llfio = fio->fioptr;
/*
 *  lyr_info is a place in the fdinfo block that holds
 *  a pointer to the layer's private information.
 */
        pinfo = (struct trace_f *)fio->lyr_info;

        _usr_enter(fio, TRC_CLOSE);
        _usr_pr_2p(fio, stat);
/*
 *  close file
 */
        ret = XRCALL(llfio, closertn) llfio, stat);
/*
 *  It is the layer's responsibility to clean up its mess.
 */
        free(pinfo->name);
        pinfo->name = NULL;
        free(pinfo);
        _usr_exit(fio, ret, stat);
        (void) close(pinfo->usrfd);
        return(0);
        }

static char USMID[] = "@(#)code/usrfcntl.c       1.0    ";
```

```
/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <ffio.h>
#include "usrio.h"
/*
 *  trace fcntl requests
 *
 *  Parameters:
 *   fd      - fdinfo pointer
 *   cmd     - command code
 *   arg     - command specific parameter
 *   stat    - pointer to status return word
 *
 * This fcntl routine passes the request down to the next lower
 * layer, so it provides nothing of its own.
 *
 * When writing a user layer, the fcntl routine must be provided,
 * and must provide correct responses to one essential function and
 * two desirable functions.
 *
 *  FC_GETINFO: (essential)
 *  If the 'cmd' argument is FC_GETINFO, the fields of the 'arg' is
 *  considered a pointer to an ffc_info_s structure, and the fields
 *  must be filled. The most important of these is the ffc_flags
 *  field, whose bits are defined in <ffio.h>.(Look for FFC_STRM
 *  through FFC_NOTRN)
 *  FC_STAT: (desirable)
 *  FC_RECALL: (desirable)
 */
int
_usr_fcntl(struct fdinfo *fio, int cmd, void *arg, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        struct trace_f *pinfo;
        int ret;

        llfio = fio->fioptr;
        pinfo = (struct trace_f *)fio->lyr_info;
        _usr_enter(fio, TRC_FCNTL);
        _usr_info(fio, "cmd=%d ", cmd);
        ret = XRCALL(llfio, fcntlrtn) llfio, cmd, arg, stat);
```

```
        _usr_exit(fio, ret, stat);
        return(ret);
        }
static char USMID[] = "@(#)code/usropen.c       1.0    ";

/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <fcntl.h>
#include <malloc.h>
#include <ffio.h>
#include "usrio.h"
#define SUFFIX      ".trc"

/*
 * trace open requests;
 * The following routines compose the user layer. They are declared
 * in "usrio.h"
 */

/*
 * Create the _usr_ffvect structure.  Note the _ff_err inclusion to
 * account for the listiortn, which is not supported by this user
 * layer
 */
struct xtr_s _usr_ffvect =
  {
  _usr_open,   _usr_read,   _usr_reada,   _usr_readc,
  _usr_write,  _usr_writea, _usr_writec,  _usr_close,
  _usr_flush,  _usr_weof,   _usr_weod,    _usr_seek,
  _usr_bksp,   _usr_pos,    _usr_err,     _usr_fcntl
   };

_ffopen_t
_usr_open(
         const char *name,
         int flags,
         mode_t mode,
         struct fdinfo * fio,
         union spec_u *spec,
```

```
          struct ffsw *stat,
          long cbits,
          int cblks,
          struct gl_o_inf *oinf)
          {
          union spec_u *nspec;
          struct fdinfo *llfio;
          struct trace_f *pinfo;
          char *ptr = NULL;
          int  namlen, usrfd;
          _ffopen_t nextfio;
          char buf[256];

          namlen = strlen(name);
          ptr = malloc(namlen + strlen(SUFFIX) + 1);
          if (ptr == NULL) goto badopen;
          pinfo = (struct trace_f *)malloc(sizeof(struct trace_f));
          if (pinfo == NULL) goto badopen;

          fio->lyr_info = (char *)pinfo;
/*
 *  Now, build the name of the trace info file, and open it.
 */
          strcpy(ptr, name);
          strcat(ptr, SUFFIX);
          usrfd = open(ptr, O_WRONLY | O_APPEND | O_CREAT, 0666);
/*
 *  Put the file info into the private data area.
 */
          pinfo->name = ptr;
          pinfo->usrfd = usrfd;
          ptr[namlen] = '\0';
/*
 *  Log the open call
 */
          _usr_enter(fio, TRC_OPEN);
          sprintf(buf,"(\"%s\", %o, %o...);\n", name, flags, mode);
          _usr_info(fio, buf, 0);
/*
 *  Now, open the lower layers
 */
          nspec = spec;
          NEXT_SPEC(nspec);
```

```
          nextfio = _ffopen(name, flags, mode, nspec, stat, cbits, cblks,
                     NULL, oinf);
          _usr_exit_ff(fio, nextfio, stat);
          if (nextfio != _FFOPEN_ERR)
                 {
                 DUMP_IOB(fio); /* debugging only */
                 return(nextfio);
                 }
/*
 *  End up here only on an error
 *
 */

badopen:
        if(ptr != NULL) free(ptr);
        if (fio->lyr_info != NULL) free(fio->lyr_info);
        _SETERROR(stat, FDC_ERR_NOMEM, 0);
         return(_FFOPEN_ERR);
        }
        _usr_err(struct fdinfo *fio)
        {
         _usr_info(fio,"ERROR: not expecting this routine\n",0);
         return(0);
        }
```

```
static char USMID[] = "@(#)code/usrpos.c       1.1    ";

/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#include <ffio.h>
#include "usrio.h"

/*
 *  trace positioning requests
 */

_ffseek_t
_usr_pos(struct fdinfo *fio, int cmd, void *arg, int len, struct ffsw *stat)
   {
   struct fdinfo *llfio;
   struct trace_f *usr_info;
   _ffseek_t ret;

   llfio = fio->fioptr;
   usr_info = (struct trace_f *)fio->lyr_info;

   _usr_enter(fio,TRC_POS);
   _usr_info(fio, " ", 0);
   ret = XRCALL(llfio, posrtn) llfio, cmd, arg, len, stat);
   _usr_exit_sk(fio, ret, stat);
   return(ret);
   }
```

```
static char USMID[] = "@(#)code/usrprint.c       1.1    ";

/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */
#include <stdio.h>
#include <ffio.h>
#include "usrio.h"

static char *name_tab[] =
                        {
```

```
                        "???",
                        "ffopen",
                        "ffread",
                        "ffreadc",
                        "ffwrite",
                        "ffwritec",
                        "ffclose",
                        "ffflush",
                        "ffweof",
                        "ffweod",
                        "ffseek",
                        "ffbksp",
                        "fflistio",
                        "fffcntl",
                        };

/*
 * trace printing stuff
 */
int
_usr_enter(struct fdinfo *fio, int opcd)
                {
                char buf[256], *op;
                struct trace_f *usr_info;

                op = name_tab[opcd];
                usr_info = (struct trace_f *)fio->lyr_info;
                sprintf(buf, "TRCE: %s ",op);
                write(usr_info->usrfd, buf, strlen(buf));
                return(0);
                }

void
_usr_info(struct fdinfo *fio, char *str, int arg1)
       {
       char buf[256];
       struct trace_f *usr_info;

       usr_info = (struct trace_f *)fio->lyr_info;
       sprintf(buf, str, arg1);
       write(usr_info->usrfd, buf, strlen(buf));
       }
```

```
void
_usr_exit(struct fdinfo *fio, int ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
        write(usr_info->usrfd, buf, strlen(buf));
        }

void
_usr_exit_ss(struct fdinfo *fio, ssize_t ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
        sprintf(buf, "TRCX:   ret=%ld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
        write(usr_info->usrfd, buf, strlen(buf));
        }

void
_usr_exit_ff(struct fdinfo *fio, _ffopen_t ret, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        sprintf(buf, "TRCX:   ret=%d, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
        write(usr_info->usrfd, buf, strlen(buf));
 }
void
_usr_exit_sk(struct fdinfo *fio, _ffseek_t ret, struct ffsw *stat)
        {
        char buf[256];
```

```
        struct trace_f *usr_info;
        usr_info = (struct trace_f *)fio->lyr_info;
        fio->ateof = fio->fioptr->ateof;
        fio->ateod = fio->fioptr->ateod;
        sprintf(buf, "TRCX:   ret=%ld, stat=%d, err=%d\n",
             ret, stat->sw_stat, stat->sw_error);
#endif
        write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_rwc(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
        write(usr_info->usrfd, buf, strlen(buf));
        if (fulp == FULL)
              sprintf(buf,"FULL");
       else
               sprintf(buf,"PARTIAL");
               write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_rww(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
```

```
        sprintf(buf,"(fd / %lx */, &memc[%lx], %ld, &statw[%lx], ",
             fio, BPTR2CP(bufptr), nbytes, stat);
        write(usr_info->usrfd, buf, strlen(buf));
        if (fulp == FULL)
             sprintf(buf,"FULL");
        else
        sprintf(buf,"PARTIAL");
        write(usr_info->usrfd, buf, strlen(buf));
        sprintf(buf,", &conubc[%d]; ", *ubc);
        write(usr_info->usrfd, buf, strlen(buf));
        }
void
_usr_pr_2p(struct fdinfo *fio, struct ffsw *stat)
        {
        char buf[256];
        struct trace_f *usr_info;

        usr_info = (struct trace_f *)fio->lyr_info;
        sprintf(buf,"(fd / %lx */,  &statw[%lx], ",
             fio,  stat);
         write(usr_info->usrfd, buf, strlen(buf));
         }
```

```
static char USMID[] = "@(#)code/usrread.c       1.0    ";
/*  COPYRIGHT CRAY INC.
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#include <ffio.h>
#include "usrio.h"

/*
 * trace read requests
 *
 * Parameters:
 *  fio     - Pointer to fdinfo block
 *  bufptr  - bit pointer to where data is to go.
 *  nbytes  - Number of bytes to be read
 *  stat    - pointer to status return word
 *  fulp    - full or partial read mode flag
 *  ubc     - pointer to unused bit count
 */
ssize_t
_usr_read(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
     {
     struct fdinfo *llfio;
     char *str;
     ssize_t ret;
     llfio = fio->fioptr;
     _usr_enter(fio, TRC_READ);
     _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
     ret = XRCALL(llfio, readrtn) llfio, bufptr, nbytes, stat,
          fulp, ubc);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
     }
```

```
/*
 * trace reada (asynchronous read) requests
 *
 * Parameters:
 *  fio     - Pointer to fdinfo block
 *  bufptr  - bit pointer to where data is to go.
 *  nbytes  - Number of bytes to be read
 *  stat    - pointer to status return word
 *  fulp    - full or partial read mode flag
 *  ubc     - pointer to unused bit count
 */
ssize_t
_usr_reada(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
    {
    struct fdinfo *llfio;
    char *str;
    ssize_t ret;

    llfio = fio->fioptr;
    _usr_enter(fio, TRC_READA);
    _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
    ret = XRCALL(llfio,readartn)llfio,bufptr,nbytes,stat,fulp,ubc);
    _usr_exit_ss(fio, ret, stat);
    return(ret);
    }
```

```
/*
* trace readc requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be read
*  stat    - pointer to status return word
*  fulp    - full or partial read mode flag
*/
ssize_t
_usr_readc(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
     {
     struct fdinfo *llfio;
     char *str;
     ssize_t ret;
     llfio = fio->fioptr;
     _usr_enter(fio, TRC_READC);
     _usr_pr_rwc(fio, bufptr, nbytes, stat, fulp);
     ret = XRCALL(llfio, readcrtn)llfio, bufptr, nbytes, stat,
         fulp);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
        }
```

```
/*
* _usr_seek()
*
* The user seek call should mimic the UNICOS/mp lseek system call as
* much as possible.
*/
_ffseek_t
_usr_seek(
struct fdinfo *fio,
off_t pos,
int whence,
struct ffsw *stat)
        {
        struct fdinfo *llfio;
        _ffseek_t ret;
        char buf[256];

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_SEEK);
        sprintf(buf,"pos %ld, whence %d\n", pos, whence);
        _usr_info(fio, buf, 0);
        ret = XRCALL(llfio, seekrtn) llfio, pos, whence, stat);
        _usr_exit_sk(fio, ret, stat);
        return(ret);
        }
```

```
static char USMID[] = "@(#)code/usrwrite.c       1.0    ";

/*  COPYRIGHT CRAY INC.
*  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
*  THE COPYRIGHT LAWS OF THE UNITED STATES.
*/

#include <ffio.h>
#include "usrio.h"

/*
* trace write requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*  ubc     - pointer to unused bit count (not used for IBM)
*/
ssize_t
_usr_write(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
     {
     struct fdinfo *llfio;
     ssize_t ret;

     llfio = fio->fioptr;
     _usr_enter(fio, TRC_WRITE);
     _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
     ret = XRCALL(llfio, writertn) llfio, bufptr, nbytes, stat,
          fulp,ubc);
     _usr_exit_ss(fio, ret, stat);
     return(ret);
     }
```

```
/*
* trace writea requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*  ubc     - pointer to unused bit count (not used for IBM)
*/
ssize_t
_usr_writea(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp,
int *ubc)
    {
    struct fdinfo *llfio;
    ssize_t ret;

    llfio = fio->fioptr;
    _usr_enter(fio, TRC_WRITEA);
    _usr_pr_rww(fio, bufptr, nbytes, stat, fulp, ubc);
    ret = XRCALL(llfio, writeartn) llfio, bufptr, nbytes, stat,
        fulp,ubc);
    _usr_exit_ss(fio, ret, stat);
    return(ret);
    }

/*
* trace writec requests
*
* Parameters:
*  fio     - Pointer to fdinfo block
*  bufptr  - bit pointer to where data is to go.
*  nbytes  - Number of bytes to be written
*  stat    - pointer to status return word
*  fulp    - full or partial write mode flag
*/
```

```
ssize_t
_usr_writec(
struct fdinfo *fio,
bitptr bufptr,
size_t nbytes,
struct ffsw *stat,
int fulp)
        {
        struct fdinfo *llfio;
        ssize_t ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WRITEC);
        _usr_pr_rwc(fio, bufptr, nbytes, stat, fulp);
        ret = XRCALL(llfio, writecrtn)llfio,bufptr, nbytes, stat,
           fulp);
        _usr_exit_ss(fio, ret, stat);
        return(ret);
        }
/*
* Flush the buffer and clean up
* This routine should return 0, or -1 on error.
*/
int
_usr_flush(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;
        llfio = fio->fioptr;

        _usr_enter(fio, TRC_FLUSH);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, flushrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }
```

```
/*
* trace WEOF calls
*
* The EOF is a very specific concept.   Don't confuse it with the
* UNICOS/mp EOF, or the truncate(2) system call.
*/
int
_usr_weof(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WEOF);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, weofrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }

/*
* trace WEOD calls
*
* The EOD is a specific concept.  Don't confuse it with the UNICOS/mp
*  EOF.  It is usually mapped to the truncate(2) system call.
*/
int
_usr_weod(struct fdinfo *fio, struct ffsw *stat)
        {
        struct fdinfo *llfio;
        int ret;

        llfio = fio->fioptr;
        _usr_enter(fio, TRC_WEOD);
        _usr_info(fio, "\n",0);
        ret = XRCALL(llfio, weodrtn) llfio, stat);
        _usr_exit(fio, ret, stat);
        return(ret);
        }

/* USMID @(#)code/usrio.h       1.1    */

/*  COPYRIGHT CRAY INC.
```

```
 *  UNPUBLISHED -- ALL RIGHTS RESERVED UNDER
 *  THE COPYRIGHT LAWS OF THE UNITED STATES.
 */

#define TRC_OPEN 1
#define TRC_READ 2
#define TRC_READA 3
#define TRC_READC 4
#define TRC_WRITE 5
#define TRC_WRITEA 6
#define TRC_WRITEC 7
#define TRC_CLOSE 8
#define TRC_FLUSH 9
#define TRC_WEOF 10
#define TRC_WEOD 11
#define TRC_SEEK 12
#define TRC_BKSP 13
#define TRC_POS 14
#define TRC_UNUSED 15
#define TRC_FCNTL 16

struct trace_f
        {
        char    *name;          /* name of the file */
        int   usrfd;            /* file descriptor of trace file */
        };
/*
 * Prototypes
 */
extern int _usr_bksp(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_close(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_fcntl(struct fdinfo *fio, int cmd, void *arg,
           struct ffsw *stat);
extern _ffopen_t _usr_open(const char *name, int flags,
           mode_t mode, struct fdinfo * fio, union spec_u *spec,
           struct ffsw *stat, long cbits, int cblks,
           struct gl_o_inf *oinf);
extern int _usr_flush(struct fdinfo *fio, struct ffsw *stat);
extern _ffseek_t _usr_pos(struct fdinfo *fio, int cmd, void *arg,
           int len, struct ffsw *stat);
extern ssize_t _usr_read(struct fdinfo *fio, bitptr bufptr,
           size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_reada(struct fdinfo *fio, bitptr bufptr,
```

```
            size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_readc(struct fdinfo *fio, bitptr bufptr,
            size_t nbytes, struct ffsw *stat, int fulp);
extern _ffseek_t _usr_seek(struct fdinfo *fio, off_t pos, int whence,
            struct ffsw *stat);
extern ssize_t _usr_write(struct fdinfo *fio, bitptr bufptr,
            size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_writea(struct fdinfo *fio, bitptr bufptr,
            size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern ssize_t _usr_writec(struct fdinfo *fio, bitptr bufptr,
            size_t nbytes, struct ffsw *stat, int fulp);
extern int _usr_weod(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_weof(struct fdinfo *fio, struct ffsw *stat);
extern int _usr_err();

/*
 * Prototypes for routines that are used by the user layer.
 */
extern int _usr_enter(struct fdinfo *fio, int opcd);
extern void _usr_info(struct fdinfo *fio, char *str, int arg1);
extern void _usr_exit(struct fdinfo *fio, int ret, struct ffsw *stat);
extern void _usr_exit_ss(struct fdinfo *fio, ssize_t ret,
            struct ffsw *stat);
extern void _usr_exit_ff(struct fdinfo *fio, _ffopen_t ret,
            struct ffsw *stat);
extern void _usr_exit_sk(struct fdinfo *fio, _ffseek_t ret,
            struct ffsw *stat);
extern void _usr_pr_rww(struct fdinfo *fio, bitptr bufptr,
            size_t nbytes, struct ffsw *stat, int fulp, int *ubc);
extern void _usr_pr_2p(struct fdinfo *fio, struct ffsw *stat);
```

# Numeric File Conversion Routines [18]

This chapter contains information about data conversion, moving data between machines, and implicit and explicit data conversion. It also explains the support provided for reading and writing files in foreign formats, including record blocking and numeric and character conversion.

These routines convert data (primarily floating-point data, but also integer and character data, as well as Fortran complex and logical data) from your system's native representation to a foreign representation, and vice versa.

## 18.1 Conversion Overview

Data can be transferred between UNICOS/mp and UNICOS/lc systems and other computer systems in several ways. These methods include the use of utilities built on TCP/IP (such as `ftp`). You can also use the data conversion library routines to convert data.

Cray X1 and X2 systems support the Institute of Electrical and Electronics Engineers (IEEE) format by default and also support conversion to and from IBM, VAX/VMS, and other formats. For each foreign file type, several supported file and record formats exist or explicit or implicit data conversion can also be used.

When processing foreign data, you must consider the interactions between the data formats and the method of data transfer. This section describes, in broad terms, the techniques available to do data conversion.

*Explicit conversion* is the process by which the user performs calls to subroutines that convert the *native* data to and from the *foreign* data formats. These routines are provided for many data formats. This is discussed in more detail in Section 18.3.1, page 365.

*Implicit conversion* is the process by which you declare that a particular file contains foreign data and/or record blocking and then request that the run-time library perform appropriate transformations on the data to make it useful to the program at I/O time. This method of record and data format conversion requires changes in command scripts. This is discussed in more detail in Section 18.3.2, page 365.

## 18.2 Transferring Data

This section describes several ways to transfer data, including using the `fdcp` and other TCP/IP tools.

### 18.2.1 Using `fdcp` to Transfer Files

The `fdcp`(1) command can handle data that is not a simple disk-resident byte stream. The `fdcp` command assumes that both the data and any record, including an end-of-file (EOF) record, can be copied from one file to another. Record structures can be preserved or removed. EOF records can be preserved either as EOF records in the output file or used to separate the delimited data in the input file into separate files.

The `fdcp` command does not perform data conversion; the only transformations done are on the record and file structures (`fdcp` transforms block, record, and file control words from one format to another).

If no `assign`(1) information is available for a file, the `system` layer is used. If the file being accessed is on disk and if no `assign -F` attribute is used, the `syscall` layer is used.

### 18.2.2 Using `ftp` to Move Data between Systems

When transferring a file to a foreign system, FFIO can create the file in the correct foreign format, but `ftp` cannot establish the right attributes on the file so that the foreign operating system can handle it correctly. Therefore, `ftp` is not useful as a transfer agent on IBM and VMS systems for binary data. Its utility is limited to those systems that do not embed record attributes in the system file information.

## 18.3 Data Item Conversion

The UNICOS/mp operating system provides both the implicit and explicit conversion of data items. Explicit conversion means that your code invokes the routines that convert between native systems and foreign representations.

Options to the `assign`(1) command control implicit conversion. Implicit conversion is usually transparent to users and is available only to Fortran programmers. The following sections describe these data conversion types and provides direction in choosing the best one for your situation.

### 18.3.1 Explicit Data Item Conversion

The Cray Fortran compiler library contains a set of subroutines that convert between Cray data formats and the formats of various vendors. These routines are callable from any programming language supported by Cray. The explicit conversion routines convert between IBM, VAX/VMS, or generic IEEE binary data formats and Cray 32-bit IEEE binary data formats. For complete details, see the individual man pages for each routine. These subroutines provide an efficient way to convert data that was read into system central memory.

Table 55 lists the explicit data conversion subroutines.

Table 55. Explicit Data Conversion Routines

| Cray X1 and X2 Systems | | |
|---|---|---|
| **Name** | **Foreign -> Cray** | **Cray -> Foreign** |
| IBM | `IBM2IEG` | `IEG2IBM` |
| VAX/VMS | `VAX2IEG` | `IEG2VAX` |
| IEEE little-endian | `IEU2IEG` | `IEG2IEU` |
| Cray T3E IEEE (64-bit) | `CRI2IEG` | `IEG2CRI` |
| SGI MIPS | `MIPS2IEG` | `IEG2MIPS` |
| User conversion | `USR2IEG` | `IEG2USR` |
| Site conversion | `STE2IEG` | `IEG2STE` |

See the individual man pages for details about the syntax and arguments for each routine.

### 18.3.2 Implicit Data Item Conversion

Implicit data conversion in Fortran requires no explicit action by the program to convert the data in the I/O stream other than using the `assign` command to instruct the libraries to perform conversion. For details, see the `assign`(1) man page.

The implicit data conversion process is performed in two steps:

1. Record format conversion
2. Data conversion

Record format conversion interprets or converts the internal record blocking structures in the data stream to gain record-level access to the data. The data contained in the records can then be converted.

Using implicit conversion, you can select record blocking or deblocking alone, or you can request that the data items be converted automatically. When enabled, record format conversion and data item conversion occur transparently and simultaneously. Changes are usually not required in your Fortran code.

To enable conversion of foreign record formats, specify the appropriate record type with the `assign -F` command. The `-N` (numeric conversion) and `-C` (character conversion) `assign` options control conversion of data contained in a record. If `-F` is specified but `-N` and `-C` are not, the libraries interpret the record format but they do not convert data. You can obtain information about the type of data that will be converted (and, therefore, the type of conversion that will be performed) from the Fortran I/O list.

If `-N` is used and `-C` is not, an appropriate character conversion type is selected by default, as shown in Table 56.

Table 56. Implicit Data Conversion Types

| -N option | -C default | Meaning |
|---|---|---|
| none | none | No numeric conversion |
| default | default | No numeric conversion; IEEE 32-bit |
| cray | ASCII | Cray "classic" floating-point |
| ibm | EBCDIC | IBM 360/370-style numeric conversion |
| vms | ASCII | VAX/VMS numeric conversion |
| ieee | ASCII | Generic IEEE data (no data conversion) |
| ieee_32 | ASCII | Generic 32-bit IEEE data. No data conversion except for items which are promoted via -s default64 (or -sreal64 or -sinteger64). They are handled as if they had not been promoted. That is, default sized variables will be read and written as if no -s option is specified. |
| mips | ASCII | SGI MIPS IEEE numeric conversion (128-bit floating-point is "double double" format) |
| ieee_64 | ASCII | Cray 64-bit IEEE numeric conversion |
| ieee_le | ASCII | Little endian 32-bit IEEE numeric conversion |
| ultrix | ASCII | Alias for above |
| t3e | ASCII | Cray 64-bit IEEE numeric conversion; denormalized numbers flushed to zero |
| t3d | ASCII | Alias for t3e |
| user | ASCII | User defined numeric conversion |
| site | ASCII | Site defined numeric conversion |
| ia | ASCII | Intel architecture |
| swap_endian | ASCII | The endian of data and control images is swapped during unformatted input and output |

Cray supports conversion of the supported formats and data types through standard Fortran formatted, unformatted, list-directed, and namelist I/O and through BUFFER IN and BUFFER OUT statements.

Generally, read, write, and rewind are supported for all record formats. Other capabilities such as backspace are usually not available but can be made to work if a blocking type can be used to support it. See the sections on the specific layers for complete details.

If you select the `-N` option, the libraries perform data conversion for Fortran unformatted statements and `BUFFER IN` and `BUFFER OUT` I/O statements. Data is converted between its native representation and a foreign representation, according to its Fortran data type.

If the value in a native element is too large to fit in the foreign element, the foreign element is set to the largest or smallest possible value; no error is generated. When converting from a native element to a smaller foreign element, precision is also lost due to truncation of the floating-point mantissa.

If the `assign -N user` or `assign -N site` command is specified, the user or site must provide `site` numeric data conversion routines. They follow the same calling conventions as the other explicit routines.

For implicit conversion, specify format characteristics on an `assign` command.

Files can be converted to either:

- A disk file
- A file transferred from a computer other than the Cray X1 or X2 system

When a Fortran I/O operation is performed on the file, the appropriate file format and data conversions are performed during the I/O operation. Data conversion is performed on each data item, based on the type of the Fortran variable in the I/O list.

For example, if the first read of a foreign file format is like the following example, the library interprets any blocking structures in the file that precede the first data record:

```
INTEGER(KIND=8) INT
REAL(KIND=8) FLOAT1, FLOAT2
READ (10) INT,FLOAT1,FLOAT2
```

These vary depending on the file type and record format. The first 32 bits of data (in IBM format, for example) are extracted, sign-extended, and stored in the `INT` Fortran variable. The next 32 bits are extracted, converted to native floating-point format, and stored in the `FLOAT1` Fortran variable.

The next 32 bits are extracted, converted, and stored into the `FLOAT2` Fortran variable. The library then skips to the end of the foreign logical record. When writing from a native system to a foreign format (for example, if in the previous example `WRITE(10)` was used), precision is lost when converting from a 64-bit representation to 32-bit representation if the program was compiled with the `-s default64` compiler option and the `INT`, `FLOAT1`, and `FLOAT2` variables are default types.

### 18.3.3 Choosing a Conversion Method

As with any software process, the various options for data conversion have advantages and disadvantages, which are discussed in this section. As a set, various data conversion options provide choices in methods of file processing for front-end systems. No one option is best for all applications.

#### 18.3.3.1 Explicit Conversion

Explicit data conversion has some distinct advantages, including:

- Providing direct control (including some options not available through implicit conversion) over data conversion
- Allowing programmers to control and schedule the conversion for a convenient and appropriate time
- Performing conversion on large data areas as vector operations, usually increasing performance

One disadvantage of using explicit conversion is that explicit routines require changes to the source code.

#### 18.3.3.2 Implicit Conversion

An advantage when using implicit conversion is that you do not have to change the source code.

Disadvantages of using implicit conversion include:

- Requiring script changes to the `assign`(1) command
- Making conversion less efficient on a record-by-record basis
- Doing conversion at I/O time according to the declared data types, allowing little flexibility for nonstandard requirements

### 18.3.4 Disabling Conversion Types

The subroutines required to handle data conversion must be loaded into absolute binary files. By default, the run-time libraries include references to routines required to support the forms of implicit conversion enabled in the foreign data conversion configuration file, usually named `fdcconfig.h`.

## 18.4 Foreign Conversion Techniques

This section contains some tips and techniques for the following conversion types:

| Conversion type | Convert data to/from |
|---|---|
| UNICOS files | Older Cray UNICOS systems |
| IBM conversion | IBM machines |
| IEEE conversion | Various types of workstations and different vendors that support IEEE floating-point format |
| VAX/VMS conversion | DEC VAX machines that run MVS |

### 18.4.1 UNICOS/mp and UNICOS/lc Conversions

The UNICOS/mp and UNICOS/lc operating systems use `f77` format as the default format for Fortran unformatted sequential files.

To swap the data and control images when accessing unformatted files created on a system with a different endian, use the following command:

```
assign -N swap_endian f:filename
```

Previous UNICOS operating systems used COS blocking for all blocked files, so conversion is necessary when moving unformatted, blocked, sequential files from those Cray systems to the UNICOS/mp and UNICOS/lc operating systems. Two common COS file types require some conversion to make them useful on the UNICOS/mp and UNICOS/lc operating systems.

To read or write unformatted files from UNICOS systems, use one of the following commands:

- If moving a Cray floating point format file from a Cray SV1 series system, use the following command:

  ```
  assign -F cos -N cray cosfile
  ```

- If moving an IEEE floating point format file from a Cray SV1 series system, use the following command:

  ```
  assign -F cos -N ieee_64 cosfile
  ```

- If moving a file from a Cray T3E system, use the following command:

  ```
  assign -F cos -N t3e cosfile
  ```

### 18.4.2 IBM Overview

To convert and transfer data between Cray X1 series or X2 systems and an IBM/MVS or VM (360/370 style) system, you must understand the differences between the UNICOS/mp and UNCOS/lc file system and file formats, and those on the IBM system(s). On both VM and MVS, the file system is record-oriented.

The most obvious form of data conversion is between the IBM EBCDIC character set and the ASCII character set used on UNICOS/mp and UNICOS/lc systems. Most of the utilities that transfer files to and from the IBM systems automatically convert both the record structures and character set to the UNICOS/mp and UNICOS/lc text format and to ASCII. For example, `ftp` performs these conversions and does not require any further conversion on UNICOS/mp and UNICOS/lc systems.

Binary data, however, is more complicated. You must first find a way to transfer the file and to preserve the record boundaries. If workstations are available, this is simple. Few problems are caused by transferring the file and preserving record boundaries.

Cray supports the following IBM record formats:

| Format | Description |
|---|---|
| U | Undefined record format |
| F | Fixed-length records, one record per block |
| FB | Fixed-length, blocked records |
| V | Variable-length records |
| VB | Variable-length, blocked records |
| VBS | Variable-length, blocked, spanned records |

### 18.4.3 IEEE Conversion

By default Cray X1 series and X2 systems use 32-bit IEEE standard floating point, with two's-complement arithmetic and the ASCII character set. This standard is also used by many workstations and personal computers. The logical values in these implementations are usually the same for Fortran and C; they use zero for false and nonzero for true. It is also common to see the Fortran record blocking used by the Fortran run-time library on unformatted sequential files.

No IEEE record format exists, but the IEEE implicit and explicit data conversion routine facilities are provided with the assumption that many of these things are true.

Most computer systems that use the IEEE data formats run operating systems based on UNIX software and use `f77` record blocking. You can use the `rcp` or `ftp` commands to transfer files. In most cases, the following command should work:

```
assign -F f77 fort.1
```

When writing files in the Fortran format, remember that you can gain a large performance boost by ensuring that the records being written fit in the working buffer of the Fortran layer.

On Cray X1 series and X2 systems, data types can be declared as 32 bits in size and can then be read or written directly. This is the most direct and efficient method to read or write data files for IEEE workstations. The user can alter the declarations of the variables used in the Fortran I/O list to declare them as `KIND=4` or as `REAL*4` (or `INTEGER*4`).

For example, to read a file on a Cray X1 series or X2 system that has 32-bit integers and 32-bit floating-point numbers, consider the following code fragments.

To swap the unformatted data and control images when accessing unformatted files created on a system with a different endian, use one of the following commands:

```
assign -N swap_endian u:unit
assign -N swap_endian f:filename
```

Existing program:

```
REAL         RVAL      ! Default size (32-bits)
INTEGER      IVAL      ! Default size (32-bits)
...
READ (1) IVAL, RVAL
```

This program will expect both the integer and floating-point data to be the same size (32 bits). However, it can be modified to explicitly declare the variables to be the same size as the expected data.

Modified program (#1):

```
REAL    (KIND=4) RVAL      ! Explicit 32-bits
INTEGER (KIND=4) IVAL      ! Explicit 32-bits
...
READ (1) IVAL, RVAL
```

This program will correctly read the expected data. However, if this type of modification is too extensive, only the variables used in the I/O statement list need be modified.

Modified program (#2):

```
REAL             RVAL      ! Default size (32-bits)
INTEGER          IVAL      ! Default size (32-bits)
REAL    (KIND=8) RTMP      ! Explicit 64-bits
INTEGER (KIND=4) ITMP      ! Explicit 32-bits
...
READ (1) ITMP, RTMP  !
```

Change explicitly sized data to default sized data:

```
RVAL = RTMP
IVAL = ITMP
```

On some systems, data types can be declared as 32 bits in size and can then be read or written directly. This is the most direct and efficient method to read or write data files for Cray X1 series and X2 systems. The user can alter the declarations of the variables used in the Fortran I/O list to declare them as `KIND=4` or as `REAL*4` (or `INTEGER*4`).

Other IEEE data conversion variants are also available, but not all variants are available on all systems:

`ieee` or `ieee_32`

The default workstation conversion specification. Data sizes are based on 32-bit words.

`ieee_64` The default IEEE specification on Cray T90/IEEE and Cray T3E systems. Data sizes are based on 64-bit words.

`ieee_le` or `ultrix`

Data sizes are based on 32-bit words and are little-endian.

`mips` Data sizes are based on 32-bit words, except for 128-bit floating point data which uses a "double double" format.

`ia` IEEE data types with Intel-style little-endian.

### 18.4.4 VAX/VMS Conversion

Nine record types are supported for VAX/VMS record conversion. This includes a combination of three record types and the three types of storage medium, as defined in the following list:

| Record type | Definition |
|---|---|
| `f` | Fixed-length records |
| `v` | Variable-length records |
| `s` | Segmented records |

| Media | Definition |
|---|---|
| `tr` | For transparent access to files |
| `bb` | For unlabeled tapes and `bb` station transfers |
| `tape` | For labeled tapes |

Segmented records are mainly used by VAX/VMS Fortran. The following examples show some combinations of segmented records in different types of storage media:

| Example | Definition |
|---|---|
| `vms.s.tr` | Use as an FFIO specification to read or write a file containing segmented records with transparent access. In the `fetch` and `dispose` commands, specify the `-f tr` option for the file. |
| `vms.s.tape` | Use as an FFIO specification to read or write a file containing segmented records on a labeled tape. |
| `vms.s.bb` | Use as an FFIO specification to read or write a file containing segmented records on an unlabeled tape. In the `fetch` and `dispose` commands, specify the `-f bb` option for the file if it is not a tape. |

The VAX/VMS system stores its data as a stream of bytes on various devices. Cray X1 series and X2 systems number their bytes from the most-significant bits to the least-significant bits, while the VAX system numbers the bytes from lowest significance up. The Cray X1 series and X2 systems make this byte-ordering transparent when you use text files. When data conversion is used, byte swapping sometimes must be done.

# Named Pipe Support [19]

Named pipes, or UNIX FIFO special files for I/O requests, are created with the `mknod`(2) system call; these special files allow any two processes to exchange information. The system call creates an inode for the named pipe and establishes it as a named pipe that can be read to or written from. It can then be used by standard Fortran I/O or C I/O. Piped I/O is faster than normal I/O and requires less memory than memory-resident files.

Fortran programs can communicate with each other using named pipes. After a named pipe is created, Fortran programs can access that pipe almost as if it were a normal file. The unique aspects of process communication using named pipes are discussed in the following list; the examples show how a Fortran program can use standard Fortran I/O on pipes:

- A named pipe must be created before a Fortran program opens it. The following syntax for the command creates a named pipe called `fort.13`. The `p` argument makes it a pipe.

  ```
  /bin/mknod fort.13 p
  ```

  A named pipe can be created from within a Fortran program by using the `pxfsystem` function. The following example creates a named pipe:

  ```
  INTEGER ILEN,IERROR
  ILEN=0
  CALL PXFSYSTEM ('/bin/mknod fort.13 p',ILEN,IERROR)
  ```

- Fortran programs can use two named pipes: one to read and one to write. A Fortran program can read from or write to any named pipe, but it cannot do both at the same time. This is a Fortran restriction on pipes, not a system restriction. It occurs because Fortran does not allow read and write access at the same time.

- I/O transfers through named pipes use memory for buffering. A separate buffer is created for each named pipe. The `PIPE_BUF` parameter defines the kernel buffer size in the `/sys/param.h` parameter file. The default value of `PIPE_BUF` is 8 blocks (`8 * 512` words), but the full size may not be needed or used.

  I/O to named pipes does not transfer to or from a disk. However, if I/O transfers fill the buffer, the writing process waits for the receiving process to read the data before refilling the buffer. If the size of the `PIPE_BUF` parameter is increased, I/O performance may decrease because of buffer contention.

If memory has already been allocated for buffers, more space will not be allocated.

- Binary data transferred between two processes through a named pipe must use the correct file structure. An undefined file structure (specified by `assign -s u`) should be specified for a pipe by the sending process. An unblocked structure (specified by `assign -s unblocked`) should be specified for a pipe by the receiving process.

  You can also select a file specification of `system` (`assign -F system`) for the sending process.

  The file structure of the receiving or read process can be set to either an undefined or an unblocked file structure. However, if the sending process writes a request that is larger than `PIPE_BUF`, it is essential for the receiving process to read the data from a pipe set to an unblocked file structure. A read of a transfer larger than `PIPE_BUF` on an undefined file structure yields only the amount of data specified by `PIPE_BUF`. The receiving process does not wait to see whether the sending process is refilling the buffer. The pipe may be less than the value of `PIPE_BUF`.

  For example, the following `assign` commands specify that the file structure of the named pipe (unit 13, file name `pipe`) for the sending process should be `undefined (-s u)`. The named pipe (unit 15, file name `pipe`) is type `unblocked (-s unblocked)` for the read process.

  ```
  assign -s u -a pipe u:13
  assign -s unblocked -a pipe u:15
  ```

- A read from a pipe that is closed by the sender causes an end-of-file (EOF). To detect EOF on a named pipe, the pipe must be opened as read-only by the receiving process. The remainder of this chapter presents more information about detecting EOF.

## 19.1 Piped I/O Example without End-of-file Detection

In this example, two Fortran programs communicate without end-of-file (EOF) detection. Program `writerd` generates an array, which contains the elements 1 to 3, and writes the array to named pipe `pipe1`. Program `readwt` reads the three elements from named pipe `pipe1`, prints out the values, adds `1` to each value, and writes the new elements to named pipe `pipe2`. Program `writerd` reads the new values from named pipe `pipe2` and prints them. The `-a` option of the `assign` command allows the two processes to access the same file with different `assign` characteristics.

**Example 8: No EOF Detection: program `writerd`**

```
       program writerd
        parameter(n=3)
        dimension ia(n)
        do 10 i=1,n
           ia(i)=i
10      continue
        write (10) ia
        read (11) ia
        do 20 i=1,n
           print*,'ia(',i,') is ',ia(i),' in writerd'
20      continue
        end
```

**Example 9: No EOF Detection: program `readwt`**

```
       program readwt
        parameter(n=3)
        dimension ia(n)
        read (15) ia
        do 10 i=1,n
           print*,'ia(',i,') is ',ia(i),' in readwt'
           ia(i)=ia(i)+1
10      continue
        write (16) ia
        end
```

The following command sequence executes the programs:

```
ftn -o readwt readwt.f
ftn -o writerd writerd.f
/bin/mknod pipe1 p
/bin/mknod pipe2 p
assign -s u -a pipe1 u:10
assign -s unblocked -a pipe2 u:11
assign -s unblocked -a pipe1 u:15
assign -s u -a pipe2 u:16
readwt &
writerd
```

This is the output of the two programs:

```
ia(1) is 1 in readwt
ia(2) is 2 in readwt
ia(3) is 3 in readwt
ia(1) is 2 in writerd
ia(2) is 3 in writerd
ia(3) is 4 in writerd
```

## 19.2 Detecting End-of-file on a Named Pipe

The following conditions must be met to detect end-of-file on a read from a named pipe within a Fortran program:

- The program that sends data must open the pipe in a specific way, and the program that receives the data must open the pipe as read-only.
- The program that sends or writes the data must open the named pipe as read and write or write-only. Read and write is the default because the `/bin/mknod` command creates a named pipe with read and write permission.
- The program that receives or reads the data must open the pipe as read-only. A read from a named pipe that is opened as read and write waits indefinitely for the data.

## 19.3 Piped I/O Example with End-of-file Detection

This example uses named pipes for communication between two Fortran programs with end-of-file detection. The programs in this example are similar to the programs used in the preceding section. This example shows that program `readwt` can detect the EOF.

Program `writerd` generates array `ia` and writes the data to the named pipe `pipe1`. Program `readwt` reads the data from the named pipe `pipe1`, prints the values, adds one to each value, and writes the new elements to named pipe `pipe2`. Program `writerd` reads the new values from `pipe2` and prints them. Finally, program `writerd` closes `pipe1` and causes program `readwt` to detect the EOF.

This command sequence executes these programs:

```
ftn -o readwt readwt.f
ftn -o writerd writerd.f
assign -s u -a pipe1 u:10
assign -s unblocked -a pipe2 u:11
assign -s unblocked -a pipe1 u:15
assign -s u -a pipe2 u:16
/bin/mknod pipe1 p
/bin/mknod pipe2 p
readwt &
writerd
```

**Example 10: EOF Detection: program `writerd`**

```
      program writerd
       parameter(n=3)
       dimension ia(n)
       do 10 i=1,n
         ia(i)=i
10     continue
       write (10) ia
       read (11) ia
       do 20 i=1,n
         print*,'ia(',i,') is',ia(i),' in writerd'
20     continue
       close (10)
       end
```

**Example 11: EOF Detection: program `readwt`**

```
      program readwt
       parameter(n=3)
       dimension ia(n)
C      open the pipe as read-only
       open(15,form='unformatted', action='read')
       read (15,end = 101) ia
       do 10 i=1,n
         print*,'ia(',i,') is ',ia(i),' in readwt'
         ia(i)=ia(i)+1
 10    continue
       write (16) ia
       read (15,end = 101) ia
       goto 102
```

```
101   print *,'End of file detected'
102   continue
      end
```

This is the output of the two programs:

```
ia(1) is 1 in readwt
ia(2) is 2 in readwt
ia(3) is 3 in readwt
ia(1) is 2 in writerd
ia(2) is 3 in writerd
ia(3) is 4 in writerd
End of file detected
```

# Glossary

**absolute address**

1. A unique, explicit identification of a memory location, a peripheral device, or a location within a peripheral device. 2. A precise memory location that is an actual address number rather than an expression from which the address can be calculated.

**accelerated mode**

One of two modes of execution for an application on UNICOS/mp systems; the other mode is flexible mode. Applications running in accelerated mode perform in a predictable period of processor time, though their wall clock time may vary depending on I/O usage, network use, and/or whether any oversubscription occurs on the relevant nodes. Due to the characteristics of the memory address space, accelerated applications must run on logically contiguous nodes. See also *flexible mode*.

**application node**

For UNICOS/mp systems, a node that is used to run user applications. Application nodes are best suited for executing parallel applications and are managed by the strong application placement scheduling and gang scheduling mechanism Psched. See also *node*; *node flavor*.

**array assignment statement**

See *array syntax statement*.

**array syntax statement**

A Fortran statement that allows you to use 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. For example, the A = B array syntax statement assigns all the values of array A to array B. Sometimes called an array assignment statement.

**assign environment**

The set of information used in Fortran to alter the details of a Fortran connection. This information includes a list of unit numbers, file names, and file name patterns that have attributes associated with them. Any file name, file name pattern, or unit number to which assign options are attached is called an assign

object. When the unit or file is opened from Fortran, the options are used to set up the properties of the connection.

**asynchronous I/O**

I/O operation during which the program performs other operations that do not involve the data in the I/O operation. Control is returned to the calling program after the I/O is initiated. The program may perform calculations unrelated to the previous I/O request, or it may issue another unrelated I/O request while waiting for the first I/O request to complete. An operation is complete when all data has been moved.

**barrier**

An obstacle within a program that provides a mechanism for synchronizing tasks. When a task encounters a barrier, it must wait until all specified tasks reach the barrier.

**barrier synchronization**

1. An event initiated by software that prevents cooperating tasks from continuing to issue new program instructions until all of the tasks have reached the same point in the program. 2. A feature that uses a barrier to synchronize the processors within a partition. All processors must reach the barrier before they can continue the program.

**basic block**

A section of a program that does not cross any conditional branches, loop boundaries, or other transfers of control. There is a single entry point and a single exit point. Many compiler optimizations occur within basic blocks.

**binary blocked**

A file format that describes blocked, nontranslatable data.

**binary stream**

An ordered sequence of characters that can transparently record internal data. Data read in from a binary stream equals data that was written earlier out to that stream under the same implementation.

**binding**

The way in which one component in a resource specification is related to another component.

**block data**

A type of Fortran program unit. A block data program unit contains only data definitions. It specifies initial values for a restricted set of data objects.

**blocking**

An optimization that involves changing the iteration order of loops that access large arrays so that groups of array elements are processed as many times as possible while they reside in cache.

**C interoperability**

A Fortran feature that allows Fortran programs to call C functions and access C global objects and also allows C programs to call Fortran procedures and access Fortran global objects.

**cache line**

A division of cache. Each cache line can hold multiple data items. For Cray X1 and X2 systems, a cache line is 32 bytes, which is the maximum size of a hardware message.

**co-array**

A syntactic extension to Fortran that offers a method for programming data passing; a data object that is identically allocated on each image and can be directly referenced syntactically by any other image.

**co-dimensions**

The dimensions of a co-array; specified within brackets ([ ]). A co-array specification consists of the local object specification and the co-dimensions specification.

**common block**

An area of memory, or block, that can be referenced by any program unit. In Fortran, a named common block has a name specified in a Fortran COMMON or TASKCOMMON statement, along with specified names of variables or arrays

stored in the block. A blank common block, sometimes referred to as blank common, is declared in the same way but without a name.

**compute module**

For a Cray X1 and X2 series mainframes, the physical, configurable, scalable building block. Each compute module contains either one node with 4 MCMs/4MSPs (Cray X1 modules) or two nodes with 4 MCMs/8MSPs (Cray X1E modules). Sometimes referred to as a node module. See also *node*.

**construct**

A sequence of statements in Fortran that starts with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with the corresponding terminal statement.

**Cray Fortran Compiler**

The compiler that translates Fortran programs into Cray object files. The Cray Fortran Compiler fully supports the Fortran language through the Fortran 2003 Standard, ISO/IEC 1539-1:2004.

**Cray pointee**

See *Cray pointer*.

**Cray pointer**

A variable whose value is the address of another entity, which is called a pointee. The Cray pointer type statement declares both the pointer and its pointee. 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.

**Cray Programming Environment Server (CPES)**

A server for the Cray X1 and X2 series systems that runs the Programming Environment software.

**Cray streaming directives (CSDs)(X1 only)**

Nonadvisory directives that allow you to more closely control multistreaming for key loops.

**Cray X1 series system**

The Cray system that combines the single-processor performance and

single-shared address space of Cray parallel vector processor (PVP) systems with the highly scalable microprocessor-based architecture that is used in Cray T3E systems. Cray X1 and Cray X1E systems utilize powerful vector processors, shared memory, and a modernized vector instruction set in a highly scalable configuration that provides the computational power required for advanced scientific and engineering applications.

**CrayDoc**

Cray's documentation system for accessing and searching Cray books, man pages, and glossary terms from a web browser.

**CrayPat**

For Cray X1 and X2 series systems, the primary high-level tool for identifying opportunities for optimization. CrayPat allows you to perform profiling, sampling, and tracing experiments on an instrumented application and to analyze the results of those experiments; no recompilation is needed to produce the instrumented program. In addition, the CrayPat tool provides access to all hardware performance counters.

**data passing**

Transferring data from one object to another; useful for programming single-program-multiple-data (SPMD) parallel computation. Its chief advantage over message passing is lower latency for data transfers, which leads to better scalability of parallel applications. Data passing can be achieved by using SHMEM library routines or by using co-arrays.

**deferred implementation**

The label used to introduce information about a feature that will not be implemented until a later release.

**direct-access I/O**

I/O operation where the a peripheral device or a channel controls data transfer in and out of the computer. The data transfers directly to or from storage and bypasses the processor.

**dynamic thread adjustment**

In OpenMP, the automatic adjustment of the number of threads between parallel regions. Also known as dynamic threads or the dynamic thread mechanism.

**entry point**

A location in a program or routine at which execution begins. A routine may have several entry points, each serving a different purpose. Linkage between program modules is performed when the linkage editor binds the external references of one group of modules to the entry points of another module.

**environment variable**

A variable that stores a string of characters for use by your shell and the processes that execute under the shell. Some environment variables are predefined by the shell, and others are defined by an application or user. Shell-level environment variables let you specify the search path that the shell uses to locate executable files, the shell prompt, and many other characteristics of the operation of your shell. Most environment variables are described in the ENVIRONMENT VARIABLES section of the man page for the affected command.

**Etnus TotalView**

A symbolic source-level debugger designed for debugging the multiple processes of parallel Fortran, C, or C++ programs.

**explicit data conversion**

The process by which the user performs calls to subroutines that convert native data to and from foreign data formats.

**flexible file I/O (FFIO)**

A method of I/O, sometimes called layered I/O, wherein each processing step requests one I/O layer or grouping of layers. A layer refers to the specific type of processing being done. In some cases, the name corresponds directly to the name of one layer. In other cases, however, specifying one layer invokes the routines used to pass the data through multiple layers.

**flexible mode**

One of two modes of execution for an application on UNICOS/mp systems; the other mode is accelerated mode. Applications running in flexible mode may run on noncontiguous nodes; they perform in a less predictable amount of processor time than applications running in accelerated mode due to the exclusive use of source processor address translation. See also *accelerated mode*.

**folding**

A basic compiler optimization that converts operations on constants to simpler forms as these examples show: `Operation to fold Folded operation 1 + 2 3 5.0/3.0 + 1.7 3.366...  (if the -O fp1 (Fortran) or -h fp1 (C/C++) or greater is used.)  sin( 1.3 ) 0.96355818... 3 + n - 4 n - 1`

**formatted I/O**

Data transfer with editing. Formatted I/O can be edit-directed, list-directed, or namelist I/O. If the format identifier is an asterisk, the I/O statement is a list-directed I/O statement. All other format identifiers indicate edit-directed I/O. Formatted I/O should be avoided when I/O performance is important.

**gather/scatter**

An operation that copies data between remote and local memory or within local memory. A gather is any software operation that copies a set of data that is nonsequential in a remote (or local) processor, usually storing into a sequential (contiguous) area of local processor memory. A scatter copies data from a sequential, contiguous area of local processor memory) into nonsequential locations in a remote (or local) memory.

**implicit data conversion**

The process by which you declare that a particular file contains foreign data and/or record blocking and then request that the run-time library perform appropriate transformations on the data to make it useful to the program at I/O time.

**implicit open**

The opening of a file or a unit when the first reference to a unit number is an I/O statement other than OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, or REWIND.

**invariant**

A rule, such as the ordering of an ordered list or heap, that applies throughout the life of a data structure or procedure. Each change to the data structure must maintain the correctness of the invariant.

**kind**

Data representation (for example, single precision, double precision). The kind of a type is referred to as a kind parameter or kind type parameter of the type. The kind type parameter `KIND` indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types.

**layered I/O**

See *flexible file I/O (FFIO)*.

**lexical extent**

In OpenMP, statements that reside within a structured block. See also *structured block*.

**list-directed I/O**

I/O where the records consist of a sequence of values separated by value separators such as commas or spaces. A tab is treated as a space in list-directed input, except when it occurs in a character constant that is delimited by apostrophes or quotation marks.

**lock**

1. Any device or algorithm that is used to ensure that only one process will perform some action or use some resource at a time. 2. A synchronization mechanism that, by convention, forces some data to be accessed by tasks in a serial fashion. Locks have two states: locked and unlocked. 3. A facility that monitors critical regions of code.

**loop collapse**

An optimization that combines loop interchange and loop fusion to convert a loop nest into a single loop, with an iteration count that is the product of the iteration counts of the original loops.

**loop fusion**

An optimization that takes the bodies of loops with identical iteration counts and fuses them into a single loop with the same iteration count.

**loop interchange**

An optimization that changes the order of loops within a loop nest, to achieve stride minimization or eliminate data dependencies.

**loop invariant**

A value that does not change between iterations of a loop.

**loop unrolling**

An optimization that increases the step of a loop and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and reduce memory access time.

**master thread**

The thread that creates a team of threads when an OpenMP parallel region is entered.

**Message Passing Interface (MPI)**

A widely accepted standard for communication among nodes that run a parallel program on a distributed-memory system. MPI is a library of routines that can be called from Fortran, C, and C++ programs.

**module file**

A metafile that defines information specific to an application or collection of applications. (This term is not related to the module statement of the Fortran language; it is related to setting up the Cray system environment.) For example, to define the paths, command names, and other environment variables to use the Programming Environment for Cray systems, you use the module file `PrgEnv`, which contains the base information needed for application compilations. The module file `mpt` sets a number of environment variables needed for message passing and data passing application development.

**multistreaming processor (MSP) (X1 only)**

For UNICOS/mp systems, a basic programmable computational unit. Each MSP is analogous to a traditional processor and is composed of four single-streaming processors (SSPs) and E-cache that is shared by the SSPs. See also *node*.

**multithreading**

The concurrent use of multiple threads of control that operate within the same address space.

**named pipe**

A first-in, first-out file that allows communication between two unrelated processes running on the same host.

**namelist I/O**

I/O that allows you to group variables by specifying a namelist group name. On input, any namelist item within that list may appear in the input record with a value to be assigned. On output, the entire namelist is written.

**NaN**

An IEEE floating-point representation for the result of a numerical operation that cannot return a valid number value; that is, not a number, NaN.

**node**

For UNICOS/mp systems, the logical group of four multistreaming processors (MSPs), cache-coherent shared local memory, high-speed interconnections, and system I/O ports. A Cray X1 system has one node with 4 MSPs per compute module. A Cray X1E system has two nodes of 4 MSPs per node, providing a total of 8 MSPs on its compute module. Software controls how a node is used: as an OS node, application node, or support node. See also *compute module*; .

**node**

In networking, a processing location. A node can be a computer (host) or some other device, such as a printer. Every node has a unique network address.

**node flavor**

For UNICOS/mp systems, software controls how a node is used. A node's software-assigned flavor dictates the kind of processes and threads that can use its resources. The three assignable node flavors are application, OS, and support. See also *application node*; *OS node*; *support node*; *system node*.

**OpenMP**

An industry-standard, portable model for shared memory parallel programming.

**OS node**

For UNICOS/mp systems, the node that provides kernel-level services, such as system calls, to all support nodes and application nodes. See also *node*; *node flavor*.

**overindexing**

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not always, leads to referencing a storage location outside of the entire array.

**page size**

The unit of memory addressable through the Translation Lookaside Buffer (TLB). For a UNICOS/mp system, the base page size is 65,536 bytes, but larger page sizes (up to 4,294,967,296 bytes) are also available.

**parallel processing**

Processing in which multiple processors work on a single application simultaneously.

**parallel region**

See *serial region*.

**partitioning**

Configuring a UNICOS/mp system into logical systems (partitions). Each partition is independently operated, booted, dumped, and so on without impact on other running partitions. Hardware and software failures in one partition do not affect other partitions.

**piped I/O**

I/O that uses named pipes; faster than normal I/O because it requires less memory than memory-resident files. See also *named pipe*.

**pointer**

A data item that consists of the address of a desired item.

**Psched**

The UNICOS/mp application placement scheduling tool. The psched command

can provide job placement, load balancing, and gang scheduling for all applications placed on application nodes.

**rank**

The number of dimensions in a Fortran array. Rank is declared when the array is declared and cannot change.

**reduction**

The process of transforming an expression according to certain reduction rules. The most important forms are beta reduction (application of a lambda abstraction to one or more argument expressions) and delta reduction (application of a mathematical function to the required number of arguments). An evaluation strategy (or reduction strategy) determines which part of an expression to reduce first. There are many such strategies. Also called contraction.

**reduction loop**

A loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

**scalar processing**

A form of fine-grain serial processing whereby iterative operations are performed sequentially on the elements of an array, with each iteration producing one result.

**scoping unit**

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

**serial region**

An area within a program in which only the master task is executing. Its opposite is a parallel region.

**SHMEM**

A library of optimized functions and subroutines that take advantage of shared memory to move data between the memories of processors. The routines can either be used by themselves or in conjunction with another programming style such as Message Passing Interface. SHMEM routines can be called from Fortran, C, and C++ programs.

**shortloop**

A loop that is vectorized but that has been determined by the compiler to have trips less than or equal to the maximum vector length. In this case, the compiler deletes the loop to the top of the loop. If the shortloop directive is used or the trip count is constant, the top test for number of trips is deleted. A shortloop is more efficient than a conventional loop.

**side effects**

The result of modifying shared data or performing I/O by concurrent streams without the use of an appropriate synchronization mechanism. Modifying shared data (where multiple streams write to the same location or write/read the same location) without appropriate synchronization can cause unreliable data and race conditions. Performing I/O without appropriate synchronization can cause an I/O deadlock. Shared data, in this context, occurs when any object may be referenced by two or more single-streaming processors (X1 only). This includes globally visible objects (for example, `COMMON`, `MODULE` data), statically allocated objects (`SAVE`, `C static`), dummy arguments that refer to `SHARED` data and objects in the `SHARED` heap.

**single-streaming processor (SSP) (X1 only)**

For UNICOS/mp systems, a basic programmable computational unit. See also *node*.

**stack allocation**

A method of allocating memory for variables used by a called routine during program execution. Variables are reset for each invocation of a subprogram. Stack mode is required for multitasked code.

**stride**

The relationship between the layout of an array's elements in memory and the order in which those elements are accessed. A stride of 1 means that

memory-adjacent array elements are accessed on successive iterations of an array-processing loop.

**structured block**

In Fortran OpenMP, a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom. Execution must always proceed with entry at the top of the block and exit at the bottom with only one exception: the block is allowed to have a STOP statement inside a structured block. This statement has the well-defined behavior of terminating the entire program.

**support node**

For UNICOS/mp systems, the node that is used to run serial commands, such as shells, editors, and other user commands (`ls`, for example). See also *node*; *node flavor*.

**symbol table**

A table of symbolic names (for example, variables) used in a program to store their memory locations. The symbol table is part of the executable object generated by the compiler. Debuggers use it to help analyze the program.

**synchronous I/O**

I/O operation during which an executing program relinquishes control until the operation is complete. An operation is not complete until all data is moved.

**system cache**

A set of buffers in kernel memory used for I/O operations by the operating system. 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. In many cases, however, it is desirable to bypass the system cache and to perform I/O directly between the user's memory and the logical device.

**system node**

For UNICOS/mp systems, the node that is designated as both an OS node and a support node; this node is often called a system node; however, there is no node flavor of "system." See also *node*; *node flavor*.

**system time**

The amount of time that the operating system spends providing services to an application.

**thread**

The active entity of execution. A sequence of instructions together with machine context (processor registers) and a stack. On a parallel system, multiple threads can be executing parts of a program at the same time.

**TKR**

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

**trigger**

A command that a user logged into a Cray X1 series system uses to launch Programming Environment components residing on the CPES. Examples of trigger commands are `ftn`, `CC`, and `pat_build`.

**type**

A means for categorizing data. Each intrinsic and user-defined data type has four characteristics: a name, a set of values, a set of operators, and a means to represent constant values of the type in a program.

**unblocked file structure**

A file that contains undelimited records. Because it does not contain any record control words, it does not have record boundaries.

**unformatted I/O**

Transfer of binary data without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written. The unit must be an external unit.

**UNICOS/lc**

The operating system for Cray X2 series systems.

**UNICOS/mp**

The operating system for Cray X1 series (Cray X1 and Cray X1E) systems.

**unrolling**

A single-processing-element optimization technique in which the statements within a loop are copied. For example, if a loop has two statements, unrolling might copy those statements four times, resulting in eight statements. The loop control variable would be incremented for each copy, and the stride through the array would also be increased by the number of copies. This technique is often performed directly by the compiler, and the number of copies is usually between two and four.

**vector**

A series of values on which instructions operate; this can be an array or any subset of an array such as row, column, or diagonal. Applying arithmetic, logical, or memory operations to vectors is called vector processing. See also *vector processing*.

**vector length**

The number of elements in a vector.

**vector processing**

A form of instruction-level parallelism in which the vector registers are used to perform iterative operations in parallel on the elements of an array, with each iteration producing up to 64 simultaneous results. See also *vector*.

**vector register**

The register that serves as a source and destination for vector operations.

**vectorization**

The process, performed by the compiler, of analyzing code to determine whether it contains vectorizable expressions and then producing object code that uses the vector unit to perform vector processing.

# Index

**D**