CF90™ Commands and Directives Reference Manual

SR–3901 3.1

Document Number 004–3901–001

New Features

CF90™ Commands and Directives Reference Manual 004–3901–001

This manual describes the commands and directives supported by the CF90 compiler, release 3.1.

Additions to this manual to support the CF90 3.1 release include the following:

- The `CNCALL` and `PERMUTATION` directives have been implemented with `!DIR$` prefixes.
- Compiler directives and environment variables to support the OpenMP Fortran API. Note that comments in your source code may need to be modified. For more information, see Section 2.2.20.27, page 39.
- Documentation that describes the application of certain compiler directives to instructions that involve array syntax.
- The `-G 2` option to the `f90`(1) command supports post mortem debugging.
- The `-Z` option to the `f90`(1) command enables recognition of co-array syntax.
- Miscellaneous corrections and additions.

With the implementation of OpenMP features, the terminology surrounding Autotasking has changed slightly, as follows:

- The *Autotasking directives* are outmoded. With release 3.1, the compiler honors and recognizes these directives, but users creating new code are now encouraged to use the standard, portable OpenMP directives.
- *Autotasking* refers to the compiler's ability to generate parallel code automatically.
- *Tasking* refers to both *user tasking*, accomplished by inserting directives, and to Autotasking, which is generated by the compiler.

Change indicators indicate information that has changed since the 3.0.2 revision.

Record of Revision

Version	*Description*
1.0	December 1993 Original Printing. This document supports the CF90 compiler release 1.0 running on CRAY Y-MP systems.
1.1	June 1994 This document supports the CF90 compiler release 1.0 running on both CRAY Y-MP systems and on SPARC systems, including those from CRS.
2.0	November 1995 This document supports the CF90 compiler release 2.0 running on Cray PVP systems, CRAY T3E systems, and SPARC systems. The implementation of features on CRAY T3E systems is deferred.
3.0	May 1997 This printing supports the CF90 3.0 release running on UNICOS and UNICOS/mk operating systems.
3.0.1	August 1997 This online revision supports the Cray Research CF90 3.0.1 release, running on UNICOS and UNICOS/mk operating systems.
3.0.2	March 1998 This online revision supports the Cray Research CF90 3.0.2 release, running on UNICOS and UNICOS/mk operating systems.
3.1	August 1998 This printing supports the Cray Research CF90 3.1 release running on UNICOS and UNICOS/mk operating systems.

Contents

Page

Page

Page

Page

Figures

Page

Tables

About This Manual

This manual describes the commands and directives for using the Cray Research CF90 compiler, which is invoked through the `f90`(1) command. The `f90` command can also invoke a source preprocessor, a source lister, an assembler, and the loader.

The CF90 3.1 compiler runs on the following systems:

- CRAY SV1, CRAY C90, CRAY J90, CRAY T90, CRAY Y-MP, and CRAY EL systems running UNICOS 9.0 or running UNICOS 10.0 or later.
- CRAY T3E systems running UNICOS/mk 2.0.3 or later.

The CF90 compiler was developed to support the Fortran standards adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). These standards, commonly referred to as *the Fortran 90 standard*, are ANSI X3.198–1992 and ISO/IEC 1539:1991–1. Because the ANSI Fortran 90 standard is a superset of the FORTRAN 77 standard, the CF90 compiler will compile code written to the FORTRAN 77 standard.

Note: The Fortran 90 standard is a substantial revision to the FORTRAN 77 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 90 standard for Silicon Graphics and for other vendors. To maintain conformance to the Fortran 90 standard, Silicon Graphics may need to change the behavior of certain CF90 features in future releases based upon the outcome of the outstanding interpretations to the standard.

Related CF90 Publications

This manual is one of a set of manuals that describes the CF90 compiler. The other manuals in the set are as follows:

- *CF90 Ready Reference*
- *Fortran Language Reference Manual, Volume I*
- *Fortran Language Reference Manual, Volume II*
- *Fortran Language Reference Manual, Volume III*
- *CF90 Co-array Programming Manual*

CF90 Messages

You can obtain CF90 compiler message explanations by using the online `explain`(1) command.

CF90 Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the CF90 compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `grep` online, you can type `man grep`. To access `egrep` online, you can type either `man grep` or `man egrep`. Both commands display the `grep` man page on your terminal.

Related Fortran Publications

The following commercially available reference books are among those that you should consult for more information on the history of Fortran and the Fortran 90 language itself:

- Adams, J., W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook — Complete ANSI/ISO Reference*. New York, NY: Intertext Publications / Multiscience Press, Inc., 1990.
- Metcalf, M. and J. Reid. *Fortran 90 Explained*. Oxford, UK: Oxford University Press, 1990.
- American National Standards Institute. *American National Standard Programming Language Fortran*, ANSI X3.198–1992. New York, 1992.

- International Standards Organization. *ISO/IEC 1539:1991, Information technology — Programming languages — Fortran*. Geneva, 1991.

The *User Publications Catalog*, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

Related Publications

The following documents contain additional programming environment publications that may interest you:

- *Segment Loader (SEGLDR) and ld Reference Manual*
- *UNICOS User Commands Reference Manual*
- *UNICOS Performance Utilities Reference Manual*
- *Scientific Libraries Reference Manual*
- *Intrinsic Procedures Reference Manual*
- *Introducing the Program Browser*
- *Application Programmer's Library Reference Manual*
- *Application Programmer's I/O Guide*
- *Guide to Parallel Vector Applications*
- *Optimizing Code on Cray PVP Systems*
- *Compiler Information File (CIF) Reference Manual*
- *Introducing the Cray TotalView Debugger*
- *Introducing the MPP Apprentice Tool*
- *CRAY T3E Fortran Optimization Guide*

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a printed copy of this document, either call the Minnesota Distribution Center at +1–651–683–5907, or send a facsimile of your request to fax number +1–651–452–0141. Silicon Graphics employees may send electronic mail to `orderdsk@cray.com` (UNIX system users).

Silicon Graphics maintains information on publicly available Cray Research documents at the following URL:

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

This Web site contains information that allows you to browse documents online and send feedback to Silicon Graphics.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts 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 command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an SPR or PV be filed. If filing an SPR, use PUBLICATIONS for the group name, PUBS for the command, and NO-LICENSE for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–651–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–651–683–5599.

We value your comments and will respond to them promptly.

Introduction [1]

This manual is organized into the following chapters:

- Chapter 1 introduces the content of this manual and provides a general description of the compiler.
- Chapter 2, page 5, describes the `f90`(1) command, which you use to invoke the compiler. This chapter includes information about the options you can use on the `f90`(1) command line and environment variables that affect the CF90 compiler.
- Chapter 3, page 71, describes the CF90 directives that the compiler recognizes.
- Chapter 4, page 119, describes the OpenMP Fortran API directives.
- Chapter 5, page 167, describes the source preprocessor.
- Appendix A, page 177, describes the Autotasking directives, which are outmoded. Cray Research encourages you to write new codes and to update existing codes using the OpenMP Fortran API directives described in Chapter 4, page 119.

The CF90 compiler is invoked through the `f90`(1) command. The `f90`(1) command also invokes `ftnlint`(1), listers, and a loader, as follows:

- If you are running CF90 on a UNICOS system, the `f90`(1) command invokes the loader `segldr`(1).
- If you are running CF90 on a UNICOS/mk system, the `f90`(1) command invokes the loader `cld`(1).

In the most basic case, the `f90`(1) command invokes the CF90 compiler, processes the input files named on the command line, and generates a binary file. The loader loads the binary file and generates an executable output file (the default output file is `a.out`). The lister generates the program's listing file.

In the following simple example, the `f90`(1) command is used to invoke the compiler. Option `-r s` is specified to generate a source listing. File `pgm.f` is the 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, page 2, illustrates this example.

```
% f90 -r s pgm.f
% ./a.out
```

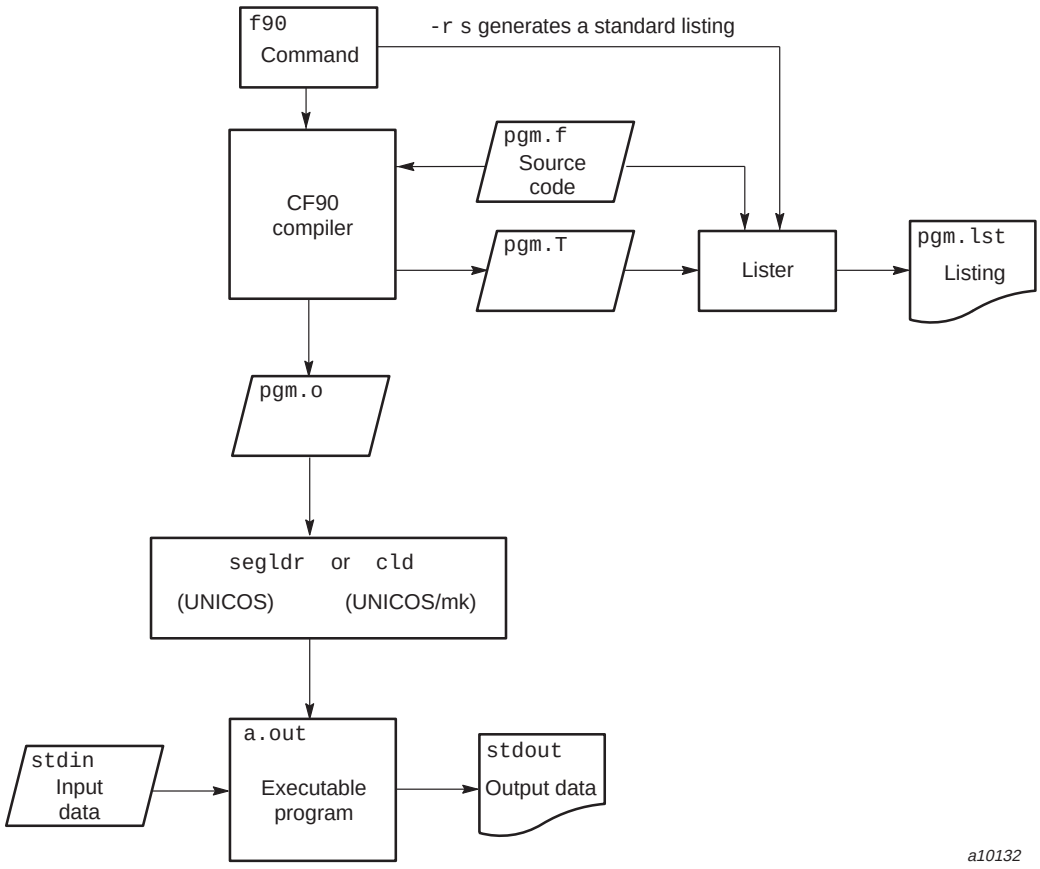

Figure 1. f90(1) command example

You can use the options on the f90(1) command line to modify the default actions; for example, you can disable the load step. For more information on f90(1) command line options, see Chapter 2, page 5.

The CF90 compiler is one of many software products that form the CF90 programming environment. This environment allows you to develop, debug, and run Fortran codes on your computer system. It includes the following products:

- Loaders. On UNICOS systems, `segldr`(1) is your loader. On UNICOS/mk systems, `cld`(1) is your loader.
- A lister, `ftnlist`(1).
- The `ftnlint`(1) utility, which checks CF90 programs for possible errors.
- A browser, `xbrowse`(1). For more information on the browser, see *Introducing the Program Browser*, publication IN–2140.
- The compiler information file (CIF) tools, which include the `cifconv`(1) command and the libraries. For more information on these see the *Compiler Information File (CIF) Reference Manual*, publication SR–2401.
- The libraries, which include functions optimized for use on Cray Research systems. Information on the individual library routines can be found in the online man pages for each routine. In addition to online man pages, the following manuals contain printed copies of the library routine man pages and other library information:

  – *Application Programmer's Library Reference Manual*, publication SR–2165

  – *Scientific Libraries Reference Manual*, publication SR–2081

  The CF90 intrinsic procedures are implemented within the math library (`libm`), within `libfi`, and within the compiler itself. The *Intrinsic Procedures Reference Manual*, publication SR–2138, contains copies of the online man pages for all the intrinsic procedures.
- The performance tools, which can help you analyze program performance. The performance tools include `apprentice`(1), `atexpert`(1), and `pat`(1). Information on the individual tools can be found in the online man pages for each tool. The *Guide to Parallel Vector Applications*, publication SG–2182, describes several tools, and more information on the MPP Apprentice tool can be found in *Introducing the MPP Apprentice Tool*, publication IN–2511.
- Online documentation utilities. The `man`(1) command allows you to retrieve online man pages. Prose reference text, such as this manual, can be retrieved online through the WWW browser supported at your site. Contact your support staff for specific information on retrieving information in this manner.
- A debugger. TotalView includes standard debugging capabilities, such as allowing you to step through code and set breakpoints. It can be used when you invoke the CF90 compiler and direct it to produce symbol tables. The `-g` and `-G` *debug* options to the `f90`(1) command line generate symbol tables.

Invoking CF90 [2]

This chapter describes the f90(1) command, which invokes the CF90 compiler. This chapter also describes environment variables used to determine shell characteristics when working with the CF90 compiler. The f90(1) online man page contains information from this chapter in an abbreviated form.

2.1 Setting Up the CF90 Programming Environment

The installation and configuration of the CF90 programming environment uses a utility called *modules*, which is provided and installed as part of the release package. File /opt/ctl/doc/README was distributed in the release package. It contains information on initializing the module(1) command and initializing the environment.

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

```
% module load modules PrgEnv
```

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

2.2 The f90(1) Command

The f90(1) command invokes the CF90 compiler. The syntax of the command is as follows:

```
f90 [-a alloc] [-A module_name[,module_name] ...] [-b bin_obj_file]
   [-c] [-C cifopts] [-d disable]
   [-D identifier[=value][,identifier[=value]] ...] [-e enable]
   [-f source_form] [-F] [-g] [-G debug_lvl] [-i 32] [-I incldir]
   [-l lib] [-L dir] [-m msg_lvl] [-M msgs] [-N col] [-o out_file]
   [-O opt[,opt] ...] [-p module_site] [-r list_opt] [-R runchk]
   [-s size] [-S asm_file] [-t num] [-T] [-U identifier[,identifier] ...]
   [-v] [-V] [-Wa"assembler_opt"] [-Wl"loader_opt"] [-Wp"srcpp_opt"]
   [-Wr"ftnlist_opt"] [-x dirlist] [-X npes] [-Z] [--]
   file.suffix[90] [file.suffix[90] ...]
```

Note: Some options are not available on all platforms. If you specify an option that is not supported on your platform, a message is issued and, when possible, compilation continues.

Note: Some default values shown for `f90`(1) command options may have been changed by your site. See your system support staff for further details.

2.2.1 `-a` *alloc*

The `-a` *alloc* option allows you to specify a storage allocation mechanism. Enter one of the following for *alloc*:

alloc	Action
`pad`[*n*] (UNICOS/mk systems only)	Pads all local static storage and most common blocks. This option does not pad common blocks that contain data that has been storage associated with an `EQUIVALENCE` statement. This option adds padding following each element in a common block. The padding improves single-PE performance by reducing primary and secondary cache conflict between elements within the same common block. If *n* is not specified, a fixed amount of padding is added after each common block element. The size of the padding that is added depends on the size of the preceding element. If *n* is specified, padding is added in 8-byte words according to the value you specify. For example, specifying `-a pad1` directs the system to add one 8-byte word after each element in a common block. Specifying `-a pad` causes the compiler to generate common blocks that do not follow standard sequence association rules. Programs that rely on data in different elements of a common block to be a certain distance apart may behave differently when compiled with `-a pad`.

Warning: When the compiler automatically generates padding, you must ensure that:

- The common blocks are declared identically throughout the program and all program files are compiled with `-a pad`.
- Arrays in common are always referenced within the range of their declared bounds.
- The same version of the compiler is used to compile all subroutines and modules using this feature.

Failure to follow the preceding rules can cause incorrect program behavior. Typically, `cld`(1) detects these errors. To ensure that you receive all messages related to this condition, compile with the `-Wl"-D msglevel=CAUTION"` option, which causes the compiler to generate messages that caution the use of common blocks of different sizes.

`taskcommon` (UNICOS systems only)

Converts all common blocks in the compilation to taskcommon blocks. Local variables named in SAVE statements are stored in taskcommon, causing each processor (task) to have a private copy of saved variables. This includes variables initialized in DATA statements or data declaration statements. For a given procedure, a COMMON compiler directive can override this command line specification. For more information on the COMMON compiler directive, see Section 3.6.3, page 110. This option is typically used when using the Message Passing Toolkit (MPT) on UNICOS systems. For more information on MPT, see the *Message Passing Toolkit: PVM Programmer's Manual*, and the *Message Passing Toolkit: MPI Programmer's Manual*.

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

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

2.2.3 -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*.

> **Note:** The -c option should not be specified with -b *bin_obj_file* because -c specifies that the binary file is to be saved in *file*.o. The -b *bin_obj_file* option specifies that the binary file is to be saved in *bin_obj_file*.

Only one input file is allowed when the -b 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 options are specified, the binary version of your program is not saved after the load is completed.

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.

2.2.4 -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.

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

2.2.5 -C *cifopts*

The -C *cifopts* option creates a compiler information file (CIF) and places it in *file*.T, where *file* is the name of the source file and .T is the suffix used. If both the -r and -C options are specified, -r overrides -C.

By default, the f90(1) command does not create a CIF. The -C option is used to create a CIF, which is required by various tools that analyze CF90 programs, such as ftnlint(1), ftnlist(1), and xbrowse(1). The user interface to the CIF information is through the CIF library, libcif.

The most common specification is -C a, which writes all CIF information possible. For information on the *cifopts* available, see the *Compiler Information File (CIF) Reference Manual*.

2.2.6 `-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, `-e af`. Enter one or more of the following *args* for *disable* or *enable*:

Table 1. Compiling options

args	Action, if enabled	Operating system
`0`	Initializes all undefined local stack variables to 0 (zero). If a user variable is of type character, it is initialized to NUL. The variables are initialized upon execution of each procedure. Enabling this option can help identify problems caused by using uninitialized numeric and logical variables. Also see the `-e i` option. A message is generated if you specify both `-e 0` and `-e i` on the command line; the rightmost option specified overrides the other. Disabled by default.	UNICOS UNICOS/mk
`a`	Aborts compilation after encountering the first error. Disabled by default.	UNICOS UNICOS/mk
`A`	Generates code necessary to use the MPP Apprentice tool. Disabled by default.	UNICOS/mk
`B`	Generates binary output. If disabled, inhibits all optimization and allows only syntactic and semantic checking. Enabled by default.	UNICOS UNICOS/mk
`f`	Generates an output file during execution that is suitable for processing by `perfview`(1) or `flowview`(1). To use `perfview`(1), you must also specify `-l perf` to ensure that the proper libraries are included at load time. See `flowtrace`(7) and `perftrace`(7) for more information. Disabled by default.	UNICOS
`i`	Generates a run-time error when an uninitialized local real or integer variable is used in a floating-point operation or array subscript. This option causes allocated but uninitialized local stack storage to be set to an undefined value. When the `-e i` option is specified, variables are reset for each invocation of a subprogram. Therefore, a `SAVE` statement is needed to preserve the value of a variable between invocations. This option does not apply to statically allocated uninitialized variables (data specified on a `COMMON`, `MODULE`, or `SAVE` statement), to dummy arguments, or to `ALLOCATABLE` arrays. Also see the `-e 0` option. A message is generated if you specify both `-e 0` and `-e i` on the command line; the rightmost option specified overrides the other. The loaders offer the `-f` option for uninitialized statically allocated variables. The `-f` loader option is related to this feature in that `-e i` and `-f` both preset uninitialized data to undefined values; `-e i` applies to stack data, and `-f` applies to static data. Also see the `-f` option on `segldr`(1) and the `-D` *preset*= option on `cld`(1). Disabled by default.	UNICOS UNICOS/mk

args	Action, if enabled	Operating system
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. Disabled by default.	UNICOS UNICOS/mk
j	Executes DO loops at least once. Disabled by default.	UNICOS
n	Generates messages to note all nonstandard Fortran 90 usage. Disabled by default.	UNICOS UNICOS/mk
p	Enables double precision arithmetic. When 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. Double precision arithmetic is not supported on UNICOS/mk systems. Enabled by default.	UNICOS UNICOS/mk
P	Performs source preprocessing on *file*.f[90] or *file*.F[90] but does not compile. When specified, source code is included by #include directives but not by Fortran 90 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 on source preprocessing, see Chapter 5, page 167. Disabled by default.	UNICOS UNICOS/mk
q	Aborts compilation if 100 or more errors are generated. Enabled by default.	UNICOS UNICOS/mk
r	Rounds multiplication results. Enabled by default.	UNICOS
R	Compiles all functions and subroutines as if they contained a RECURSIVE keyword. Disabled by default.	UNICOS UNICOS/mk
S	Generates assembly language output and saves it in *file*.s. When the -e S option is specified on the command line with the -S *asm_file* option, the -S *asm_file* option overrides the -e S option. Disabled by default.	UNICOS UNICOS/mk

args	Action, if enabled	Operating system
t	Determines the memory allocation method used for automatic variables. An *automatic variable* is a variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length). Storage is allocated for automatic variables and array temporaries upon entry to a procedure. Storage is deallocated upon exit from the procedure. When disabled, automatic variables are allocated memory on the heap. When enabled, the compiler attempts to limit the amount of memory requested from the heap for automatic variables; that is, an attempt is made to allocate storage for them on the stack. Generally, this reduces system call overhead. If not enough stack memory is available, the heap is used. By default, the CF90 compiler requests 16,000 words of stack space for automatic arrays and array temporaries, with a stack increment size of 8,000 words. Initial size may be increased, and increment size may be increased or decreased through directives passed to the loader. Initial stack and increment sizes can affect performance because of the number of stack overflows that might occur. Multitasking can increase the occurrences of stack overflows and decrease optimal performance. Stack overflow occurrences can be minimized and performance increased by tuning initial stack and increment values. Enabled by default.	UNICOS

args	Action, if enabled	Operating system

u	Rounds floating-point division, so quotients are the exact result when truncated to an integer if the correct quotient was a whole number. Also see `-O ieeeconform` and `-O noieeconform` in Section 2.2.20.9, page 28.
	When disabled, faster code sequences are generated for floating-point divides. This can result in slightly less accurate results because when results of floating-point divides are later assigned to integer variables, truncation errors can occur. For example, the following code fragment may result in the value of `i`, an integer, being truncated to an unexpected value:
	`x = 6.0`
	`r = x/3.0`
	`...`
	`i = r`
	On UNICOS systems, excluding CRAY T90 systems that support IEEE floating-point arithmetic, the result of the division may yield a result of 1.99999.... When assigned to `i`, the value is truncated, which produces a value of 1 instead of the expected result of 2.
	On UNICOS/mk systems and CRAY T90 systems that support IEEE floating-point arithmetic, floating-point division may not produce correctly rounded results when disabled because of the faster divide sequence that is used. This may affect the results when converting to an integer value or when comparing results with those obtained from another system that supports IEEE floating-point arithmetic.
	When enabled, additional instructions, and thus, slower code, are generated that produce more accurate results. On systems that support IEEE floating-point arithmetic, a true IEEE divide sequence is generated (instead of possibly calculating a reciprocal and multiply sequence). On UNICOS systems, other than CRAY T90 systems that support IEEE floating-point arithmetic, results of floating-point divides are rounded by adding a floating point amount and clearing the lower four bits of the mantissa. Disabled by default.
		UNICOS
		UNICOS/mk

args	Action, if enabled	Operating system
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, 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 declared with the RECURSIVE attribute. The STACK compiler directive overrides -e v; for more information on this compiler directive, see Section 3.6.4, page 111. Disabled by default.	UNICOS UNICOS/mk
X	Generates additional run-time code needed to support the Autotasking Expert System, ATExpert. To use ATExpert, you must have specified either -O 3, -O task2, or -O task3 on the command line, or you must have included OpenMP (!$OMP) or Autotasking (!MIC$) directives in your source code and compiled with -O task1, -O task2, or -O task3. See atexpert(1) for more information about ATExpert. Disabled by default.	UNICOS
Z	Performs source preprocessing and compilation on *file*.f[90] or *file*.F[90]. When specified, source code is included by #include directives but not by Fortran 90 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 on source preprocessing, see Chapter 5, page 167. Disabled by default.	UNICOS UNICOS/mk

2.2.7 -D *identifier*[=*value*][,*identifier*[=*value*]]...

The -D *identifier*[=*value*][,*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 on the -U option, see Section 2.2.28, page 54.

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

- The Fortran input source file is specified as either *file*.F or *file*.F90.

- The `-e P` or `-e Z` options have been specified.

For more information on source preprocessing, see Chapter 5, page 167.

2.2.8 `-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.

The default source form is `fixed` for input files that end with a `.f` or `.F` suffix. The default source form is `free` for input files that end with a `.f90` or `.F90` suffix.

If the file ends in `.F` or `.F90`, the source preprocessor is invoked.

2.2.9 `-F`

The `-F` option enables macro expansion throughout the source file. Typically, macro expansion occurs only on source preprocessing directive lines.

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

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

For more information on source preprocessing, see Chapter 5, page 167.

2.2.10 `-g`

The `-g` option provides debugging support identical to specifying the `-G 0` option.

2.2.11 `-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. Enter one of the following for *debug_lvl*:

debug_lvl	Support
`0`	Default debugging support. Breakpoints can be set at each line. This level of debugging is supported when optimization is disabled (when `-O 0`, `-O scalar0`, `-O task0`, and `-O vector0` are in effect). If `-G 0` has been specified on the command line along with an optimization level other than `-O 0`, `-O scalar0`, `-O task0`, or `-O vector0`, the compiler issues a message and disables most optimization. On UNICOS systems, 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. 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. Only user tasking, enabled through `!$OMP` or `!MIC$` directives, is performed. On UNICOS systems, 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. All symbol table information is provided in a format suitable for `debugview`(1). 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.

2.2.12 `-i 32`

The `-i 32` option specifies 32-bit integer arithmetic for default integers. Specifying 32-bit integer arithmetic enables the compiler to generate faster code for some expressions. For more information on generating faster code, also see the `-O fastint` option in Section 2.2.20.7, page 26.

If a module is compiled separately from a program unit that uses the module, they both must be compiled with `-i 32`. If program units are compiled separately and linked, they must all be compiled with `-i 32`.

If specifying `-i 32`, see the `-s` *size* option, Section 2.2.24, page 52, for information on conformance to the Fortran 90 standard.

2.2.13 `-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.

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

```
% f90 -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.

2.2.14 `-l` *lib*

The `-l` *lib* option directs the loader to search for the specified object library file when loading an executable file.

If *lib* begins with a period (`.`) or a slash (`/`), it is assumed to be a full path name, and the loader uses it as is. Otherwise, the loader searches for a file named `lib`*lib*`.a` in each directory specified in the library search path. For more information on library search rules, see Section 2.2.15, page 18.

Example 1: On a UNICOS/mk system, the following command line loads in the Cray Research library that includes faster, but less accurate, vector versions of some intrinsic procedures and operations (such as exponentiation):

```
% f90 -O vector3 -l mfastv gazelle.f
```

Example 2: On a UNICOS/mk system, the following command line loads in the library that allows users to capture data for later performance analysis by `pat`(1):

```
% f90 -l pat camel.f
```

Example 3: On a UNICOS or UNICOS/mk system, the following command line loads in the library that allows users to capture data for later performance analysis by `flowview`(1) or `perfview`(1):

```
% f90 -l perf panther.f
```

2.2.15 `-L` *dir*

The `-L` *dir* option directs the loader to search for object library files in the specified directory before searching in the standard directories. The loader searches for library files specified by `-l` options in the directories specified by preceding `-L` options.

The *dir* argument can take the form of a comma-separated list of directories. The loader searches for library files in directory *dir* before checking in the standard directories. The standard system directories are `/opt/ctl/craylibs/craylibs`, `/lib`, and `/usr/lib`.

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

```
../mylibs/libm.a
/loclib/libm.a
/opt/ctl/craylibs/craylibs/libm.a
/lib/libm.a
/usr/lib/libm.a
```

See `segldr`(1) or `cld`(1) for more information on library searches. Note that the `f90`(1) command or `/opt/ctl/bin/segldr` adds `/opt/ctl/craylibs/craylibs` to your search path.

For information on specifying module locations, see Section 2.2.21, page 45.

2.2.16 `-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`(1) command. This command retrieves message explanations and displays them online. For example, to obtain documentation on message 500, enter the following command:

```
% explain cf90-500
```

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 2.2.17, page 19.

2.2.17 `-M` *msgs*

The `-M` *msgs* option suppresses specific messages at the comment, note, caution, and warning levels. For *msgs*, specify one or more integer numbers that correspond to the CF90 messages you want to suppress. If you want to specify more than one message number, enter a comma (but no spaces) between the message numbers. For example, to disable messages `cf90-100` and `cf90-200`, specify `-M 100,200`.

2.2.18 `-N` *col*

The `-N` *col* option specifies the line width for fixed-format source lines. For *col*, enter `72` to specify 72-column lines, enter `80` to specify 80-column lines, and enter `132` to specify 132-column lines. Characters in columns beyond the *col* specification are ignored.

By default, fixed-format source lines are 72 characters wide.

2.2.19 `-o` *out_file*

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

2.2.20 `-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.

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, user tasking, and Autotasking. Generally, as the `-O` level increases, compilation time increases and execution time decreases.

The `-O 1`, `-O 2`, and `-O 3` specifications do not directly correspond to the numeric optimization levels for scalar optimization, vectorization, and tasking. For example, specifying `-O 3` does not necessarily enable `scalar3` and `vector3`. Cray Research reserves the right to alter the specific optimizations performed at these levels from release to release. You can use the `ftnlist`(1) to obtain information on the specific optimizations used at compile time.

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

Table 2, page 21, summarizes the optimization features enabled when different *opt* levels are specified. The optimization specifications available differ depending on your operating system. If your command line includes an option that is not supported on your platform, the compiler issues a message and compilation continues. Default settings are shown in bold print in the leftmost column.

Table 2. `-O` *opt* summary

opt	Description	Operating system

`0`
`1`
`2`
`3`
	General optimization levels. Optimizations performed are none; conservative; moderate; and aggressive with moderate Autotasking.
	Note that Autotasking is enabled at `-O 3` only on UNICOS systems.
	Default is `2`.
	For more information, see Section 2.2.20.1, page 24, through Section 2.2.20.4, page 25.
		UNICOS
		UNICOS/mk

`aggress` **`noaggress`**	`aggress` raises the limits for internal tables, which increases opportunities for optimization. Default is `noaggress`. For more information, see Section 2.2.20.5, page 26.	UNICOS UNICOS/mk
`bl` **`nobl`**	`bl` enables full bottom loading of scalar operands in loops. Default is `nobl` (which performs only safe bottom loading). For more information, see Section 2.2.20.6, page 26.	UNICOS UNICOS/mk

`allfastint`
`fastint`
`nofastint`
	`allfastint` performs fast multiplication, division, and compare sequences for all integer data objects, regardless of how they are declared.
	`fastint` performs fast multiplication, division, and compare sequences for integer data objects that are default declared (with no `KIND=` or `*` specification).
	Default is `fastint`.
	For more information, see Section 2.2.20.7, page 26.
		UNICOS

`fusion` **`nofusion`**	`fusion` enables loop fusion, which is an optimization technique that merges loops. Default is `nofusion`. For more information, see Section 2.2.20.8, page 28.	UNICOS/mk
`ieeeconform` **`noieeeconform`**	`ieeeconform` causes the executable code to conform more closely to the IEEE floating-point standard than the default mode. Default is `noieeeconform`. This option is supported on UNICOS/mk systems and on CRAY T90 systems that support IEEE floating-point arithmetic. For more information, see Section 2.2.20.9, page 28.	UNICOS UNICOS/mk

opt	Description	Operating system
inline0 inline1 inline2 inline3	Specifies various levels of inlining. Default is inline0. For more information, see Section 2.2.20.10, page 29.	UNICOS UNICOS/mk
inlinefrom=*source*	Specifies a file or directory that contains procedures for inline code expansion. For more information, see Section 2.2.20.10, page 29.	UNICOS UNICOS/mk
jump nojump	jump generates jumps instead of branches to external functions. Default is jump. For more information, see Section 2.2.20.11, page 32.	UNICOS/mk
loopalign **noloopalign**	loopalign causes the compiler to attempt to align DO and DO WHILE loops on instruction buffer boundaries, comparing buffer length with the number of generated instructions in each loop. Default is noloopalign. For more information, see Section 2.2.20.12, page 33.	UNICOS
modinline **nomodinline**	modinline prepares module procedures so that they can be inlined. Default is nomodinline. For more information, see Section 2.2.20.13, page 33.	UNICOS UNICOS/mk
msgs **nomsgs**	msgs writes optimization messages to stderr. Default is nomsgs. For more information, see Section 2.2.20.14, page 33.	UNICOS UNICOS/mk
negmsgs **nonegmsgs**	negmsgs writes messages to stderr that indicate why a specific optimization did not occur. Default is nonegmsgs. For more information, see Section 2.2.20.15, page 34.	UNICOS UNICOS/mk
nointerchange	nointerchange inhibits the compiler's attempts to interchange loops. Default is to perform loop interchange optimizations. For more information, see Section 2.2.20.16, page 34.	UNICOS UNICOS/mk
overindex **nooverindex**	nooverindex asserts that there are no array subscripts that index a dimension of an array that are outside the declared bounds of that dimension. Default is nooverindex. For more information, see Section 2.2.20.17, page 34.	UNICOS UNICOS/mk

opt	Description	Operating system
pattern nopattern	pattern enables pattern matching for library substitution. Default is pattern. For more information, see Section 2.2.20.18, page 35.	UNICOS UNICOS/mk
pipeline0 pipeline1 pipeline2 pipeline3	Specifies various levels of software pipelining. Default is pipeline0. For more information, see Section 2.2.20.19, page 36.	UNICOS/mk
recurrence norecurrence	On UNICOS systems, recurrence enables vectorization of reduction loops. On UNICOS/mk systems, recurrence may rewrite some multiplication operations to be a series of addition operations. Default is recurrence. For more information, see Section 2.2.20.20, page 37.	UNICOS UNICOS/mk
scalar0 scalar1 **scalar2** scalar3	Specifies various levels of scalar optimization. Default is scalar2. For more information, see Section 2.2.20.21, page 37, through Section 2.2.20.24, page 38.	UNICOS UNICOS/mk
split0 **split1** split2	Specifies various levels of loop splitting. Default is split1. For more information, see Section 2.2.20.25, page 38.	UNICOS/mk
task0 **task1** task2 task3	Specifies various levels of tasking. Default is task1. For more information, see Section 2.2.20.26, page 39, through Section 2.2.20.29, page 40.	UNICOS
taskinner **notaskinner**	taskinner specifies Autotasking for innermost loops and requests that a threshold test be performed prior to Autotasking. The default is notaskinner. For more information, see Section 2.2.20.30, page 40.	UNICOS
threshold nothreshold	threshold performs threshold testing to determine whether there is sufficient work in a loop nest before Autotasking is attempted. The default is threshold. For more information, see Section 2.2.20.31, page 40.	UNICOS
unroll0 unroll1 unroll2	Specifies various levels of unrolling. The default is unroll0. For more information, see Section 2.2.20.32, page 41.	UNICOS/mk

opt	Description	Operating system

`vector0`
`vector1`
`vector2`
`vector3`
	Specifies various levels of vectorization.
	Default is `vector2`.
	For more information, see Section 2.2.20.33, page 41, through Section 2.2.20.36, page 42.
		UNICOS
		UNICOS/mk

`vsearch` `novsearch`	`vsearch` vectorizes search loops. The default is `vsearch`. For more information, see Section 2.2.20.37, page 43. The implementation of this feature on UNICOS/mk systems is deferred.	UNICOS UNICOS/mk
`zeroinc` **`nozeroinc`**	`zeroinc` specifies that constant increment variables (CIVs) can be incremented by zero. The default is `nozeroinc`. For more information, see Section 2.2.20.38, page 43.	UNICOS UNICOS/mk

For information on the Autotasking directives, see Appendix A, page 177.

The following sections describe the effects of various `-O` specifications. Some optimization specifications are not available on all platforms. If your command line includes an option that is not supported on your platform, the compiler issues a message and compilation continues.

2.2.20.1 `-O 0`

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

On UNICOS systems, all tasking is disabled. Most array syntax statements are vectorized, but all other vectorization is disabled.

On UNICOS/mk systems no vectorization occurs at this level.

2.2.20.2 `-O 1`

The `-O 1` option specifies conservative optimization. This option's characteristics include moderate compile time and size, global scalar optimizations, and no 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.

On UNICOS systems, only array syntax statements and inner loops are vectorized, and the system does not perform some vector reductions. All tasking is disabled.

On UNICOS/mk systems, implementation of vectorization at this level is deferred.

2.2.20.3 -O 2

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. The system does not perform optimizations that might create false exceptions.

On UNICOS systems, results may differ from results obtained when `-O 1` is specified because of vector reductions. Vectorizations, including outer loop vectorization, are enabled. User tasking is enabled, so `!$OMP` directives and `!MIC$` directives are recognized; note that the `!MIC$` directives are outmoded. Only safe *bottom loading* is performed.

On UNICOS/mk systems, implementation of vectorization at this level is deferred.

This is the default level of optimization.

2.2.20.4 -O 3

The `-O 3` option specifies aggressive optimization and moderate Autotasking. 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.

On UNICOS systems, results may differ from results obtained when `-O 1` is specified because of vector reductions. Moderate Autotasking is performed, and autoscoping rules are in effect. For more information on these rules, see Section A.12, page 195.

On UNICOS/mk systems, some intrinsic procedures and operations are vectorized. Autotasking is not supported.

2.2.20.5 `-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, but it increases compile time and size. In particular, specifying both `-O aggress` and `-O inline3` causes increased compile time and space. The default is `-O noaggress`.

On UNICOS systems, specifying `-O aggress` can result in better optimized code for large program units.

On UNICOS/mk systems, specifying `-O aggress` causes aggressive register assignment and instruction scheduling to be performed on additional loop code blocks that are not inner loops.

2.2.20.6 `-O bl`, `-O nobl`

The `-O bl` option enables full *bottom loading* of scalar operands in loops. The term *bottom loading* describes an optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

Bottom loading is enabled only when scalar optimization is enabled. `-O nobl` specifies safe bottom loading. The default is `-O nobl`. This feature is also available through compiler directives; for more information, see Section 3.4.2, page 96.

2.2.20.7 `-O allfastint`, `-O fastint`, `-O nofastint` (UNICOS Systems Only)

The `-O allfastint` option performs fast multiplication, division, and compare sequences for all integer data objects, regardless of how they are declared.

The `-O fastint` option performs fast multiplication, division, and compare sequences for integer data objects that are default declared (with no `KIND=` or * specification).

`-O nofastint` performs full 64-bit integer operations. The default is `-O fastint`.

Note: The `-O allfastint` and `-O fastint` options do not produce a result that is precise to a full 64 bits of precision.

Multiply and divide operations on default integers can be done as full 64-bit operations that yield the largest-magnitude results possible but use slower hardware instruction sequences. These operations can also be done with faster hardware instruction sequences that yield only 46-bit accuracy. Similarly, comparisons of integers can compare 64 bits or 46 bits, but comparisons of 64-bit lengths are slower.

The fast mode option, `-O fastint`, is the default. The DIGITS(3I), HUGE(3I), RANGE(3I) intrinsic functions return a smaller value for default integers in fast mode. When the MAXVAL(3I), MINVAL(3I), MAXLOC(3I), and MINLOC(3I) intrinsic functions are called with a default integer and a mask of false, they return a smaller value of plus or minus HUGE(3I).

When comparing the results of the MAXVAL(3I) and MINVAL(3I) intrinsic functions with the value returned by HUGE(3I), use the same type and kind type parameter for the argument for HUGE(3I) and these functions to get a correct comparison. To conform to the Fortran 90 standard, you must ensure that arguments to the MIN(3I) and MAX(3I) intrinsic functions have the same type and kind type parameters.

To obtain 64-bit accuracy, perform one of the following tasks:

- Declare integer variables using `INTEGER(KIND=8)`.
- Compile with `-O nofastint`. This command line option affects integer variables that do not have a kind type specifier.

The following program is compiled with two different sets of compiler options:

```
      PROGRAM PRINTFAST
      INTEGER :: INTMAX
      INTEGER (KIND=8) :: INT8MAX
      INT8MAX = HUGE(INT8MAX)
      INTMAX = HUGE(INTX)
!
! THE FOLLOWING STATEMENT COMPARES TWO INTEGERS OF THE
! SAME TYPE THAT MAY HAVE DIFFERENT KIND TYPE PARAMETERS.
!
      IF (INTMAX .EQ. INT8MAX) THEN
          PRINT *,'DEFAULT HUGE(INTX) .EQ. HUGE(INT8MAX)'
      ELSE
          PRINT *,'DEFAULT HUGE(INTX) .NE. HUGE(INT8MAX)'
      ENDIF
      END
```

The following shows the f90(1) command line and output with the -O fastint option:

```
% f90 -O fastint printfastint.f
% a.out
 DEFAULT HUGE(INTX) .NE. HUGE(INT8MAX)
```

The following shows the f90(1) command line and output with the -O nofastint option in effect:

```
% f90 -O nofastint printfastint.f
% a.out
DEFAULT HUGE(INTX) .EQ. HUGE(INT8MAX)
```

2.2.20.8 -O fusion, -O nofusion (UNICOS/mk Systems Only)

The -O fusion option enables aggressive loop fusion. *Loop fusion* is an optimization process by which two loops are merged into one loop. The loops to be fused must have identical trip counts.

Loop fusion can lower the number of memory references and improve cache behavior. Specifying the -O fusion option can lead to lower performance for some loops due to stream buffer thrashing.

The default is nofusion, which performs only conservative loop fusion.

Note: This option is implemented on UNICOS/mk systems. If -O fusion is specified on UNICOS systems, unsafe code may be generated.

2.2.20.9 -O ieeeconform, -O noieeeconform

The -O ieeeconform option causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode. When specified, many identity optimizations are disabled, executable code is slower, 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 noieeeconform option causes the compiler to optimize expressions such as X.NE.X to false and X/X to 1, where X is a floating-point value. With -O ieeeconform in effect, these and other similar arithmetic identity optimizations are not performed.

This option interacts with the -d u and -e u specifications. -O ieeeconform is compatible with -e u. If both -O ieeeconform and -d u are specified, however, -e u is set and a warning message is generated.

The default is `-O noieeeconform`. This option is supported on UNICOS/mk systems and on CRAY T90 systems that support IEEE floating-point arithmetic.

2.2.20.10 `-O inline`*n* and `-O inlinefrom=`*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 vectorization or tasking. The CF90 compiler supports the following command line options for controlling inlining:

- `-O inline0`, `-O inline1`, `-O inline2`, `-O inline3`
- `-O inlinefrom=`*source*

The following conditions inhibit inlining:

- Dummy argument types and kind type parameter values in the called procedure that differ from corresponding actual argument types and kind type parameter values.
- The number of dummy arguments being not equal to the number of actual arguments.
- A call site that is within the range of a `NOINLINE` directive.
- A procedure being called is specified on an `INLINENEVER` directive.
- A constant actual argument that has a corresponding dummy argument that is defined by assignment in the procedure.
- The routine being called is declared `RECURSIVE`.
- A dummy argument of a host procedure is referenced in an internal procedure of the host procedure. If this condition exists, the host is not inlined.
- The compiler determines that the routine is too big to inline. This is determined by an internal limit of the text size of the routine. You can override this limit by inserting an `INLINEALWAYS` directive. For information on the `INLINEALWAYS` directive, see Section 3.3.2, page 93.
- The procedure being called cannot contain any of the following:
  - A `LOC`(3I) of a variable declared in a common block
  - Calls to the `NUMARG`(3I) intrinsic procedure

- Calls to the PRESENT(3I) intrinsic procedure
- ASSIGN statements
- Alternate RETURN statements
- Dummy procedures
- Dummy arguments declared with the OPTIONAL attribute
- Fortran 90 pointers in static storage (COMMON, MODULE, DATA, or SAVE)
- Dummy arguments that are Cray pointers

The two inlining modes are *automatic* and *explicit*. You cannot invoke both automatic and explicit inlining modes at the same time. These modes can be characterized as follows:

- *Automatic inlining* is invoked with the `-O inline`*n* option on the command line. Routines that are potential targets for inline expansion include all the routines within the input file to the compilation. In automatic mode, you can choose the level of heuristics to be applied to the input program. The higher the level, the more aggressive the inlining.

  The `-O inline`*n* options let you specify the amount of automatic inline code expansion desired. The `-O inline0` option disables all inline code expansion, and the `-O inline3` option specifies aggressive inline code expansion. Table 3, page 31, explains the levels in more detail.

Table 3. Automatic inlining specifications

Level	Description
`0`	No inlining. All inlining disabled. All inlining compiler directives are ignored. Default.
`1`	Conservative inlining. Inlining attempted for call sites and routines that are under the control of a compiler directive. See Chapter 3, page 71, for more information on the inlining directives.
`2`	Moderate inlining. Inlining attempted on calls described in `-O inline1`. In addition, inlining is attempted for call sites that exist within `DO` loops.
`3`	Aggressive inlining. Inlining attempted for all call sites in the input program.

- *Explicit inlining* is invoked with the `-O inlinefrom=`*source* option. All inlining directives are recognized with explicit inlining. For information on inlining directives, see Chapter 3, page 71.

  The `-O inlinefrom=`*source* option lets you explicitly state the routines that are targets for inline expansion. (Note that blanks are not allowed on either side of the equal sign.) With this mode of inlining, you must put the routines to be inlined in *source*.

  Note: Module procedures contained in *source* must be precompiled with the `-O modinline` option. You cannot just use the Fortran source of a module procedure as input to the `-O inlinefrom=` option.

  Whenever a call is encountered in the input program to a routine that exists in *source*, inlining is attempted for that call site.

  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 as an input file to the compilation. Table 4, page 32, describes *source*.

Table 4. Description of *source*

source	Description
file.o	Precompiled module. *file*.o contains templates of module procedures that are precompiled for inlining. You can save this *file*.o and use it for inlining at a later time. The -O modinline command line option creates these templates.See Section 2.2.20.13, page 33, for information on using -O modinline.
file.a	Precompiled archive library of modules. *file*.a is an archive library containing one or more *file*.o files.
file.f *file*.F *file*.f90 *file*.F90	Fortran source file. These files contain error-free Fortran source code. The routines in these files are candidates for inline expansion. **Note:** Module procedures contained in a Fortran source file must be precompiled using -O modinline for them to be inlined.
dir	A directory. A directory that contains any of the file types described in this table.

Note: You cannot invoke both the automatic and the explicit inlining modes at the same time.

2.2.20.11 -O jump, -O nojump (UNICOS/mk Systems Only)

The -O jump option causes jumps, instead of branches, to external functions. By default, jumps are generated instead of branches because this is safer. Branches are limited in the distance over which they can transfer control; jumps have no such limitations.

Large programs may benefit when -O jump is in effect. This benefit can be seen when compiling files that generate calls to functions that are loaded at a large offset from the position of the call site that is invoking the procedure. A loader message similar to the following alerts you to the necessity of compiling with -O jump:

```
cld-130
The dex expression dex-index in relocatable
object 'relo-obj-name' for symbol 'symbol-name'
calculated a relative branch target
too distant.
```

The default is jump.

2.2.20.12 `-O loopalign`, `-O noloopalign` (UNICOS Systems Only)

The `-O loopalign` option causes the compiler to attempt to align `DO` and `DO WHILE` loops on instruction buffer boundaries, comparing buffer length with the number of generated instructions in each loop. Loop alignment is useful when program execution is dominated by specific blocks of code. If such a block crosses a buffer boundary, the overhead caused by frequent reloading of instruction buffers degrades program performance.

When `-O loopalign` is specified, the compiler counts the number of generated instructions in each `DO` or `DO WHILE` loop and compares this to the length of the instruction buffers. If the loop body fits in the buffers and would otherwise cross over the buffer boundary, the loop is aligned on a buffer boundary. Short loops, unwound loops, and loops containing external references are not aligned.

This option does not interact with the `ALIGN` compiler directive. The default is `-O noloopalign` (no attempt to align loops).

2.2.20.13 `-O modinline`, `-O nomodinline`

The `-O modinline` option directs the compiler to create templates for module procedures encountered in a module. These templates are attached to *file*`.o`. The files that contain these inlinable templates can be saved and used later to inline call sites within a program being compiled with the `-O inlinefrom=`*source* command line option. When `-O modinline` is specified, the `MODINLINE` and `NOMODINLINE` directives are recognized. Using the `-O modinline` option increases the size of *file*`.o`. The default is `-O nomodinline`.

To ensure that *file*`.o` is not removed, specify this option in conjunction with the `-c` option. For information on the `-c` option, see Section 2.2.4, page 8.

Note: This option cannot be specified in conjunction with the `-O inlinefrom=`*source* or `-O inline`*n* options.

2.2.20.14 `-O msgs`, `-O nomsgs`

The `-O msgs` option causes the compiler to write optimization messages to `stderr`. These messages include `VECTOR`, `SCALAR`, and `TASK` messages.

The default is `-O nomsgs`. When `-O nomsgs` is in effect, you may request that a listing be produced so that you can see the optimization messages in the listing. For information on obtaining listings, see Section 2.2.22, page 47.

2.2.20.15 -O negmsgs, -O nonegmsgs

The -O negmsgs option causes the compiler to generate messages that indicate why optimizations such as vectorization or tasking did not occur in a given instance. This option must be specified in conjunction with the -O msgs option.

The default is -O nonegmsgs.

2.2.20.16 -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 on the NOINTERCHANGE directive, see Section 3.4.4, page 100.

2.2.20.17 -O overindex, -O nooverindex

The -O nooverindex option declares that there are no array subscripts that index a dimension of an array and that are outside the declared bounds of that dimension. On UNICOS systems, shortloop 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 shortloop 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 can 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`.

2.2.20.18 `-O pattern`, `-O nopattern`

The `-O pattern` option enables pattern matching. The pattern matching feature searches your code for specific code patterns and replaces them with calls to scientific library routines. The scientific library used is `libsci.a`. These routines are highly optimized and may contain multitasked code.

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.

On UNICOS/mk systems, only PE-private data is supported.

Specifying `-O nopattern` disables pattern matching and causes the compiler to ignore the PATTERN and NOPATTERN directives. For information on the PATTERN and NOPATTERN directives, see Section 3.2.5, page 81.

The default is `-O pattern`.

2.2.20.19 `-O pipeline`*n* (UNICOS/mk Systems Only)

The pipelining options specify various levels of software pipelining ranging from no pipelining, at `-O pipeline0`, to a pipelining level that also includes speculative loads and operations, at `-O pipeline3`.

Software pipelining is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase the instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to *bottom loading*, but it includes additional, and more efficient, scheduling optimizations.

The various software pipelining levels you can specify perform the following types of operations:

- `-O pipeline0` disables pipelining. Default.
- `-O pipeline1` specifies conservative pipelining. Compile times at this level are lower than if `-O pipeline2` is specified. At this level, only loops of the following type are software pipelined:
  - Parallel loops (those without data dependencies between iterations).
  - Vectorizable loops with an infinite safe vector length.
  - Loops marked with a `CONCURRENT` directive. For more information on the `CONCURRENT` directive, see Section 3.7.1, page 113.
- `-O pipeline2` specifies safe pipelining on inner loops. Safe operator reassociations are performed. Numeric results obtained at this level do not differ from results obtained at `pipeline0`.
- `-O pipeline3` specifies aggressive pipelining. The system performs software pipelining, speculative loads, and speculative operations. These optimizations could lead to floating-point exceptions and operand range errors.

  Implementation of features at this level is deferred. Functionality is the same at that obtained when `-O pipeline2` is specified.

At the `-O pipeline1`, `-O pipeline2`, and `-O pipeline3` levels, compile times may be longer, but execution times are shorter. The aggressive scheduling that can be obtained with this option increases the instruction issue rate and hides latency better than the default code generation method.

You can use the CONCURRENT directive to convey array dependency information to the compiler. For information on the CONCURRENT directive, see Section 3.7.1, page 113.

2.2.20.20 -O recurrence, -O norecurrence

On UNICOS systems, -O recurrence enables vectorization for all reduction loops. A *reduction loop* is a loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

On UNICOS/mk systems, -O recurrence may rewrite some multiplication operations to be a series of addition operations.

The default is -O recurrence. This feature is also available through compiler directives; for more information, see Section 3.2.9, page 84.

2.2.20.21 -O scalar0

The -O scalar0 option disables scalar optimization. Characteristics include low compile time and size.

On UNICOS systems, -O scalar0 is compatible with -O task0 or -O task1 and with -O vector0.

On UNICOS/mk systems, -O scalar0 is compatible with -O vector0.

2.2.20.22 -O scalar1

The -O scalar1 option 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; for example, on UNICOS systems, only safe *bottom loading* is performed.

On UNICOS systems, -O scalar1 is compatible with -O vector0 or -O vector1 and with -O task0 or -O task1.

On UNICOS/mk systems, -O scalar1 is compatible with -O vector0 or -O vector1.

2.2.20.23 `-O scalar2`

The `-O scalar2` option 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 loop nest restructuring. Generally, no optimizations are done that could create false exceptions. For example, only safe *bottom loading* is performed.

On UNICOS systems, `-O scalar2` is compatible with all vectorization and tasking levels.

On UNICOS/mk systems, `-O scalar2` is compatible with all vectorization levels.

This is the default scalar optimization level.

2.2.20.24 `-O scalar3`

The `-O scalar3` option 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 loop nest restructuring. On UNICOS/mk systems and on CRAY T90 systems that support IEEE arithmetic, strength reduction of floating-point values is performed.

The optimization techniques used can create false exceptions in rare instances; for example, full *bottom loading* is performed. Analysis that determines whether a variable is used before it is defined is enabled at this level. The `-O scalar3` optimization level is never enabled automatically, even when `-O 3` is specified. This scalar optimization level must be requested specifically on the command line.

On UNICOS systems, `-O scalar3` is compatible with all tasking and vectorization levels.

On UNICOS/mk systems, `-O scalar3` is compatible with all vectorization levels.

2.2.20.25 `-O split`*n* (UNICOS/mk Systems Only)

The `-O split0`, `-O split1`, and `-O split2` options specify loop splitting, as follows:

- `-O split0` disables loop splitting and directs the compiler to ignore `SPLIT` compiler directives.

- `-O split1` causes the compiler to split only the loops that are preceded by a `SPLIT` compiler directive. Default.
- `-O split2` causes the compiler to evaluate all loops in the compilation as candidates for splitting except those that are preceded by a `NOSPLIT` directive.

Loop splitting is a code optimization technique by which a loop that contains both vectorizable work and scalar work is split into two loops; one that vectorizes, and one that does not. On CRAY T3E systems, even if splitting does not result in a vectorizable loop, the program can benefit by reducing the number of memory accesses that are not in cache.

The default is `-O split1`. This feature is also available through compiler directives. For more information, see Section 3.4.6, page 102.

2.2.20.26 `-O task0` (UNICOS Systems Only)

The `-O task0` option disables tasking. Characteristics include low compile time and size. `!$OMP` and `!MIC$` directives are ignored.

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

2.2.20.27 `-O task1` (UNICOS Systems Only)

The `-O task1` option specifies user tasking, so `!$OMP` directives and `!MIC$` directives are recognized; note that the `!MIC$` directives are outmoded.

Characteristics include low compile time and size. No level for scalar optimization is enabled automatically. This is the default optimization level for tasking.

Note: In releases prior to 3.1, lines beginning with `!$`, `C$`, or `*$` were always treated as comments. With the introduction of the OpenMP Fortran API, these lines are now treated as conditional compilation lines and are compiled as source code when tasking is in effect. To have these lines treated as comments, remove the dollar sign ($) from these lines or compile with the `-x conditional_omp` command line option.

For more information on the `-x conditional_omp` command line option, see Section 2.2.35, page 56. For more information OpenMP, see Chapter 4, page 119.

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

2.2.20.28 `-O task2` (UNICOS Systems Only)

The `-O task2` option specifies moderate Autotasking. Characteristics include moderate compile time and size and possible loop nest restructuring. No optimizations that can create exceptions or differing results are performed. Autoscoping rules are in effect. For more information on these rules, see Section A.12, page 195.

The `-O task2` option is compatible with `-O scalar2` or `-O scalar3` and with `-O vector2` or `-O vector3`.

Lines that begin with `!$`, `C$`, or `*$` may have to be rewritten when tasking is enabled. For more information on this, see Section 2.2.20.27, page 39.

2.2.20.29 `-O task3` (UNICOS Systems Only)

The `-O task3` option specifies aggressive Autotasking. Characteristics include a potentially high compile time and size. Possible loop nest restructuring. Results can differ slightly from those obtained when `-O task2` is specified, and from run to run, because of parallel reductions and operator reassociations. Autoscoping rules are in effect. For more information on these rules, see Section A.12, page 195.

The `-O task3` option is compatible with `-O scalar2` or `-O scalar3` and with `-O vector3`.

Lines that begin with `!$`, `C$`, or `*$` may have to be rewritten when tasking is enabled. For more information on this, see Section 2.2.20.27, page 39.

2.2.20.30 `-O taskinner`, `-O notaskinner` (UNICOS Systems Only)

The `-O taskinner` option specifies Autotasking for innermost loops. Autotasking must be enabled for this directive to take effect. Autotasking is enabled when `-O 3`, `-O task2`, or `-O task3` is specified.

The default is `-O notaskinner`.

2.2.20.31 `-O threshold`, `-O nothreshold` (UNICOS Systems Only)

The `-O threshold` option generates a runtime threshold test to determine whether there is sufficient work in a loop nest before Autotasking is attempted.

Autotasking must be enabled for this directive to take effect. Autotasking is enabled when `-O 3`, `-O task2`, or `-O task3` is specified.

The default is `-O threshold`.

2.2.20.32 `-O unroll`*n* (UNICOS/mk Systems Only)

The unrolling arguments specify loop unrolling, as follows:

- `-O unroll0` disables loop unrolling and directs the compiler to ignore `UNROLL` compiler directives. Default.
- `-O unroll1` causes the compiler to unroll only the loops that are preceded by an `UNROLL` compiler directive.
- `-O unroll2` causes the compiler to evaluate all loops in the compilation as candidates for unrolling except those that are preceded by a `NOUNROLL` compiler directive.

On UNICOS systems, `-O unroll2` is in effect at all times.

The default is `-O unroll0`. This feature interacts with the `UNROLL` compiler directive. For more information, see Section 3.2.12, page 87.

2.2.20.33 `-O vector0`

The `-O vector0` option specifies low vectorization. Characteristics include low compile time and small compile size.

On UNICOS systems, `-O vector0` 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.

On UNICOS/mk systems, `-O vector0` is compatible with all scalar optimization levels. Vector versions of intrinsic functions and library-based operators are not used at this level.

2.2.20.34 `-O vector1`

The `-O vector1` option specifies conservative vectorization. Characteristics include moderate compile time and size. No loop nests are restructured. Only inner loops are vectorized. Not all vector reductions are performed, so results do not differ from results obtained when `-O vector0` is specified. No vectorizations that might create false exceptions are performed.

On UNICOS systems, `-O vector1` is compatible with `-O task0` or `-O task1` and with `-O scalar1`, `-O scalar2`, or `-O scalar3`.

On UNICOS/mk systems, `-O vector1` is compatible with `-O scalar1`, `-O scalar2`, or `-O scalar3`. The use of vector versions of intrinsic functions and library-based operators, such as `**`, at this level is deferred.

2.2.20.35 `-O vector2`

The `-O vector2` option specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-O vector1` is specified because of vector reductions. No vectorizations that might create false exceptions will be performed. Pattern matching is enabled.

On UNICOS systems, `-O vector2` is compatible with `-O scalar2` or `-O scalar3` and with `-O task0`, `-O task1`, or `-O task2`.

On UNICOS/mk systems, `-O vector2` is compatible with `-O scalar2` or `-O scalar3`. The use of vector versions of intrinsic functions and library-based operators, such as `**`, at this level is deferred.

This is the default vectorization level.

2.2.20.36 `-O vector3`

The `-O vector3` option specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-O vector1` is specified because of vector reductions. Vectorizations that might create false exceptions in rare cases may be performed. Pattern matching is enabled.

On UNICOS systems, `-O vector3` is compatible with `-O scalar2` and `-O scalar3` and with all tasking levels.

On UNICOS/mk systems, `-O vector3` is compatible with `-O scalar2` and `-O scalar3`. Vector versions of the following intrinsic functions and library-based operators are used when found in a vectorizable loop: ACOS(3M), ALOG(3M), ALOG10(3M), ASIN(3M), ATAN(3M), ATAN2(3M), the BMM(3I) routines, COS(3M), COSS(3M), EXP(3M), LOG(3M), LOG10(3M), POPCNT(3I), RANF(3I), RTOR(3M), SIN(3M), SQRT(3M), SQRTINV(3M), and `**`. These vector routines operate on an array of elements and return an array of results. This vectorization is performed using the following process:

1. The loop is stripmined. See the glossary for information on *stripmining*.

2. If necessary, a strip of operands is stored into a temporary array. The vector version of the intrinsic function is called, which stores the strip of results into a temporary array.

3. The remainder of the loop is computed using the results from step 2.

See the man pages for these intrinsics for more specific information on how data sizes affect vectorization.

On UNICOS/mk systems, the vector routines from `libm`, the default math library, return results identical to those obtained when scalar routines are used. For better performance, but slightly less accuracy, specify `-l mfastv` on the command line. This option loads a nondefault math library. For more information on this option, see Section 2.2.14, page 17.

For more information on vectorization on UNICOS/mk systems, see the *CRAY T3E Fortran Optimization Guide*.

2.2.20.37 `-O vsearch`, `-O novsearch`

The `-O vsearch` option vectorizes search loops. `-O novsearch` disables vectorization of search loops. A *search loop* is one that can be exited by means of a `GO TO` statement or `EXIT` statement.

The `-O vsearch` option is the default when `-O vector2` or `-O vector3` are enabled. `-O novsearch` is the default when `-O vector0` or `-O vector1` are enabled. The implementation of this feature on UNICOS/mk systems is deferred.

This feature is also available through compiler directives; for more information, see Section 3.2.15, page 92.

2.2.20.38 `-O zeroinc`, `-O nozeroinc`

The `-O zeroinc` option causes the compiler to assume that constant increment variables (CIVs) 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.

2.2.20.39 Optimization Values

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

	scalar0	scalar1	scalar2	scalar3	vector0	vector1	vector2	vector3	task0	task1	task2	task3
Low compile cost	✘				✘				✘	✘		
Moderate compile cost		✘	✘			✘	✘				✘	
Potentially high compile cost				✘				✘				✘
No numerical differences from serial execution	✘	✘										
No numerical differences from serial execution (no vector reductions)					✘	✘						
Potential numerical differences from serial execution (vector reductions)							✘	✘				
Potential numerical differences from serial execution and from run to run due to parallel reductions												✘
Potential numerical differences from serial execution (operator reassociation)		✘	✘	✘								
No optimizations that may create exceptions (safe bottom loading)	✘	✘	✘									
No optimizations that may create exceptions	✘	✘	✘		✘	✘	✘		✘	✘	✘	
Optimizations that may create exceptions (full bottom loading)				✘								
Implies at least scalar1						✘				✘		
Implies at least scalar2							✘	✘			✘	✘
No loop nest restructuring					✘	✘			✘	✘		
Loop nest restructuring							✘	✘			✘	✘
Vectorize array syntax statements					✘[1]	✘[1]	✘[1]	✘[1]				
Vectorize only inner loops						✘						
No vectorization may create exceptions					✘	✘	✘					
Vectorization that may create exceptions								✘				
All tasking disabled									✘			
User tasking enabled										✘	✘	✘

[1] Not supported on UNICOS/mk systems

a10133

Figure 2. Optimization values

2.2.21 `-p` *module_site*

The `-p` *module_site* option allows you to specify a file or directory that contains modules.

The *module_site* 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.

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`. After searching the directory named in *module_site*, the compiler searches for modules in the current directory.

The module files are not interchangeable. For example, it is not possible to create modules for UNICOS systems and read the files on UNICOS/mk systems.

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.

A module called `FTN_IEEE_DEFINITIONS` is provided as part of the CF90 Programming Environment on UNICOS/mk systems and on CRAY T90 systems that support IEEE floating-point arithmetic. On these systems, the system file that contains this module is searched last. To reference this module, specify `USE FTN_IEEE_DEFINITIONS`.

Example 1: Consider the following command line:

```
% f90  -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`:

```
% f90  -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`:

```
% f90  -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 `f90`(1) command and a relative path name:

```
% f90  -p progone.o  progone.f  progtwo.f
```

2.2.22 `-r` *list_opt*

The `-r` *list_opt* option produces a listing file. If the `-C` option is specified along with the `-r` *list_opt* option, the `-C` option is overridden and a warning message is generated. If both the `-r` *list_opt* option and the `-Wr"`*ftnlist_opt*`"` options are specified, the listing options specified by all the `-r` *list_opt* options are placed on the `ftnlist`(1) command line before those specified by the `-Wr"`*ftnlist_opt*`"` option.

By default, if only one input file is specified on the `f90`(1) command line, the listing is placed in *file*`.lst`. If more than one input file is specified on the command line, the listing is placed in file `ftnlist.out`.

The `-r` *list_opt* option allows you to generate a listing directly from the `f90`(1) command. The *list_opt* argument produces listings with commonly needed information. If you need a customized report, you can use `ftnlist`(1) directly.

The `-r` *list_opt* option accepts only one argument unless you are specifying the `T` argument; `T` can be specified in conjunction with one other argument.

The *list_opt* values are as follows:

list_opt	Listing type
`0`	Listing includes the standard `ftnlist`(1) listing information. Includes global reports when you have more than one compilation unit.
`1`	Listing includes the source listing, loop marks, and parallel marks. Regardless of the number of compilation units, no global reports are included. This is a minimal listing.
`2`	Listing includes the source listing, loop marks, parallel marks, considerata, and compiler messages. Regardless of the number of compilation units, no global reports are included. Produces a short, standard listing.
`3`	Listing includes the source listing, loop marks, parallel marks, an argument report, a parallelism report, identifier cross-references, considerata, and compiler messages. Regardless of the number of compilation units, no global reports are included.
`4`	Listing includes all subprogram items that can be listed with the default `ftnlist`(1) options. Regardless of the number of compilation units, no global reports are included.
`5`	Listing includes all subprogram items that can be listed with the default `ftnlist`(1) options. If more than one compilation unit is present, the Global Considerata report is included.
`6`	Listing includes all items that can appear in a listing. All `ftnlist`(1) options are enabled. This option produces the maximum amount of listing information.
`c`	Listing includes all common blocks and all members of each common block.

`g`	Saves the generated binary code and its assembly language equivalent to *file*`.L`. Unlike most other `-r` *list_opt* arguments, specifying `-r g` does not create a CIF.
`l`	Invokes `ftnlint`(1) with default `ftnlint`(1) options. You cannot pass option information directly to `ftnlint`(1). If you require a particular `ftnlint`(1) analysis, you must call it separately. If more than one source file is listed on the command line, the output is written to file `ftnlint.out`.
`m`	Produces a source listing with loopmark information. Loopmark information will not be displayed if the `-d B` option has been specified.
`n`	Suppresses page breaks in the listing.
`s`	Lists source code. Messages are interspersed with the source lines. Produces 80-column output by default.
`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. 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 an error message.
`x`	Generates a cross-reference listing. Produces 80-column output by default.
`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 on *file*`.T`, see the `-C` option.

2.2.23 `-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, as follows: `-R ab`.

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.

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 inline2`, or `-O inline3`.

`b` Enables checking of array bounds. If a problem is detected at run time, a message is issued but execution continues. Arrays in formatted `WRITE` and `READ` statements are not checked. The `NOBOUNDS` directive overrides this option. For more information on `NOBOUNDS`, see Section 3.5.1, page 105.

Note: Bounds checking behavior differs with the optimization level. At the default optimization level, `-O 2`, some runtime checking is inhibited. Complete checking is guaranteed only when optimization is turned off by specifying `-O 0` on the `f90`(1) 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.

`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`).

`s` Enables checking of character substring bounds. Arrays in formatted `WRITE` and `READ` statements are not checked. This option behaves similarly to option `-R a` or `-R b`.

Note: Bounds checking behavior differs with the optimization level. At the default optimization level, `-O 2`, some runtime checking is inhibited. Complete checking is guaranteed only when optimization is turned off by specifying `-O 0` on the `f90`(1) command line.

`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.

`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.

Note: CF90 2.0 binaries compiled with the `-R a`, `-R C`, or `-R E` options must be recompiled with CF90 3.0 in order to execute correctly with PE 3.0 libraries. Without the recompilation, the program aborts or generates unexpected results due to a change in the calling sequence for argument checking.

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.

2.2.24 `-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. For *size*, enter one of the following:

size	Action
`default32` (UNICOS/mk systems only)	Adjusts the default sizes as follows: real, integer, and logical are set to 32 bits; complex and double precision are set to 64 bits; double complex is set to 128 bits.
`cf77types` `i`	Relaxes the strict data typing rules of Fortran 90 when more than one kind exists for a type. Specifying `-s cf77types` or `-s i` causes data types to map to only the standard intrinsic FORTRAN 77 types. The `-s i` option is effective only for data types declared using the asterisk (`*`) format (for example, data items declared as `INTEGER*2`). Data types declared with the `KIND=` syntax are not affected.

Caution: The ability to specify `-s i` is outmoded. The new option used to specify this functionality is `-s cf77types`. The `-s i` option will be removed in the CF90 3.3 release.

If you specify `-s` *size* in conjunction with data declarations that include size in your source code, the default size is not used. For example, if you specify `REAL(KIND=`*n*`) R` or `COMPLEX(KIND=`*n*`) Z`, the values specified in *n* are used.

When `-s default32` or `-i 32` is specified, the CF90 compiler does not comply with Fortran 90 standard requirement for aligning data in a storage sequence. Numeric sequence derived types might not align properly with the numeric entities that are equivalenced to them.

Assume that the following example is compiled with `-s default32`:

```
TYPE T
  SEQUENCE
  REAL :: A(3)
END TYPE
TYPE T1
  SEQUENCE
  TYPE(T) :: F
  TYPE(T) :: S
END TYPE
TYPE(T1) :: X
REAL :: C(6)
EQUIVALENCE (C,X)
```

In this example, array C would be stored as shown in the following figure:

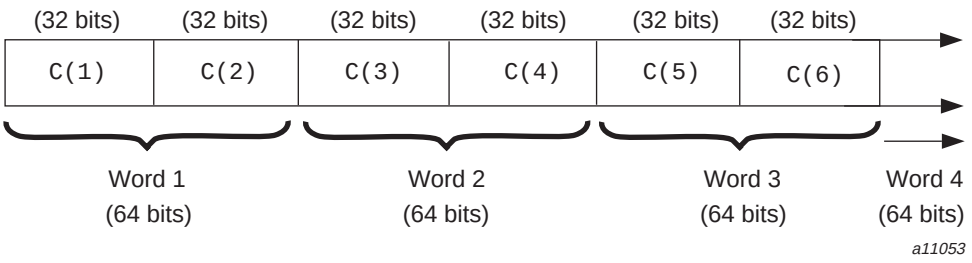

Figure 3. Array storage

However, the derived type is stored as shown in the following figure:

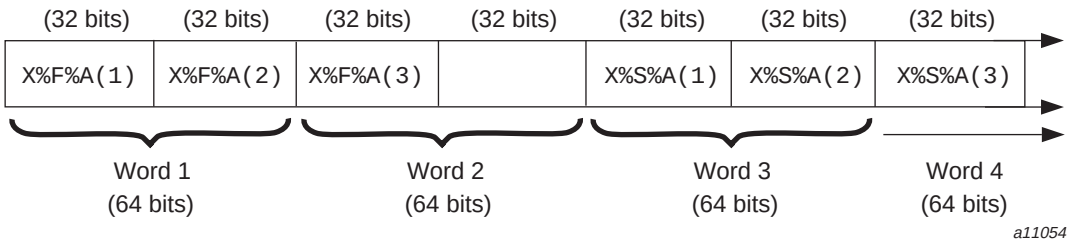

Figure 4. Derived type storage

As the preceding figure shows, each component of a derived type that is, itself, a derived type, is aligned on a word boundary. Consequently, the EQUIVALENCE statement in this example probably does not produce the intended storage association.

2.2.25 -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.

2.2.26 -t *num* (UNICOS Systems Only)

The -t *num* option specifies the number of bits to be truncated on floating-point operations. For *num*, enter an integer in the range $0 \leq num \leq 47$. The default is 0.

This option is not valid on UNICOS/mk systems. If specified, it is ignored, and a value of 0 is used.

2.2.27 -T

The -T option disables the compiler but displays all options currently in effect. The CF90 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).

2.2.28 -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*][,*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 on the -D option, see Section 2.2.7, page 14.

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

- The Fortran input source file is specified as either *file*.F or *file*.F90.

- The `-e P` or `-e Z` options have been specified.

For more information on source preprocessing, see Chapter 5, page 167.

2.2.29 -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.

2.2.30 -V

The `-V` option directs each compilation phase to send a message containing version information to the standard error file (`stderr`). Unlike all other command-line options, you can specify this option without specifying an input file name; that is, specifying `f90 -V` is valid.

2.2.31 -Wa"*assembler_opt*"

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

2.2.32 -Wl"*loader_opt*"

The `-Wl"`*loader_opt*`"` option passes *loader_opt* directly to the loader. For example, specifying `-Wl"-D MAP=FULL"` passes the argument `MAP=FULL` directly to the loader's `-D` option. For more information on loader options, see `segldr`(1) or `cld`(1).

2.2.33 -Wp"*srcpp_opt*"

The `-Wp"`*srcpp_opt*`"` option passes *srcpp_opt* to the source preprocessor. For *srcpp_opt*, enter one of the following:

srcpp_opt	Action
`-M`	Prints the `#include` file hierarchy to the standard output file, `stdout`. This is often used to help create `make`(1) files. Must be specified in conjunction with `-e P`.
`-P`	Suppresses line number information in the output file. Do not specify `-Wp"-P"` with any options that create a listing.

The `-Wp"`*srcpp_opt*`"` option is ignored unless one of the following is true:

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

For more information on source preprocessing, see Chapter 5, page 167.

2.2.34 `-Wr"`*ftnlist_opt*`"`

The `-Wr"`*ftnlist_opt*`"` option passes *ftnlist_opt* directly to `ftnlist`(1). For example, specifying `-Wr"-o cfile.o"` passes the argument `cfile.o` directly to the `ftnlist`(1) command's `-o` option; this directs `ftnlist`(1) to override the default output listing and put the output file in `cfile.o`. If you specify the `-Wr"`*ftnlist_opt*`"` option, you must specify the `-r` *list_opt* option. For more information on options, see `ftnlist`(1).

2.2.35 `-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:

dirlist	Item disabled
`all`	All compiler directives, OpenMP Fortran API directives, and Autotasking directives. For descriptions of these directives, see Chapter 3, page 71.
`dir`	All compiler directives.
directive	One or more compiler directives, OpenMP Fortran API directives, or Autotasking directives.

	If specifying more than one, separate them with commas, as follows: `-x INLINEALWAYS,"NO SIDE EFFECTS",BOUNDS`.
`mic`	All Autotasking directives.
`omp`	All OpenMP Fortran API directives.
`conditional_omp`	All `C$` and `!$` conditional compilation lines.

2.2.36 `-X` *npes* (UNICOS/mk Systems Only)

The `-X` *npes* option specifies the number of processing elements (PEs) used during execution. Enter one of the following for *npes*:

npes	PEs
n	An integer in the range 1 through 2048.
`m`	Directs the compiler to generate a malleable `a.out` file. Specifying `-X m` allows you to change the number of PEs used each time the executable `a.out` file is run. If you specify `-X m`, use the `mpprun`(1) command and its `-n` option to specify the number of PEs you want to use. For more information, see `mpprun`(1).

You cannot use the `-X m` option if you have used the `N$PES` constant in declarations in your program. The compiler issues an error message when it detects `N$PES` used in a declaration when `-X m` is specified.

A *malleable* `a.out` file is one for which the number of processors used can be specified at run time. If you do not use `mpprun`(1) on the `a.out` file that is generated when `-X m` is specified, the operating system executes the file on a single processor just as if you had invoked `mpprun`(1) with one processor.

The `-X` *npes* option is passed from the command line to both the compiler and the loader as follows:

- If your command line excludes the loader, the `-X` *npes* specification is a compile-time value and cannot be changed at load time. In the following example, the `-c` option specifies that the loader is not to be invoked:

  ```
  % f90 -X8 -c myfile.f
  ```

- If your command line calls the compiler and the loader, the compiler is called first. In the following example, the `-X` *npes* specification is a compile-time value and cannot be changed at load or run time:

```
% f90 -X8 hisfile.f
```

- If your command line calls the loader and not the compiler, the `-X` *npes* option is a load-time value and cannot be changed at `mpprun`(1) time. Example:

```
% f90 -X8 herfile.o
```

The `N$PES` constant is a special constant that can be used when programming UNICOS/mk systems. The value of `N$PES` is equal to the number of PEs, and thus the number of tasks, available to your program. The number of the first PE is always 0, and the number of the last PE is `N$PES-1`.

`N$PES` can be used in some of the same places as any named constant. For example, `N$PES` can be used in declaration statements and as an extent for array dimensions. The following restrictions apply:

- `N$PES` cannot be used in a `CHARACTER`, `DATA`, or `FORMAT` statement.
- `N$PES` cannot be used in a complex constant.
- Arrays with bounds that depend on the value of `N$PES` at load time or run time cannot be specified in an `EQUIVALENCE` statement or in a `PARAMETER` statement.
- `N$PES` can be used only as an operand in an addition, subtraction, multiplication, or division process.

One of the many uses for `N$PES` 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-1)
```

Using `N$PES` does not make a program faster, but it saves time in other ways by enhancing program portability and maintainability.

2.2.37 `-Z` (UNICOS/mk Systems Only)

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.

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.

Previously, the sole method for performing data passing on UNICOS/mk platforms was through the shared memory (SHMEM) library routines. The chief advantage of using co-arrays, as compared to SHMEM, is enhanced readability and, thus, increased programmer productivity. As a language extension, the code can also be conditionally analyzed and optimized by the compiler.

For more information on co-arrays, see the *CF90 Co-array Programming Manual*.

2.2.38 `--`

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 files begin with one or more dash (`-`) characters.

2.2.39 *file.suffix*[`90`] [*file.suffix*[`90`]] `...`

This option names the file or files to be processed, where *suffix* is either an uppercase `F` or a lowercase `f`. The file suffixes indicate the content of each file and determine whether the compiler, assembler, or loader will be invoked.

If the file ends in `.F` or `.F90`, the source preprocessor is invoked.

Files containing uncompiled Fortran code must be in one of the following forms:

- *file*`.f` or *file*`.F`, which indicates that this is a fixed source form file.
- *file*`.f90` or *file*`.F90`, which indicates that this is a free source form file.

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

By default, several files are created during processing. The CF90 compiler adds a suffix to the *file* portion of the file name and places the files it creates into your working directory.

Assembly language output is sent to *file*`.s`. File names ending with `.s` are assembled, and the assembled code is written to the corresponding *file*`.o`.

The compiled code is written to *file*`.o` in the current directory.

You can specify precompiled `.o` files as input files. Input file names ending with `.o` are passed to the loader.

If your input files include precompiled assembler, CF90, Cray Standard C, or Cray C++ files, they should be specified as files suffixed with `.o` on the command line. Files suffixed with `.o`, including any `.o` files written by the CF90 compiler, are passed to the loader in the order in which they appear on the `f90`(1) command line. If, however, loading has been disabled by specifying options `-c` or `-b` *bin_obj_file*, no files are passed to the loader.

The loader produces an executable file; by default, `a.out`. See the `-o` *out_file* option for information on specifying a different executable file. 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.

The following is a file summary:

File	Type
`a.out`	Executable output file.
file`.a`	Library files to be searched for external references.
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` or *file*`.F90`	Input Fortran source file in free source form. If *file* ends in `.F90`, 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.
`ftnlint.out`	`ftnlint`(1) output file created when the `-r l` option is specified and there are multiple source files.

`ftnlist.out`	`ftnlist`(1) output file created when the `-r` option is specified and there are multiple source files.

The loader allows other file types. See the `segldr`(1), `cld`(1), and `ld`(1) man pages for more information on these files.

2.3 Environment Variables

Environment variables are predefined shell variables, taken from the execution environment, that determine some of your shell characteristics. Several environment variables pertain to the CF90 compiler. The CF90 compiler recognizes general and multiprocessing environment variables. On UNICOS systems, it also recognizes a variable for CPU targeting.

The multiprocessing variables in the following sections affect the way your program will perform on multiple processors. Many of these control the same keywords as those of the `TSKTUNE`(3F) multitasking library call. Using environment variables lets you tune the system for parallel processing without rebuilding libraries or other system software.

The environment variables apply to all compilations in a session. The following examples show how to set an environment variable:

- With the standard shell, enter:

  ```
  TARGET=cpu
  export TARGET
  ```

- With the C shell, enter:

  ```
  setenv TARGET cpu
  ```

The following sections describe the environment variables recognized by the CF90 compiler.

Note: Many of the environment variables described in this chapter refer to the default system locations of programming environment components. If the CF90 programming environment has been installed in a nondefault location, see your system support staff for path information.

2.3.1 AUXBUF (UNICOS Systems Only)

The AUXBUF environment variable specifies the number of buffers, for auxiliary arrays, that are held in memory. The default value is 64 buffers.

2.3.2 AUXPAGE (UNICOS Systems Only)

The AUXPAGE environment variable specifies the size of each AUX buffer. The size is given in units of 512 words each and must be a power of 2. The default value is 2, indicating 1024 words per buffer.

2.3.3 CRI_F90_OPTIONS

The CRI_F90_OPTIONS environment variable specifies additional options to be attached to the command line. These options are added following the options specified directly on the command line. File names cannot appear. These options are inserted at the rightmost portion of the command line before the input files and binary files are listed.

This allows 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_F90_OPTIONS -G0
```

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

```
% f90 -c t.f
% f90 -c -G0 t.f
```

2.3.4 LD_OPTIONS (UNICOS/mk Systems Only)

The LD_OPTIONS environment variable specifies a default set of options to cld(1). cld(1) interprets LD_OPTIONS just as though its value had been placed on the command line immediately following cld(1). For more information on other cld(1) environment variables, see cld(1).

2.3.5 LISTIO_PRECISION

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

- FULL prints full precision (default).
- PRECISION prints *x* or *x* + 1 decimal digits, where *x* is 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.

2.3.6 LPP

The LPP environment variable controls page breaks in listings. Setting LPP to 0 prevents any page breaks from occuring in listings. The -r n option can also be used to suppress page breaks.

2.3.7 MP_DEDICATED (UNICOS Systems Only)

The MP_DEDICATED variable specifies the tasking environment.

If MP_DEDICATED is set to 1, it specifies that you are the only user on the system, which allows library scheduling to take advantage of the dedicated environment.

If MP_DEDICATED is set to 0 or not set at all, slave processors return to the operating system after waiting in user space for 50,000 clock periods.

If MP_DEDICATED is set to a value other than 0 or 1, the behavior is undefined.

2.3.8 MP_HOLDTIME (UNICOS Systems Only)

The MP_HOLDTIME tasking environment variable specifies the number of clock periods (CPs) to hold a processor before giving up the CPU when no parallel work is available.

In nondedicated mode, when MP_HOLDTIME clock periods have gone by without there being any parallel work for a processor, that processor yields its CPU back to the kernel for use by other processes in the system. This yield is temporary; the processors continue to exist, but they do not use as many

system resources while yielded. They reacquire a CPU when the next parallel region occurs in the program.

The default is 50,000 CPs.

2.3.9 MP_SAMPLE (UNICOS Systems Only)

The MP_SAMPLE tasking environment variable specifies the sample rate at which the ready mask is read when in the hold loop. The default is 150 CPs, which means that a process checks for a task every 150 CPs while it is waiting for parallel work.

2.3.10 MP_SLVSIN (UNICOS Systems Only)

The MP_SLVSIN tasking environment variable specifies the stack increment, in words, for slave processes.

2.3.11 MP_SLVSSZ (UNICOS Systems Only)

The MP_SLVSSZ environment variable specifies the initial task size, in words, for slave processes.

2.3.12 NCPUS (UNICOS Systems Only)

The NCPUS environment variable specifies the number of CPUs to use when running parallel code. The default is 4 or the number of physical CPUs, whichever is less. If the MP_DEDICATED environment variable is set, the default is the number of physical CPUs.

The value of the NCPUS environment variable overrides the value of the OMP_NUM_THREADS environment variable. For information on the OMP_NUM_THREADS environment variable, see Section 2.3.17, page 66.

2.3.13 NLSPATH

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

2.3.14 NPROC

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

The effect of NPROC is seen at compilation time, not at execution time. NPROC requests a number of compilations to be done in parallel. It affects all the compilers and also make(1).

For example, assume that NPROC is set as follows:

```
setenv NPROC 2
```

The following command is entered:

```
f90 -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.

2.3.15 OMP_DYNAMIC (UNICOS Systems Only)

The OpenMP Fortran API defines the OMP_DYNAMIC environment variable as one that can enable or disable the dynamic adjustment of threads available for execution of parallel regions. On UNICOS systems, however, the dynamic adjustment of threads is always enabled and cannot be disabled.

If a program calls the OMP_SET_DYNAMIC(3) library routine with an argument of .FALSE., intending to turn off dynamic adjustment of the number of threads, the library routine is ignored. The OMP_GET_DYNAMIC(3) routine always returns .TRUE. .

2.3.16 OMP_NESTED (UNICOS Systems Only)

The OpenMP Fortran API defines the OMP_NESTED environment variable as one that can enable or disable nested parallelism. On UNICOS systems, however, nested parallelism is not supported.

If a program calls the OMP_SET_NESTED(3) library routine with an argument of .TRUE., intending to turn on nested parallelism, the routine prints a warning message. The OMP_GET_NESTED(3) routine always returns .FALSE. .

2.3.17 OMP_NUM_THREADS (UNICOS Systems Only)

The OMP_NUM_THREADS environment variable can affect the number of threads used when executing a program that uses the OpenMP Fortran API directives.

When using the OpenMP directives, you can never create more threads than the number automatically created at the time the program starts. This number is the value of NCPUS, or if NCPUS is not defined, OMP_NUM_THREADS, or if OMP_NUM_THREADS is not defined, the system default. That is, the value of the NCPUS environment variable overrides the value of the OMP_NUM_THREADS environment variable. The default is 4 or the number of CPUs on the system, whichever is less.

If the value of NCPUS or OMP_NUM_THREADS is greater than the number of CPUs on the system, the number of threads initially created is the number of CPUs on the system. After the threads are created, no more than that number can ever be initiated.

Example 1. Assume that a program that uses OpenMP directives relies on having 16 threads for correct execution. This program must be run on a system with 16 or more CPUs, and it must be run with OMP_NUM_THREADS or NCPUS set to at least 16. This limitation is enforced by the library.

Example 2. Assume that a program calls the OMP_SET_NUM_THREADS(3) library routine to set the number of threads to 10. If the program is running with OMP_NUM_THREADS set to 4, a message is generated and the system sets the number of threads to the highest value possible, which is 4 in this example.

2.3.18 OMP_SCHEDULE (UNICOS Systems Only)

The OMP_SCHEDULE environment variable sets the schedule type and (optionally) the chunk size for DO and PARALLEL DO loops declared with a schedule of RUNTIME. For these loops, the schedule is set at run time when the system reads the value of this environment variable. Valid values for this environment variable are STATIC, DYNAMIC, and GUIDED. The default value for this environment variable is DYNAMIC.

For DO and PARALLEL DO directives that have a schedule type other than RUNTIME, this environment variable is ignored.

If the optional chunk size is not set, a chunk size of 1 is assumed.

Examples:

```
setenv OMP_SCHEDULE "GUIDED,4"
setenv OMP_SCHEDULE "dynamic"
```

2.3.19 SEGDIR (UNICOS Systems Only)

The SEGDIR environment variable specifies SEGLDR directives. This variable contains one or more strings separated by semicolons (;). Each string can be either a SEGLDR directive or the name of a file containing SEGLDR directives. For more information, see segldr(1).

2.3.20 TARGET

The TARGET environment variable specifies a cross-compiling environment. *Cross-compiling* is compiling a program on one system to execute on another.

The TARGET environment variable allows you to cross-compile from most UNICOS systems to other Cray Research UNICOS systems. Cross-compiling from a CRAY T90 system that supports IEEE floating-point arithmetic to another type of UNICOS system is not possible. Cross compiling from a UNICOS/mk system to a UNICOS system, or from a UNICOS system to a UNICOS/mk system, is not possible.

The target system is the type of machine upon which the code will be executed. The TARGET environment variable recognizes the following values:

Value	Generates code for
cray-sv1	CRAY SV1 systems
cray-t3e	UNICOS/mk systems
cray-ts	CRAY T90 systems without support for IEEE floating-point arithmetic
cray-ts,ieee	CRAY T90 systems with support for IEEE floating-point arithmetic
cray-c90	CRAY C90 systems
cray-j90	CRAY J90 systems
cray-ymp	CRAY Y-MP E and CRAY Y-MP M90 systems

`cray-el` CRAY EL systems

See the `target`(1) man page for more information on setting this environment variable.

It is recommended that you load your binary program on the target system. If you load it on the host system, issues over differences between the libraries needed on the target and host systems may arise. The `TARGET` settings do not automatically cause the correct library to be selected. In many cases, host and target libraries will be compatible, but there may be incompatibilities between host and target systems.

The need for a particular set of libraries, as well as the correct specification to identify the set, must be determined for your particular site. If you need to specify a set of libraries contained in a directory, use the `-L` *dir* option, as follows:

```
% f90 -L /lib/ylib pgm.f   # For a library directory
```

The following example shows how to compile a program on a CRAY J90 system and execute it on a CRAY C90 system.

On the CRAY J90 system:

1. Set the `TARGET` environment variable to `cray-c90` using the procedure described in this section.
2. Compile your program using a command line that includes the `-c` option, as follows:

   ```
   % f90 -c bigfile.f
   ```

   The `-c` option disables the load step and returns `bigfile.o`.

On the CRAY C90 system:

1. Move file `bigfile.o` from the CRAY J90 system to the CRAY C90 system.
2. Use the `f90`(1) command to link `bigfile.o` with the correct libraries:

   ```
   % f90 bigfile.o
   ```

   Note: The `TARGET` environment variable applies until the end of an interactive session, so it affects other commands that use this environment variable to modify different CPU types. You must cancel these settings if you do not want to use them. You can prevent unexpected effects by setting the environment variable in a script or in a make file; the setting is effective only within that file.

2.3.21 TMPDIR

The TMPDIR environment variable specifies the directory to contain temporary files. By default, the CF90 compiler creates temporary files in /var/tmp. You can specify a different location by setting TMPDIR to your chosen directory. If TMPDIR is not a valid directory, the CF90 compiler uses /var/tmp.

CF90 Directives [3]

Directives are lines inserted into source code that specify actions to be performed by the compiler. They are not Fortran 90 statements.

This chapter describes the CF90 directives and notes whether particular directives are supported on specific platforms. 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 CF90 compiler supports two other classes of directives: the OpenMP Fortran API directives, and the Autotasking directives. For information on the OpenMP directives, see Chapter 4, page 119. For information on the Autotasking directives, which are outmoded, see Appendix A, page 177.

Table 5 categorizes the CF90 directives according to purpose and platform. It also indicates the pages that contain the main descriptions of the individual directives.

Table 5. Directives

Purpose and Name	Operating system	Description
Vectorization and tasking:		
CNCALL	UNICOS	Section 3.2.1, page 78
COPY_ASSUMED_SHAPE	UNICOS, UNICOS/mk	Section 3.2.2, page 78
IVDEP	UNICOS, UNICOS/mk	Section 3.2.3, page 80
NEXTSCALAR	UNICOS, UNICOS/mk	Section 3.2.4, page 81
PERMUTATION	UNICOS	Section 3.2.6, page 82
PATTERN, NOPATTERN	UNICOS, UNICOS/mk	Section 3.2.5, page 81
PREFERTASK	UNICOS	Section 3.2.7, page 83
PREFERVECTOR	UNICOS	Section 3.2.8, page 83
RECURRENCE, NORECURRENCE	UNICOS, UNICOS/mk	Section 3.2.9, page 84
SHORTLOOP, SHORTLOOP128	UNICOS, UNICOS/mk	Section 3.2.10, page 86

Purpose and Name	Operating system	Description
TASK, NOTASK	UNICOS	Section 3.2.11, page 87
UNROLL, NOUNROLL	UNICOS, UNICOS/mk	Section 3.2.12, page 87
VECTOR, NOVECTOR	UNICOS, UNICOS/mk	Section 3.2.13, page 90
VFUNCTION	UNICOS	Section 3.2.14, page 90
VSEARCH, NOVSEARCH	UNICOS	Section 3.2.15, page 92
Inlining:		
INLINE, NOINLINE	UNICOS, UNICOS/mk	Section 3.3.1, page 93
INLINENEVER, INLINEALWAYS	UNICOS, UNICOS/mk	Section 3.3.2, page 93
MODINLINE, NOMODINLINE	UNICOS, UNICOS/mk	Section 3.3.3, page 94
Scalar optimization:		
ALIGN	UNICOS	Section 3.4.1, page 96
BL, NOBL	UNICOS, UNICOS/mk	Section 3.4.2, page 96
CACHE_BYPASS	UNICOS/mk	Section 3.4.3, page 99
NOINTERCHANGE	UNICOS, UNICOS/mk	Section 3.4.4, page 100
NOSIDEEFFECTS	UNICOS	Section 3.4.5, page 101
SPLIT, NOSPLIT	UNICOS/mk	Section 3.4.7, page 103
SUPPRESS	UNICOS, UNICOS/mk	Section 3.4.7, page 103
Local use of compiler features:		
BOUNDS, NOBOUNDS	UNICOS, UNICOS/mk	Section 3.5.1, page 105
FREE, FIXED	UNICOS, UNICOS/mk	Section 3.5.2, page 107
Storage:		
AUXILIARY	UNICOS	Section 3.6.1, page 108
CACHE_ALIGN	UNICOS/mk	Section 3.6.2, page 110
COMMON	UNICOS, UNICOS/mk	Section 3.6.3, page 110
STACK	UNICOS, UNICOS/mk	Section 3.6.4, page 111
SYMMETRIC	UNICOS/mk	Section 3.6.5, page 112
TASKCOMMON	UNICOS	Section 3.6.6, page 112
Miscellaneous:		

Purpose and Name	Operating system	Description
CONCURRENT	UNICOS/mk	Section 3.7.1, page 113
FLOW, NOFLOW	UNICOS	Section 3.7.2, page 114
ID	UNICOS, UNICOS/mk	Section 3.7.3, page 115
IGNORE_TKR	UNICOS, UNICOS/mk	Section 3.7.4, page 117
NAME	UNICOS, UNICOS/mk	Section 3.7.5, page 117
USES_EREGS	UNICOS/mk	Section 3.7.6, page 118

3.1 Using Directives

The following sections describe how to use directives and the effects they have on programs.

3.1.1 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$ Auxiliary
!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 CF90 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$).

3.1.2 Range and Placement of Directives

The range and placement of directives is as follows:

- The following directives can appear anywhere in your source code:
  - CACHE_ALIGN
  - FIXED, FREE

  All other directives must appear within a program unit.
- The following directives apply only to the program unit in which they appear:
  - COPY_ASSUMED_SHAPE
  - FLOW, NOFLOW
  - STACK
  - USES_EREGS

  The directives that are paired do not toggle their respective features within the program unit. The last directive of each pair encountered within the program unit applies to the whole program unit.

- The following directives toggle a compiler feature on or off at the point at which the directive appears in the code:
  - BL, NOBL
  - BOUNDS, NOBOUNDS
  - INLINE, NOINLINE
  - PATTERN, NOPATTERN
  - RECURRENCE, NORECURRENCE
  - TASK, NOTASK
  - VECTOR, NOVECTOR

  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.
- Similar to the previous grouping the SUPPRESS directive applies at the point at which it appears, but it causes no continuing condition that can be toggled.
- 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 encountered lexically:
  - ALIGN
  - CACHE_BYPASS
  - CNCALL
  - CONCURRENT
  - IVDEP
  - NEXTSCALAR
  - NOINTERCHANGE
  - PERMUTATION
  - PREFERTASK
  - PREFERVECTOR

  - SHORTLOOP, SHORTLOOP128
  - SPLIT, NOSPLIT
  - UNROLL, NOUNROLL
  - VSEARCH, NOVSEARCH

- The following directives do not apply to particular ranges of code:
  - AUXILIARY
  - COMMON
  - IGNORE_TKR
  - INLINEALWAYS, INLINENEVER
  - NAME
  - NOSIDEEFFECTS
  - SYMMETRIC
  - TASKCOMMON
  - VFUNCTION

  These declarative directives alter the status of entities in ways that affect compilation.

- 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.

3.1.3 Interaction of Directives with the -x Command Line Option

The -x option on the f90(1) 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 on the -x option, see Section 2.2.35, page 56.

3.1.4 Command Line Options and Directives

Some features activated by directives can also be specified on the `f90`(1) 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, 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 `f90`(1), the associated `!`*prefix*`$` directives are ignored. The following list shows CF90 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, and tasking directives are ignored.
- Specifying the `-O scalar0` option disables scalar optimization and causes the compiler to ignore all scalar optimization and all vectorization 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.
- Specifying the `-O task0` option disables tasking and causes the compiler to ignore tasking directives.

The following sections describe directive syntax and the effects of directives on CF90 programs.

3.2 Vectorization and Tasking Directives

The following sections describe the directives used to control vectorization and tasking, which are as follows:

- `CNCALL`
- `COPY_ASSUMED_SHAPE`
- `IVDEP`
- `NEXTSCALAR`
- `PATTERN`, `NOPATTERN`
- `PERMUTATION`

- PREFERTASK
- PREFERVECTOR
- RECURRENCE, NORECURRENCE
- SHORTLOOP, SHORTLOOP128
- TASK, NOTASK
- UNROLL, NOUNROLL
- VECTOR, NOVECTOR
- VFUNCTION
- VSEARCH, NOVSEARCH

The `-O 0`, `-O scalar0`, `-O task0`, and `-O vector0` options on the `f90`(1) command override these directives.

3.2.1 Declare Lack of Side Effects: CNCALL

The `!DIR$ CNCALL` directive allows a loop to be Autotasked by asserting that subroutines called from the loop have no loop-related side effects (that is, they do not modify data referenced in other iterations of the loop) and therefore can be called concurrently by separate iterations of the loop. `CNCALL` is inserted immediately preceding the loop.

The format for this directive is as follows:

```
!DIR$ CNCALL
```

Example:

```
!DIR$ CNCALL
      DO I = 1, N
        CALL CRUNCH(A(I), B(I))
      END DO
```

3.2.2 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.

3.2.3 Ignore 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 90 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. *Array syntax statements* are Fortran 90 methods for referencing arrays that are more compact than FORTRAN 77 methods. 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 ]
```

vlen Specifies a vector length in which no dependency will occur. If *vlen* is not specified, the vector length used is the maximum possible for the target machine.

Implementation of the *vlen* specification is deferred on UNICOS/mk systems.

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 CF90 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 CF90 compiler does not reorder loop statements. For information on vector dependencies, see *Optimizing Code on Cray PVP Systems*, publication SG–2192.

3.2.4 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.

3.2.5 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 is in effect, the PATTERN and NOPATTERN compiler directives are ignored. For more information on -O nopattern, see Section 2.2.20.18, page 35.

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
```

3.2.6 Declare an Array with No Repeated Values: PERMUTATION

The !DIR$ PERMUTATION directive declares that an integer array has no repeated values. This is useful when the integer array is used as a subscript for another array (vector-valued subscript). 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 both sides of the equal sign in a loop, many-to-one assignment is possible even when the subscript is identical. 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)) = A(IPNT(I)) + B(I)
      END DO
```

3.2.7 Designate Nested Loops: PREFERTASK (UNICOS Systems Only)

This directive allows loops with large iteration counts to be considered as candidates for Autotasking.

The PREFERTASK directive disables threshold checking. For more information on threshold checking, see Section 2.2.20.31, page 40.

This directive can be used if there is more than one loop in the nest that can be Autotasked. `-O task2` or `-O task3` must be enabled for this directive to take effect. The format of this directive is as follows:

```
!DIR$ PREFERTASK
```

In the following example, both loops can be Autotasked, but the PREFERTASK directive directs the compiler to Autotask the inner `DO J` loop. Without the directive and without any knowledge of `N` and `M`, the compiler would task the outer `DO I` loop. With the directive, the loops are interchanged, to increase parallel granularity, and the resulting outer `DO J` loop is autotasked.

```
      DO I = 1, N
!DIR$ PREFERTASK
        DO J = 1, M
          E(J,I) = F(J,I) + G(J,I)
        END DO
      END DO
```

3.2.8 Designate Loop Nest for Vectorization: PREFERVECTOR (UNICOS Systems Only)

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
```

3.2.9 Designate Reduction Loops: RECURRENCE, NORECURRENCE

The effect of the RECURRENCE and NORECURRENCE directives differs depending on your platform. Regardless of platform, however, the formats of these directives are as follows:

```
!DIR$ RECURRENCE

!DIR$ NORECURRENCE
```

The following sections describe the effects of the RECURRENCE and NORECURRENCE directives on different systems.

3.2.9.1 Using RECURRENCE and NORECURRENCE on UNICOS Systems

The RECURRENCE and NORECURRENCE directives control vectorization for all reduction loops within a program unit and override the -O recurrence option on the f90(1) command. The RECURRENCE directive causes loops that contain reductions to be vectorized; the NORECURRENCE directive causes some loops that contain reductions to go unvectorized. Either directive applies until the end of a program unit or until the opposite directive is encountered.

A *reduction loop* is a loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

The NORECURRENCE directive disables vectorization of any loop that contains a reduction in which the order of evaluation is numerically significant. The specific reductions that are disabled are floating point, double precision, complex summation and product reductions, and alternating value computations.

Example 1: The NORECURRENCE directive disables vectorization of the following loop:

```
!DIR$ NORECURRENCE
      SUM = 0.0
      DO I = 1,1000
           SUM = SUM + A(I)
      END DO
```

Example 2: In the following code, an alternating value computation is not strictly a reduction, but it is included under this heading because the vector version may generate more precise results than the scalar version. The loop in this code fragment would vectorize with the CF90 compiler default settings, but because of the NORECURRENCE directive, it does not vectorize.

```
      SUM     = 0.0
      PRODUCT = 1.0
      TOGGLE  = 5.0
!DIR$ NORECURRENCE
      DO I = 1,1000
        SUM     = SUM + A(I)     ! Summation reduction
        PRODUCT = PRODUCT * A(I) ! Product reduction
        TOGGLE  = 5.0 - TOGGLE   ! Alternating value
        B(I)    = TOGGLE         !        computation
      END DO
```

Both directives are ignored if vectorization is disabled or if `-O norecurrence` is specified on the f90(1) command line.

3.2.9.2 Using RECURRENCE and NORECURRENCE on UNICOS/mk Systems

The RECURRENCE and NORECURRENCE directives convert floating-point operations that involve multiplication or exponentiation by an induction

variable to be a series of additions or multiplications of a value. This produces faster code but can create numeric differences.

These directives apply to all reduction loops within a program unit, and they override the `-O recurrence` option on the `f90`(1) command. The `RECURRENCE` directive rewrites multiplication operations. Either directive applies until the end of a program unit or until the opposite directive is encountered.

The `NORECURRENCE` directive stops the compiler from rewriting multiplication operations for loops in which the order of evaluation is numerically significant.

Both directives are ignored if `-O norecurrence` is specified on the `f90`(1) command line or if scalar optimization has been disabled.

3.2.10 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 `SHORTLOOP` directive is supported on UNICOS and UNICOS/mk systems. The `SHORTLOOP128` directive is supported on UNICOS systems only.

The formats of these directives are as follows:

```
!DIR$ SHORTLOOP

!DIR$ SHORTLOOP 128
```

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* ≤ 64. If *trip_count* equals 0 or exceeds 64, results are unpredictable.
- If you specify `!DIR$ SHORTLOOP 128`, the loop trip count must be in the range $1 \leq$ *trip_count* ≤ 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 loop trip count is known at compile time and is greater than the target machine's vector length. The vector length of CRAY C90 systems and CRAY T90 systems is 128. The vector length of all other UNICOS systems is 64.

3.2.11 Enable and Disable Tasking: TASK and NOTASK (UNICOS Systems Only)

The NOTASK directive suppresses compiler attempts to task loops and disables recognition of !MIC$ (Autotasking) directives. NOTASK takes effect at the next statement and applies to the rest of the program unit unless it is superseded by a TASK directive. These directives are disabled if tasking is disabled. The formats of these directives are as follows:

```
!DIR$ TASK

!DIR$ NOTASK
```

When !DIR$ NOTASK has been used within the same program unit, !DIR$ TASK causes the compiler to resume its attempts to task loops and array syntax statements. After a TASK directive is specified, the compiler again attempts to Autotask loops and array syntax statements and !MIC$ (Autotasking) directives are again recognized.

The TASK directive affects subsequent loops. The NOTASK 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.

The TASK and NOTASK directives have no effect on OpenMP Fortran API directives described in Chapter 4, page 119.

3.2.12 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* must be a positive integer.

If you specify a value for *n*, the compiler does not attempt to determine the number of copies to generate based on the number of inner loops in the loop nest.

The subsequent DO loop is not unrolled if you specify UNROLL0, UNROLL1, or NOUNROLL.

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

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

The UNROLL directive can be used only on loops whose iteration counts can be calculated before entering the loop. If UNROLL is specified on loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only 1 loop, and the innermost loop can contain work.

The NOUNROLL directive inhibits loop unrolling and overrides a `-O unroll2` command line specification.

Note: UNROLL directives are ignored when `-O unroll0` is specified on the command line.

On UNICOS/mk systems, no unrolling is the default setting, so you must specify `-O unroll1` or `-O unroll2` to enable loop unrolling. For more information on the unrolling options, see Section 2.2.20.32, page 41.

On UNICOS systems, 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 (following) in which unroll-and-jam conditions can occur. Such loop unrolling is associated with compiler message `cf90-6005`. 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. The UNROLL directive does not affect unrolling associated with compiler message `cf90-8135`.

Example 1: 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: 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
```

3.2.13 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, vectorization is attempted for loops with a trip count of 3 or more, or with an unknown trip count.

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.

3.2.14 Specify a Vectorizable Function: `VFUNCTION` (UNICOS Systems Only)

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. !DIR$ VFUNCTION cannot be specified for internal or module procedures. !DIR$ VFUNCTION cannot be specified for functions within interface blocks. The format of this directive is as follows:

```
!DIR$ VFUNCTION f [,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 or have its interface specified in an interface body.
- *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.

3.2.15 Vectorize Search Loops: VSEARCH and NOVSEARCH (UNICOS Systems Only)

The VSEARCH directive indicates that the compiler should vectorize the search loop that follows. The NOVSEARCH directive disables vectorization of search loops. The formats of these directives are as follows:

```
!DIR$ VSEARCH

!DIR$ NOVSEARCH
```

A *search loop* is one that can be exited by means of a GO TO statement or EXIT statement.

The VSEARCH and NOVSEARCH directives are enabled only when vectorization is enabled. These directives override `-O vsearch`, but they do not override `-O novsearch`.

3.3 Inlining Directives

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

- INLINE, NOINLINE
- INLINEALWAYS, INLINENEVER
- MODINLINE, NOMODINLINE

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

- `-O inline`*n* and `-O inlinefrom`, described in Section 2.2.20.10, page 29.
- `-O modinline` and `-O nomodinline`, described in Section 2.2.20.13, page 33.

The following sections describe the inlining directives.

3.3.1 Disable or Enable Inlining for a Block of Code: INLINE and NOINLINE

The INLINE and NOINLINE 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. The formats of these directives are as follows:

```
!DIR$ INLINE

!DIR$ NOINLINE
```

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 inline0 is in effect.

3.3.2 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 ] ...
```

name The name of a procedure.

The following rules determine the scope of these directives:

- A !DIR INLINENEVER directive suppresses inlining for *name*. For example, 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*. For example, 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 inline1`:

```
      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
```

3.3.3 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 unless `-O modinline` is specified on the f90(1) 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 use associated must have been compiled with `-O modinline`.

Example: The following file is compiled as **`f90 -O modinline file.f`**:

```
      MODULE BEGIN
!DIR$ MODINLINE
      ...
      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`.

3.4 Scalar Optimization Directives

The following directives control aspects of scalar optimization:

- `ALIGN`

- BL and NOBL
- CACHE_BYPASS
- NOINTERCHANGE
- NOSIDEEFFECTS
- SPLIT and NOSPLIT
- SUPPRESS

The following sections describe these directives.

3.4.1 Align Loops on Buffer Boundaries: ALIGN (UNICOS Systems Only)

The ALIGN directive causes a block of code to begin on an instruction buffer boundary, as follows:

- Placing the directive immediately before a DO or DO WHILE statement causes the body of the loop to begin on an instruction buffer boundary.
- Placing the directive immediately before a statement containing a referenced label (for example, the first statement of an IF loop) causes the generated code for that statement to begin on an instruction buffer boundary.
- Placing the directive immediately before an ENTRY statement causes the generated code for the source code following the ENTRY statement to be aligned on an instruction buffer boundary.

The format of this directive is as follows:

```
!DIR$ ALIGN
```

The ALIGN directive is useful when program execution is dominated by specific small blocks of code. If such a block crosses a buffer boundary, the overhead caused by frequent reloading of instruction buffers hurts program performance. By aligning such a dominant loop on a buffer boundary, the ALIGN directive can decrease this overhead.

3.4.2 Bottom Load Operands: BL and NOBL

When scalar optimization is enabled, the NOBL directive causes the use of safe bottom loading within loops; the BL directive causes the compiler to use full

bottom loading. Safe bottom loading involves a small cost in performance but removes a possible cause of run-time errors (indicated by an `Operand Range Error` message). Either directive overrides the `-O bl` option on the `f90`(1) command. The formats of these directives are as follows:

```
!DIR$ BL

!DIR$ NOBL
```

Bottom loading is an optimization technique that improves program performance and, typically, does not cause an error. This section describes the purpose of bottom loading and why it can cause an error.

Full bottom loading, which is used only on eligible scalar loops, consists of prefetching operands during each iteration of a loop for use in the next iteration, so that the operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

The final prefetch, using the next computed address, can access a location outside the array. If this address is outside the user program area, your program can fail, generating an operand range error. An out-of-bounds address typically is accessed in the following situations:

- A loop with a very large increment, for example:

  ```
  DO I = 10, 1000000, 100000
  ... = A(I)
  ```

- An array reference with a variant index that is other than the first subscript, for example:

  ```
  DIMENSION A(1000,1000)
  DO J = 1, 1000
  ... = A(100,J)
  ```

  An increment of this size might occur unnoticed when it is defined by a variable.

Safe bottom loading disables operand prefetching on the final loop iteration. The run-time test required to detect the final iteration entails a cost in performance but allows bottom loading to be performed on previous iterations. Despite the extra cost, safe bottom loading is faster than no bottom loading. You can enable safe bottom loading by specifying the `-O nobl` option on the `f90`(1) command

or the NOBL directive. When the -O scalar0 or -O 0 option is specified on the f90(1) command, no bottom loading is performed. When -O scalar1 or -O scalar2 (the default) is in effect, safe bottom loading is performed. Full bottom loading is done at -O scalar3.

Note: When a range error is caused by bottom loading, this does not indicate either a fault in your source program or a compiling error. The error results from an attempt to go outside the range of an array address; the problem is not caused by exceeding the array subscript bounds (as specified in your source code), so it cannot be detected.

The scope of the BL and NOBL directives is a single program unit. Either directive applies for the remainder of the program unit or until the appearance of the other directive in the same program unit. Both directives can be specified in a single program unit, and both are ignored if scalar optimization is disabled by the -O scalar0 or -O 0 options on the f90(1) command.

Example:

```
...
REAL ARRAY(0:M)
...
INC = IRESULT
! Potentially large value
...
DO 10 I = 0, M, INC
...
Y = X*ARRAY(I) + ...
...
! Fetch next operand, ARRAY(I+INC)
10 CONTINUE
```

3.4.3 Bypass Cache References: CACHE_BYPASS (UNICOS/mk Systems Only)

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

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

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

The format of this directive is as follows:

```
!DIR$ CACHE_BYPASS array [, array ] ...
```

array An array name. Only arrays containing data of the following types can be named in the directive: INTEGER(KIND=8), REAL(KIND=8), LOGICAL(KIND=8), and COMPLEX(KIND=16).

array can be either an array or a Fortran 90 pointer to an array. Cray pointee arrays are permitted, as are allocatable and deferred-shape arrays. Arrays or Fortran 90 pointers that are components of objects of derived type cannot be named in the directive.

This directive should immediately precede the loop that contains arrays to be accessed through E registers. It applies to the loop that follows the directive, but it does not affect other loops in the program.

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

- The loop must be an inner loop.
- The loop must be vectorizable. You may need to use the IVDEP directive in conjunction with CACHE_BYPASS to ensure that the loop is processed.

- The base array or pointer within the loop must be invariant.

Example:

```
      ! REFERENCES OF ARRAYS A, B AND C BYPASS CACHE.
      ! REFERENCES TO IX AND D GO THROUGH CACHE.
      SUBROUTINE FOO(A,B,C,D,IX,N)
      DIMENSION A(*), B(*), C(*), D(*), IX(*)
!DIR$ CACHE_BYPASS A, B, C
      DO I = 1, N
        A(IX(I)) = B(IX(I)) + C(IX(I)) * D(I)
      END DO
      END
```

To see the most benefit from the CACHE_BYPASS directive, you may wish to enable loop unrolling. For information on the command line option to control unrolling, see Section 2.2.20.32, page 41. For information on the unrolling compiler directives, see Section 3.2.12, page 87.

This directive may disable the CRAY T3E stream buffer hardware feature for the entire application. This is done because the compiler cannot guarantee correctness on some UNICOS/mk platforms in terms of the interaction of the stream buffers and the E register operations generated by this directive. Disabling stream buffers can cause considerable performance degradation for other parts of your program. You can use the set_d_stream(3) library routine to reenable the stream buffer feature. Consult your system administrator to determine whether your program could benefit from using this library routine. See streams_guide(7) for details on how and when streams can be safely reenabled in the presence of E register operations.

3.4.4 Inhibit Loop Interchanging: NOINTERCHANGE

The NOINTERCHANGE directive inhibits the compiler's ability to interchange the loop that follows the directive with another inner or outer loop. The format of this directive is as follows:

```
!DIR$ NOINTERCHANGE
```

3.4.5 Determine Register Storage: NOSIDEEFFECTS (UNICOS Systems Only)

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 ensures to have 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 be intent IN; 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 on the PURE prefix, see *Fortran Language Reference Manual, Volume 2*, publication SR–3903.

3.4.6 Request Loop Splitting: SPLIT and NOSPLIT (UNICOS/mk Systems Only)

Loop splitting improves performance by making best use of the six stream buffers of UNICOS/mk systems. It achieves this by splitting an inner loop into a set of smaller loops, each of which allocates no more than six stream buffers, thus avoiding stream buffer thrashing. The stream buffer feature reduces memory latency and increases memory bandwidth by prefetching for long, small-strided sequences of memory references. The formats of these directives are as follows:

```
!DIR$ SPLIT

!DIR$ NOSPLIT
```

The SPLIT directive should be placed immediately before the DO statement of the loop that should be split. The SPLIT directive asserts that the loop can profit by splitting. It will not cause incorrect code.

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

The SPLIT directive also causes the original loop to be stripmined. (See the glossary for a definition of *stripmining*.) This is done to increase the potential for cache hits between the resultant smaller loops.

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

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

The NOSPLIT directive inhibits loop splitting and overrides a `-O split2` command line specification. For more information on the `-O` loop splitting options, see Section 2.2.20.25, page 38.

Example:

```
!DIR$ SPLIT
      DO I=1,1000
         A(I) = B(I) * C(I)
         T = D(I) + A(I)
         E(I) = F(I) + T * G(I)
```

```
         H(I) = H(I) + E(I)
      END DO
```

First, the compiler generates the following loop (notice the expansion of the scalar temporary T into the compiler temporary array TA):

```
DO I = 1, 1000
   A(I) = B(I) * C(I)
   TA(I) = D(I) + A(I)
END DO
DO I = 1, 1000
   E(I) = F(I) * TA(I) * G(I)
   H(I) = H(I) + E(I)
END DO
```

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

```
DO I1 = 1, 1000, 256
   I2 = MIN(I1+255, 1000)
     DO I = I1, I2
        A(I) = B(I) * C(I)
        TA(I-I1+1) = D(I) + A(I)
     END DO
     DO I = I1, I2
        E(I) = F(I) * TA(I-I1+1) * G(I)
        H(I) = H(I) + E(I)
     END DO
END DO
```

3.4.7 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`.

3.5 Local Use of Compiler Features

Certain directives provide local control over specific compiler features. They are as follows:

- `BOUNDS` and `NOBOUNDS`
- `FREE` and `FIXED`

The `-f` and `-i` command line options apply to an entire compilation, but these directives override any command line specifications for source form or integer length. The following sections describe these directives.

3.5.1 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 `f90`(1) 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. See Section 3.4.2, page 96, for information on bottom loading. On UNICOS systems, 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 of out of range for dimension 1 for
  array 'A' at line 3 in file 't.f' with bounds:
  Lower bound is 1
  Upper bound is 10
lib-1962 a.out: WARNING
  Subscript 11:20:1 is of out of range for dimension 1
  for array 'A' at line 5 in file 't.f' with bounds:
  Lower bound is 1
```

```
Upper bound is 10
```

3.5.2 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.

Note: The source preprocessor does not recognize the FREE and FIXED directives. These directives must not be specified in a file that is submitted to the source preprocessor. To specify source form, specify -f fixed or the -f free option on the f90(1) command line.

3.6 Storage Directives

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

- AUXILIARY
- CACHE_ALIGN
- COMMON
- STACK
- SYMMETRIC
- TASKCOMMON

The following sections describe these directives.

3.6.1 Allocating to SSD: AUXILIARY (UNICOS Systems Only)

The AUXILIARY directive causes the compiler to allocate arrays and common blocks to the SSD solid-state storage device. This directive is intended for programs with very large data structures. The SSD, configured as secondary memory, functions much the same as main memory and considerably faster than a disk drive. When an entity has been allocated to the SSD, that entity becomes *auxiliary*; for example, an *auxiliary array*.

This directive must appear after the declaration of auxiliary arrays or common blocks but before the first executable statement in any program unit that references them.

If you want to specify auxiliary data in the specification part of a module, the data must also be declared in a common block.

The format of this directive is as follows:

```
!DIR$ AUXILIARY n [, n ] ...
```

n Symbolic name of an array to be allocated to the SSD.

Note: Any arrays or scalar variables that are in the same common block as an auxiliary array or are equivalenced to an auxiliary array become auxiliary variables.

Example 1: In the following code example, arrays A and B reside on the SSD. In addition, because array T is also specified in !DIR$ AUXILIARY, array U and variables X, Y, and Z are auxiliary because they appear in common block COMBLK along with T. COMBLK contains a mixture of arrays and scalar variables; individual variables can be allocated to the SSD only as members of a common block that contains an auxiliary array.

```
      DIMENSION A(1000000), B(1000000)
      COMMON /COMBLK/ T(1000000), U(1000000), X, Y, Z
!DIR$ AUXILIARY A, B, T
```

Example 2: In the following code example, element A(1,1) is equivalenced to element X(50,50) of auxiliary array X; therefore, A becomes an auxiliary array.

```
      COMMON /COM_BLK/ X(100,100), Y(1000), Z(100,100)
      DIMENSION A(100,100)
      EQUIVALENCE (A(1,1), X(50,50))
!DIR$ AUXILIARY X
```

3.6.1.1 Restrictions

The AUXILIARY directive has the following restrictions:

- The name of an auxiliary array or scalar variable must not appear in an I/O, DATA, or Cray pointer statement.
- If an auxiliary array or scalar variable is passed as an actual argument to a subprogram, the corresponding dummy argument must also have been declared auxiliary. This does not apply to intrinsic functions or to functions declared as VFUNCTIONs.
- The name of an auxiliary array or scalar variable must not appear as an argument to the LOC(3I) or CLOC(3I) intrinsic functions.
- Arrays in blank common or those declared by TASK COMMON must not be declared auxiliary.
- Character arrays and arrays in common blocks containing character scalar variables or arrays must not be declared auxiliary.
- Auxiliary array elements and vectors should not be used in signal-handling routines.
- Arrays of structures containing character components or pointer components must not be declared AUXILIARY.
- An auxiliary array must not have the attributes POINTER, TARGET, or ALLOCATABLE.

3.6.1.2 Auxiliary Arrays and Memory

Auxiliary arrays or scalar variables are read from the SSD solid-state storage device and stored in memory in buffers to read and write to the SSD. Two environment variables, AUXBUF and AUXPAGE, allow you to specify the size and number of these buffers. Both variables are examined at the beginning of program execution. For more information on these environment variables, see Section 2.3.1, page 62, and Section 2.3.2, page 62.

For array syntax statements that need compiler-generated temporary arrays (called *temps*), the temps are always allocated on the heap. This means that if the array syntax involves auxiliary arrays, there may be insufficient space on the heap to create the temps to perform the assignment. Temps are sometimes required when the same array appears on both sides of the equal sign or when a function or subroutine call requires an array temp (either for an argument or for a function result).

Example:

```
      DIMENSION A(100000)
!DIR$ AUXILIARY A
      A(1:100000) = A(100000:1:-1)
```

The preceding is processed in the following form, where T is a temp array:

```
T=A(100000:1:-1)
A = T
```

3.6.2 Align on Cache Line Boundaries: CACHE_ALIGN (UNICOS/mk Systems Only)

The CACHE_ALIGN directive aligns each specified variable on a cache line boundary. This is useful for frequently referenced variables. A *cache* is storage that can be accessed more quickly than conventional memory. A *cache line* is a division within a cache. When properly used, this directive can minimize cache conflicts.

The directive's effect is independent of its position in source code. The format of this directive is as follows:

```
!DIR$ CACHE_ALIGN var [, var ] ...
```

var A common block name or a PE-private variable name, separated from an adjoining *var* by a comma. If specifying a common block name, the names of the items in the common block need not be specified. A common block name must be in the following form:

/ *common_block_name* /

var cannot have the ALLOCATABLE, POINTER, or TARGET attributes, and *var* cannot appear in a Cray POINTER statement.

3.6.3 Declare Common Blocks Global to All Tasks: COMMON (UNICOS Systems Only)

The COMMON directive overrides the -a taskcommon command line specification for a given procedure. This information is not inherited from the parent scoping unit. All common blocks remain as single-copy common blocks when this directive appears in a procedure. The format of this directive is as follows:

```
!DIR$ COMMON b [, b ] ...
```

b Common block name. The name must also appear as a common block name in a COMMON statement; for information on the COMMON statement, see the *Fortran Language Reference Manual, Volume 1*, publication SR–3902.

Common blocks named in this directive cannot also be named in a TASKCOMMON directive. All procedures that reference a common block named in this directive that are compiled with `-a taskcommon` must declare the block in a COMMON directive.

For more information on the `-a taskcommon` command line specification, see Section 2.2.1, page 6.

3.6.4 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 `-e v` command line option in specific program units of a compilation unit. For more information on the `-e v` command line option, see Section 2.2.6, page 9.

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.

3.6.5 Declare Local Addressing: SYMMETRIC (UNICOS/mk Systems Only)

The SYMMETRIC directive declares that a PE-private stack variable has the same local address on all PEs. This is useful for global addressing using the SHMEM library routines. For more information on the SHMEM library routines, see intro_shmem(3). The format of this directive is as follows:

```
!DIR$ SYMMETRIC var [, var ] ...
```

var A variable that is PE private. Variables that are specified are allocated at the same local address on all PEs. There is an implicit barrier before the first executable statement in the routine.

When specified without the optional *var* argument, all PE-private stack variables are declared symmetric. All local variables are allocated at the same local address on all PEs.

All PEs must participate in the allocation of symmetric stack variables.

3.6.6 Declare Common Blocks Local to Each Task: TASKCOMMON (UNICOS Systems Only)

When multitasking is used, some common blocks may need to be local to a task. The TASKCOMMON directive declares all variables in a common block to be local to a task; the common block is referred to as a *task common block.* If multiple tasks execute code containing the same task common block, each task will have a separate copy of the block.

The TASKCOMMON directive must appear before the first executable statement of a program unit. Task common blocks must be named. A task common block is allocated at task invocation. The format of this directive is as follows:

```
!DIR$ TASKCOMMON cb [, cb ] ...
```

cb Common block name. Common blocks named in this directive must not also be named in a COMMON directive.

If *cb* is declared as TASKCOMMON, *cb* must be declared as TASKCOMMON in all program units that reference *cb.*

Using !DIR$ TASKCOMMON with the COMMON statement is equivalent to using the TASK COMMON statement, described in the *Fortran Language Reference Manual, Volume 3*, publication SR–3905. This directive is recommended because

it increases code portability; a different compiler would recognize the COMMON statement and ignore the directive, for example:

```
      COMMON /CB1/ A,B  /CB2/ C,D
!DIR$ TASKCOMMON CB1, CB2
```

Arrays in a task common block must not be declared as auxiliary. With these exceptions, these variables can be used like other variables in common storage.

Task common blocks are always given stack allocation, regardless of the kind of allocation used for other data. Unlike other stack variables, task common variables can be initially defined in a DATA statement or a type declaration statement. Such initialization of task common variables is supported with the SEGLDR 9.2, UNICOS 9.2, and libc 9.2 or later releases.

3.7 Miscellaneous Directives

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

- CONCURRENT
- FLOW and NOFLOW
- ID
- IGNORE_TKR
- NAME
- USES_EREGS

3.7.1 Specify Array Dependencies: CONCURRENT (UNICOS/mk Systems Only)

The CONCURRENT directive conveys array dependency information to the compiler. This directive is useful when software pipelining is requested on 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 consecutive loop iterations that can be executed in parallel without danger of data conflict. The *n* argument allows you to specify a collision distance of *n* or more iterations for all data dependencies.

If SAFE_DISTANCE=*n* is not specified, the distance is assumed to be infinite, and the compiler ignores all cross-iteration data dependencies.

This directive affects the loop that immediately follows it.

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)
```

For more information on software pipelining, see Section 2.2.20.19, page 36.

3.7.2 Flowtracing Directives: FLOW and NOFLOW (UNICOS Systems Only)

The FLOW and NOFLOW directives control the flowtrace(1) feature and override the -e f or -d f options on the f90(1) command line. flowtrace(1) monitors your program during execution and generates a dynamic call tree for your program. It also collects calling and timing information for each called procedure. These directives also control the perftrace(1) utility. flowtrace(1) and perftrace(1) are described in the *UNICOS Performance Utilities Reference Manual*, publication SR–2040.

If the -e f option is not specified, !DIR$ FLOW activates flowtrace(1) for the subprogram that contains it. If flowtrace(1) is activated by the presence of the -e f option on the command line, !DIR$ NOFLOW deactivates flowtrace(1) for the subprogram that contains it. The two directives cannot be used to toggle flowtrace(1) on and off within a program unit. The formats of these directives are as follows:

```
!DIR$ FLOW

!DIR$ NOFLOW
```

These directives apply to the entire program unit in which they appear. If both directives appear in the same program unit, the last one encountered takes effect for the entire program unit.

3.7.3 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`(1) command. To use the `what`(1) 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 1997'
        PRINT *, 'Hello, world'
        END
  ```

  The next step is to use file `id.o` as the argument to the `what`(1) command, as follows:

  ```
  % what id.o
  % id.o:
  %   file.f 03 February 1997
  ```

  Note that `what`(1) 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`(1) does not recognize the string.

  Input file `id2.o` contains the following:

  ```
  !DIR$ ID  'file.f 03 February 1997'
        PRINT *, 'Hello, world'
        END
  ```

The what(1) command generates the following output:

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

- Method 2 — Using strings(1) or od(1). The following example shows how to obtain output using the strings(1) command.

  Input file id.f contains the following:

```
!DIR$ ID  "File: id.f  Date: 03 February 1997"
      PRINT *, 'Hello, world'
      END
```

  The strings(1) command generates the following output:

```
% strings id.o
02/03/9713:55:52f90
3.0.0.0.9.0.2cn
CRAY-YMP
$MAIN
@CODE
@DATA
@WHAT
$MAIN
$STKOFEN
f$init
_FWF
$END
*?$F(6(
Hello, world
$MAIN
File: id.f  Date: 03 February 1997
% 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   7  \0  \0  \0  \0  \0  \0
```

... portion of dump deleted

3.7.4 Disregard Dummy Argument Type, Kind, and Rank: IGNORE_TKR

The IGNORE_TKR directive directs the compiler to ignore the type, kind, and rank (TKR) of specified dummy arguments in a procedure interface. For information on Fortran 90 TKR rules, see chapters 4 and 6 of the *Fortran Language Reference Manual, Volume 2*, publication SR–3903.

The format for this directive is as follows:

```
!DIR$ IGNORE_TKR [ darg_name [ , darg_name ] ... ]
```

darg_name If specified, 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 type and kind and rank of the specified dummy arguments when resolving a generic to a specific call. The compiler also ignores type and kind and 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 rank rules for the dummy arguments supplied for the SHMEM_PUT64(3) function call:

```
      INTERFACE SHMEM_PUT64
         SUBROUTINE SHMEM_PUT64(TARG, SRC, LEN, PE)
!DIR$    IGNORE_TKR TARG, SRC
         INTEGER(KIND=4) LEN
         INTEGER(KIND=4) PE
         END SUBROUTINE SHMEM_PUT64
      END INTERFACE
```

The preceding code specifies that TARG and SRC can be any data type, but LEN and PE must be INTEGER(KIND=4) data.

3.7.5 External Name Mapping Directive: 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
```

3.7.6 Reserve E Registers: USES_EREGS (UNICOS/mk Systems Only)

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

```
!DIR$ USES_EREGS
```

USES_EREGS applies only to the program unit in which it appears. Your code must comply with E register conventions as described in *Cray Assembler for MPP (CAM) Reference Manual*, publication SR–2510.

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

OpenMP Fortran API Directives (UNICOS Systems Only) [4]

This chapter describes the multiprocessing directives that the CF90 compiler supports. These directives are based on the OpenMP Fortran application program interface (API) standard. Programs that use these directives are portable and can be compiled by other compilers that support the OpenMP standard.

Note: Programs containing OpenMP directives must be compiled on a system running UNICOS 10.0.0.3, or later, and must be loaded with `segldr`(1) 9.2, or later.

In releases prior to CF90 3.1, lines beginning with `!$`, `C$`, or `*$` were always treated as comments. With the introduction of the OpenMP Fortran API, these lines are now treated as conditional compilation lines and are compiled as source code when tasking is in effect. To have these lines treated as comments, remove the dollar sign (`$`) from these lines or compile with the `-x conditional_omp` command line option. For more information on the `-x conditional_omp` command line option, see Section 2.2.35, page 56.

In addition to directives, the OpenMP Fortran API describes several library routines and environment variables. Information on the library routines and environment variables can be found on the `omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages. Information on the environment variables can also be found in Section 2.3, page 61.

The sections in this chapter are as follows:

- Section 4.1, page 120, describes using directives and the directive format.
- Section 4.2, page 122, describes conditional compilation.
- Section 4.3, page 123, describes the parallel region construct.
- Section 4.4, page 125, describes work-sharing constructs.
- Section 4.5, page 131, describes the combined parallel work-sharing constructs.
- Section 4.6, page 135, describes synchronization constructs.
- Section 4.7, page 141, describes the data environment, which includes directives and clauses that affect the data environment.

- Section 4.8, page 151, describes directive binding.
- Section 4.9, page 153, describes directive nesting.
- Section 4.10, page 156 and Section 4.11, page 165 describe optimization.

Note: The Cray Research Autotasking directives are outmoded. Their preferred alternatives are the OpenMP Fortran API directives described in this chapter.

The Autotasking directives and OpenMP directives can be mixed in the same program unit, but they cannot work together. For example, you cannot put an OpenMP directive inside a `!MIC$` parallel region. They must be independent.

For more information on the Autotasking directives, see Appendix A, page 177.

4.1 Using Directives

All multiprocessing directives are case-insensitive and are of the following form:

prefix directive [*clause*[[,] *clause*]...]

prefix Each directive begins with a prefix, and the prefixes you can use depend on your source form, as follows:

- If you are using fixed source form, the following prefixes can be used: `!$OMP`, `C$OMP`, or `*$OMP`.

  Prefixes must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the directive line.

- If you are using free source form, the following prefix can be used: `!$OMP`.

  A prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free form line length, case sensitivity, white space, and continuation rules apply to the directive line.

directive The name of the directive.

clause One or more directive clauses. Clauses can appear in any order after the directive name and can be repeated as needed, subject to the restrictions listed in the description of each clause.

Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Comments cannot appear on the same line as a directive.

In fixed source form, initial directive lines must have a space or zero in column six, and continuation directive lines must have a character other than a space or a zero in column six.

In free source form, initial directive lines must have a space after the prefix. Continued directive lines must have an ampersand as the last nonblank character on the line. Continuation directive lines can have an ampersand after the directive prefix with optional white space before and after the ampersand.

Example 1 (fixed source form). The following formats for specifying directives are equivalent (the first line represents the position of the first 9 columns):

```
C23456789
!$OMP PARALLEL DO SHARED(A,B,C)

C$OMP PARALLEL DO
C$OMP+SHARED(A,B,C)

C$OMP PARALLELDOSHARED(A,B,C)
```

Example 2 (free source form). The following formats for specifying directives are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
       !$OMP PARALLEL DO &
                 !$OMP SHARED(A,B,C)

!$OMP PARALLEL &
       !$OMP&DO SHARED(A,B,C)

      !$OMP PARALLEL DO SHARED(A,B,C)
```

Note: In order to simplify the presentation, the remainder of this chapter uses the !$OMP prefix in all syntax descriptions and examples.

4.2 Conditional Compilation

Fortran statements can be compiled conditionally as long as they are preceded by one of the following conditional compilation prefixes: `!$`, `C$`, or `*$`. The prefix must be followed by a Fortran 90 statement on the same line. During compilation, the prefix is replaced by two spaces, and the rest of the line is treated as a normal Fortran statement.

The `!$` prefix is accepted when compiling either fixed source form files or free source form files. The `C$` and `*$` prefixes are accepted only when compiling fixed source form. The source form you are using also dictates the following:

- In fixed source form, the prefixes must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the line. Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six.

  Example. The following forms for specifying conditional compilation are equivalent:

```
C23456789
!$ 10 IAM = OMP_GET_THREAD_NUM() +
!$   &          INDEX

#ifdef _OPENMP
   10 IAM = OMP_GET_THREAD_NUM() +
     &          INDEX
#endif
```

- In free source form, the `!$` prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free source form line length, case sensitivity, white space, and continuation rules apply to the line. Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix, with optional white space before and after the ampersand.

In addition to the conditional compilation prefixes, a preprocessor macro, `_OPENMP`, can be used for conditional compilation. The `_OPENMP` conditional compilation macro is predefined whenever tasking is enabled. That is, `_OPENMP` is predefined unless one of the following options appears on the `f90`(1) command line: `-O 0`, `-O task0`, `-g`, or `-G 0`. For more information on source preprocessing and conditional compilation, see Chapter 5, page 167.

Specifying `-x conditional_omp` on the `f90`(1) command line disables all `C$` and `!$` conditional compilation lines. For more information on the `-x conditional_omp` option, see Section 2.2.35, page 56.

Example. The following example illustrates the use of the conditional compilation prefix. Assuming Fortran 90 fixed source form, the following statement is invalid when using OpenMP constructs:

```
C234567890
!$  X(I) = X(I) + XLOCAL
```

With OpenMP compilation, the conditional compilation prefix `!$` is treated as two spaces. As a result, the statement infringes on the statement label field. To be valid, the statement should begin after column six, like any other fixed source form statement:

```
C234567890
!$    X(I) = X(I) + XLOCAL
```

In other words, conditionally compiled statements need to meet all applicable language rules when the prefix is replaced with two spaces.

4.3 Parallel Region Constructs (PARALLEL and END PARALLEL Directives)

The PARALLEL and END PARALLEL directives define a *parallel region*. A parallel region is a block of code that is to be executed by multiple threads in parallel. This is the fundamental OpenMP parallel construct that starts parallel execution. These directives have the following format:

```
!$OMP PARALLEL [clause[[,] clause]...]

block

!$OMP END PARALLEL
```

clause — *clause* can be one or more of the following:

- PRIVATE(*var*[, *var*] ...)
- SHARED(*var*[, *var*] ...)
- DEFAULT(PRIVATE | SHARED | NONE)
- FIRSTPRIVATE(*var*[, *var*] ...)

- REDUCTION ({*operator*|*intrinsic*}:*var*[, *var*] ...)
- IF(*scalar_logical_expression*)
- COPYIN(*var*[, *var*] ...)

The IF clause is described in this section. For information on the PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, and COPYIN clauses, see Section 4.7.2, page 142.

block — *block* denotes a structured block of Fortran statements. You cannot branch into or out of the block. The code contained within the dynamic extent of the parallel region is executed on each thread.

The END PARALLEL directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution past the end of a parallel region.

When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The master thread is a member of the team and it has a thread number of 0 within the team. The number of threads in the team is controlled by environment variables and/or library calls.

The number of physical processors actually hosting the threads at any given time depends on the number of CPUs available and the system load. Once created, the number of threads in the team remains constant for the duration of that parallel region, but it can be changed either explicitly by the user or automatically by the run-time system from one parallel region to another. The OMP_SET_DYNAMIC(3) library routine and the OMP_DYNAMIC environment variable can be used to enable and disable the automatic adjustment of the number of threads. For more information on environment variables that affect OpenMP directives, see Section 2.3, page 61.

OpenMP: The OpenMP Fortran API does not specify the number of physical processors that can host the threads at any given time.

If a thread in a team executing a parallel region encounters another parallel region, it creates a new team, and it becomes the master of that new team. By default, nested parallel regions are serialized; that is, they are executed by a team composed of one thread. This default behavior can be changed by using either the OMP_SET_NESTED(3) library routine or the OMP_NESTED environment variable. For more information on environment variables that affect OpenMP directives, see Section 2.3, page 61.

If an IF clause is present, the enclosed code region is executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. The expression must be a scalar Fortran logical expression.

The following restrictions apply to parallel regions:

- The PARALLEL/END PARALLEL directive pair must appear in the same routine in the executable section of the code.
- The code contained by these two directives must be a structured block. You cannot branch into or out of a parallel region.
- Only a single IF clause can appear on the directive.

Example. The PARALLEL directive can be used for exploiting coarse-grained parallelism. In the following example, each thread in the parallel region decides what part of the global array X to work on based on the thread number:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      NP =  OMP_GET_NUM_THREADS()
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

4.4 Work-sharing Constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed within a parallel region in order for the directive to execute in parallel. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The following restrictions apply to the work-sharing directives:

- Work-sharing constructs and BARRIER directives must be encountered by all threads in a team or by none at all.
- Work-sharing constructs and BARRIER directives must be encountered in the same order by all threads in a team.

The following sections describe the work-sharing directives:

- Section 4.4.1, page 126, describes the DO and END DO directives.

- Section 4.4.2, page 129, describes the SECTIONS, SECTION, and END SECTIONS directives.
- Section 4.4.3, page 130, describes the SINGLE and END SINGLE directives.

4.4.1 Specify Parallel Execution: DO and END DO Directives

The DO directive specifies that the iterations of the immediately following DO loop must be divided among the threads in the parallel region. If there is no enclosing parallel region, the DO loop is executed serially.

The loop that follows a DO directive cannot be a DO WHILE or a DO loop without loop control.

The format of this directive is as follows:

```
!$OMP DO [clause[[,] clause]...]

do_loop

[!$OMP END DO [NOWAIT]]
```

clause *clause* can be one of the following:

- PRIVATE(*var*[, *var*] ...)
- FIRSTPRIVATE(*var*[, *var*] ...)
- LASTPRIVATE(*var*[, *var*] ...)
- REDUCTION({*operator*|*intrinsic*}:*var*[, *var*] ...)
- SCHEDULE(*type*[,*chunk*])
- ORDERED

The SCHEDULE and ORDERED clauses are described in this section. The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 4.7.2, page 142.

do_loop A DO loop.

If ordered sections are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. The code enclosed within an ordered section is executed in the order in which it would be executed in a sequential execution

of the loop. For more information on ordered sections, see the ORDERED directive in Section 4.6.6, page 140.

The SCHEDULE clause specifies how iterations of the DO loop are divided among the threads of the team. Within the SCHEDULE(*type*[,*chunk*]) clause syntax, *type* can be one of the following:

type	Effect
STATIC	When SCHEDULE(STATIC,*chunk*) is specified, iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. *chunk* must be a scalar integer expression. When no *chunk* is specified, the iterations are divided among threads in contiguous pieces, and one piece is assigned to each thread. Deferred Implementation.
DYNAMIC	When SCHEDULE(DYNAMIC,*chunk*) is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes its iterations, it dynamically obtains the next set of iterations. When no *chunk* is specified, it defaults to 1. Performance, however, may be better when *chunk* is set to a small multiple of the vector length of your machine. This is particularly true when the loop body is small. The vector length of CRAY SV1, CRAY J90, CRAY Y-MP E, CRAY Y-MP M90, and CRAY EL systems is 64. The vector length of CRAY C90 and CRAY T90 systems is 128. This is the default SCHEDULE *type*.
GUIDED	When SCHEDULE(GUIDED,*chunk*) is specified, each of the iterations are handed out in pieces of exponentially decreasing size. *chunk* specifies the minimum number of iterations to dispatch each time, except when there are less than *chunk* number of iterations, at which point the rest are dispatched. When no *chunk* is specified, it defaults to 1.
RUNTIME	When SCHEDULE(RUNTIME) is specified, the decision regarding scheduling is deferred until run time and you cannot specify a *chunk*.

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 resulting schedule is DYNAMIC.

For more information on the OMP_SCHEDULE environment variable, see Section 2.3, page 61.

OpenMP: The OpenMP Fortran API does not define a default scheduling mechanism. You should not rely on a particular implementation of a schedule type for correct execution because it is possible to have variations in the implementations of the same schedule type across different compilers.

If an END DO directive is not specified, it is assumed at the end of the DO loop. If NOWAIT is specified on the END DO directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instructions following the loop without waiting for the other members of the team to finish the DO directive.

Example. If there are multiple independent loops within a parallel region, you can use the NOWAIT clause to avoid the implied BARRIER at the end of the DO directive, as follows:

```
!$OMP PARALLEL
!$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END DO NOWAIT
!$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Parallel DO loop control variables are block-level entities within the DO loop. If the loop control variable also appears in the LASTPRIVATE variable list of the parallel DO, it is copied out to a variable of the same name in the enclosing PARALLEL region. The variable in the enclosing PARALLEL region must be SHARED if it is specified on the LASTPRIVATE variable list of a DO directive.

The following restrictions apply to the DO directives:

- You cannot branch out of a DO loop associated with a DO directive.

- The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.
- The DO loop iteration variable must be of type integer.
- If used, the END DO directive must appear immediately after the end of the loop.
- Only a single SCHEDULE clause can appear on a DO directive.
- Only a single ORDERED clause can appear on a DO directive.

4.4.2 Mark Code for Specific Threads: SECTION, SECTIONS and END SECTIONS Directives

The SECTIONS directive specifies that the enclosed sections of code are to be divided among threads in the team. It is a noniterative work-sharing construct. Each section is executed once by a thread in the team.

The format of this directive is as follows:

```
!$OMP SECTIONS [clause[[,] clause]...]

[!$OMP SECTION]

block

[!$OMP SECTION

block]

. . .

!$OMP END SECTIONS [NOWAIT]
```

clause The *clause* can be one of the following:

- PRIVATE(*var*[, *var*] ...)
- FIRSTPRIVATE(*var*[, *var*] ...)
- LASTPRIVATE(*var*[, *var*] ...)
- REDUCTION({ *operator*|*intrinsic*}:*var*[, *var*] ...)

The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 4.7.2, page 142.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

Each section must be preceded by a SECTION directive, though the SECTION directive is optional for the first section. The SECTION directives must appear within the lexical extent of the SECTIONS/END SECTIONS directive pair. The last section ends at the END SECTIONS directive. Threads that complete execution of their sections wait at a barrier at the END SECTIONS directive unless a NOWAIT is specified.

The following restrictions apply to the SECTIONS directive:

- The code enclosed in a SECTIONS/END SECTIONS directive pair must be a structured block. In addition, each constituent section must also be a structured block. You cannot branch into or out of the constituent section blocks.
- You cannot have a SECTION directive outside the lexical extent of the SECTIONS/END SECTIONS directive pair.

4.4.3 Request Single–thread Execution: SINGLE and END SINGLE Directives

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE directive wait at the END SINGLE directive unless NOWAIT is specified.

The format of this directive is as follows:

```
!$OMP SINGLE [clause[[,] clause]...]

 block

!$OMP END SINGLE [NOWAIT]
```

clause The *clause* can be one of the following:

- PRIVATE(*var*[, *var*] ...)
- FIRSTPRIVATE(*var*[, *var*] ...)

The PRIVATE and FIRSTPRIVATE clauses are described in Section 4.7.2, page 142.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

Example. In the following code fragment, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT. You must not make any assumptions as to which thread will execute the SINGLE section. All other threads will skip the SINGLE section and stop at the barrier at the END SINGLE construct. If other threads can proceed without waiting for the thread executing the SINGLE section, a NOWAIT clause can be specified on the END SINGLE directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE
      CALL WORK(Y)
!$OMP END PARALLEL
```

4.5 Combined Parallel Work-sharing Constructs

The combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a PARALLEL directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing directives:

- Section 4.5.1, page 131, describes the PARALLEL DO and END PARALLEL DO directives.
- Section 4.5.2, page 133, describes the PARALLEL SECTIONS and END PARALLEL SECTIONS directives.

4.5.1 Declare a Parallel Region: PARALLEL DO and END PARALLEL DO Directives

The PARALLEL DO directive provides a shortcut form for specifying a parallel region that contains a single DO directive.

The format of this directive is as follows:

```
!$OMP PARALLEL DO [clause[[,] clause]...]

do_loop

[!$OMP END PARALLEL DO]
```

clause *clause* can be one or more of the clauses accepted by the PARALLEL directive or the DO directive. These clauses are as follows:

- PRIVATE(*var*[, *var*] ...)
- FIRSTPRIVATE(*var*[, *var*] ...)
- LASTPRIVATE(*var*[, *var*] ...)
- REDUCTION({*operator*|*intrinsic*}:*var*[, *var*] ...)
- SCHEDULE(*type*[,*chunk*])
- ORDERED
- SHARED(*var*[, *var*] ...)
- DEFAULT(PRIVATE | SHARED | NONE)
- IF(*scalar_logical_expression*)
- COPYIN(*var*[, *var*] ...)

The SCHEDULE and ORDERED clauses are described in Section 4.4.1, page 126. The IF clause is described in Section 4.3, page 123. The SHARED, DEFAULT, COPYIN, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 4.7.2, page 142.

For information on the PARALLEL directive, see Section 4.3, page 123. For information on the DO directive, see Section 4.4.1, page 126.

do_loop A DO loop.

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

Example. The following example shows how to parallelize a simple loop:

```
!$OMP PARALLEL DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END PARALLEL DO
```

In the preceding code, the loop iteration variable is private by default, so it is not necessary to declare it explicitly. The END PARALLEL DO directive is optional.

Note: Localized ALLOCATABLE or POINTER arrays are not supported on the DO, PARALLEL, or PARALLEL DO directives.

4.5.2 Declare Sections within a Parallel Region: PARALLEL SECTIONS and END PARALLEL SECTIONS Directives

The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel region that contains a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.

The format of this directive is as follows:

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]

[!$OMP SECTION ]

block

[!$OMP SECTION

block]

. . .

!$OMP END PARALLEL SECTIONS
```

clause — *clause* can be one or more of the clauses accepted by the PARALLEL directive or the SECTIONS directive. These clauses are as follows:

- PRIVATE(*var*[, *var*] ...)
- FIRSTPRIVATE(*var*[, *var*] ...)
- LASTPRIVATE(*var*[, *var*] ...)
- REDUCTION({ *operator*|*intrinsic*}:*var*[, *var*] ...)
- SHARED(*var*[, *var*] ...)
- DEFAULT(PRIVATE | SHARED | NONE)
- IF(*scalar_logical_expression*)
- COPYIN(*var*[, *var*] ...)

The IF clause is described in Section 4.3, page 123. The SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, COPYIN, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 4.7.2, page 142.

For more information on the PARALLEL directive, see Section 4.3, page 123. For more information on the SECTIONS directive, see Section 4.4.2, page 129.

block — Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

The last section ends at the END PARALLEL SECTIONS directive.

Example. In the following code fragment, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently. The first SECTION directive is optional. All the SECTION directives need to appear in the lexical extent of the PARALLEL SECTIONS/END PARALLEL SECTIONS construct.

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL XAXIS
!$OMP SECTION
      CALL YAXIS
!$OMP SECTION
      CALL ZAXIS
!$OMP END PARALLEL SECTIONS
```

4.6 Synchronization Constructs

The following sections describe the synchronization constructs:

- Section 4.6.1, page 135, describes the MASTER and END MASTER directives.
- Section 4.6.2, page 135, describes the CRITICAL and END CRITICAL directives.
- Section 4.6.3, page 137, describes the BARRIER directive.
- Section 4.6.4, page 137, describes the ATOMIC directive.
- Section 4.6.5, page 138, describes the FLUSH directive.
- Section 4.6.6, page 140, describes the ORDERED and END ORDERED directives.

4.6.1 Request Execution by the Master Thread: MASTER and END MASTER Directives

The code enclosed within MASTER and END MASTER directives is executed by the master thread.

These directives have the following format:

```
!$OMP MASTER

block

!$OMP END MASTER
```

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

The other threads in the team skip the enclosed section of code and continue execution. There is no implied barrier either on entry to or exit from the master section.

4.6.2 Request Execution by a Single Thread: CRITICAL and END CRITICAL Directives

The CRITICAL and END CRITICAL directives restrict access to the enclosed code to one thread at a time.

These directives have the following format:

```
!$OMP CRITICAL [(name)]

block

!$OMP END CRITICAL [(name)]
```

name Identifies the critical section.

If a *name* is specified on a CRITICAL directive, the same *name* must also be specified on the END CRITICAL directive. If no *name* appears on the CRITICAL directive, no *name* can appear on the END CRITICAL directive.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section with the same name. All unnamed CRITICAL directives map to the same name. Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is undefined.

Example. The following code fragment includes several CRITICAL directives. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical section. Because there are two independent queues in this example, each queue is protected by CRITICAL directives with different names, XAXIS and YAXIS, respectively.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

4.6.3 Synchronize All Threads in a Team: BARRIER Directive

The BARRIER directive synchronizes all the threads in a team. When it encounters a barrier, a thread waits until all other threads in that team have reached the same point.

This directive has the following format:

```
!$OMP BARRIER
```

4.6.4 Protect a Location from Multiple Updates: ATOMIC Directive

The ATOMIC directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

This directive has the following format:

```
!$OMP ATOMIC
```

This directive applies only to the immediately following statement, which must have one of the following forms:

```
x = x operator expr

x = expr operator x

x = intrinsic (x, expr)

x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type. All references to storage location *x* must have the same type and type parameters.
- *expr* is a scalar expression that does not reference *x*.
- *intrinsic* is one of MAX, MIN, IAND, IOR, or IEOR.
- *operator* is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV. .

Only the load and store of *x* are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location in parallel must be protected with the ATOMIC directive, except those that are known to be free of race conditions.

Example 1. The following code fragment uses the ATOMIC directive:

```
!$OMP ATOMIC
      X(INDEX(I)) = Y(INDEX(I)) + B
```

Example 2. The following code fragment avoids race conditions by protecting all simultaneous updates of the location, by multiple threads, with the ATOMIC directive:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
      DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)
!$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
      ENDDO
```

Note that the ATOMIC directive applies only to the Fortran 90 statement that immediately follows it. As a result, Y is not updated atomically in the preceding code.

4.6.5 Read and Write Variables to Memory: FLUSH Directive

The FLUSH directive identifies synchronization points at which thread-visible variables are written back to memory. This directive must appear at the precise point in the code at which the synchronization is required.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

This directive has the following format:

```
!$OMP FLUSH [(var[, var] ...)]
```

var Variables to be flushed.

An implicit FLUSH directive is assumed for the following directives:

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED

The directive is not implied if a NOWAIT clause is present.

Example. The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK()
!
!I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
!
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
!
!WAIT TILL NEIGHBOR IS DONE
!
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
      END DO
!$OMP END PARALLEL
```

4.6.6 Request Sequential Ordering: ORDERED and END ORDERED Directives

The code enclosed within ORDERED and END ORDERED directives is executed in the order in which it would be executed in a sequential execution of an enclosing parallel loop.

These directives have the following format:

```
!$OMP ORDERED

block

!$OMP END ORDERED
```

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. This DO directive must have the ORDERED clause specified. For more information on the DO directive, see Section 4.4.1, page 126. For information on directive binding, see Section 4.8, page 151.

Only one thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it is guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel. ORDERED sections that bind to different DO directives are independent of each other.

The following restrictions apply to the ORDERED directive:

- An ORDERED directive cannot bind to a DO directive that does not have the ORDERED clause specified.
- An iteration of a loop with a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

Example. Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
```

```
        CALL WORK(I)
      END DO

      SUBROUTINE WORK(K)
!$OMP ORDERED
      WRITE(*,*) K
!$OMP END ORDERED
      END
```

4.7 Data Environment Constructs

The following subsections present constructs for controlling the data environment during the execution of parallel constructs. Section 4.7.1, page 141, describes the THREADPRIVATE directive, which makes common blocks local to a thread. Section 4.7.2, page 142, describes directive clauses that affect the data environment.

4.7.1 Declare Common Blocks Private to a Thread: THREADPRIVATE Directive

The THREADPRIVATE directive makes named common blocks private to a thread but global within the thread. In other words, each thread executing a THREADPRIVATE directive receives its own private copy of the named common blocks, which are then available to it in any routine within the scope of an application.

This directive must appear in the declaration section of the routine after the declaration of the listed common blocks. Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads. During serial portions and MASTER sections of the program, accesses are to the master thread's copy of the common block.

On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified on the PARALLEL directive. When a common block that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use. For subsequent parallel regions, the data in the THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

For more information on dynamic threads, see the OMP_SET_DYNAMIC(3) library routine.

The format of this directive is as follows:

```
!$OMP THREADPRIVATE(/cb/[,/cb/]...)
```

cb The name of the common block to be made private to a thread. Only named common blocks can be made thread private.

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- You cannot use a THREADPRIVATE common block or its constituent variables in any clause other than a COPYIN clause. As a result, they are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

4.7.2 Data Scope Attribute Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the clauses in this section are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. Usually, if no data scope clauses are specified for a directive, the default scope for variables affected by the directive is SHARED. Exceptions to this are described in Section 4.7.3, page 149.

The following sections describe the data scope attribute clauses:

- Section 4.7.2.1, page 143, describes the PRIVATE clause.
- Section 4.7.2.2, page 144, describes the SHARED clause.
- Section 4.7.2.3, page 144, describes the DEFAULT clause.
- Section 4.7.2.4, page 145, describes the FIRSTPRIVATE clause.
- Section 4.7.2.5, page 145, describes the LASTPRIVATE clause.
- Section 4.7.2.6, page 146, describes the REDUCTION clause.
- Section 4.7.2.7, page 149, describes the COPYIN clause.

4.7.2.1 PRIVATE Clause

The PRIVATE clause declares variables to be private to each thread in a team.

This clause has the following format:

```
PRIVATE(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

The behavior of a variable declared in a PRIVATE clause is as follows:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the storage location of the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the construct and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

Example. The following example shows how to scope variables with the PRIVATE clause:

```
      INTEGER I,J
      I = 1
      J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
      I = 3
      J = J+ 2
!$OMP END PARALLEL
      PRINT *, I, J
```

In the preceding code, the values of I and J are undefined on exit from the parallel region.

4.7.2.2 SHARED Clause

The SHARED clause makes variables shared among all the threads in a team. All threads within a team access the same storage area for SHARED data.

This clause has the following format:

```
SHARED(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

4.7.2.3 DEFAULT Clause

The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE default scope attribute for all variables in the lexical extent of any parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause.

This clause has the following format:

```
DEFAULT(PRIVATE | SHARED| NONE)
```

The PRIVATE, SHARED, and NONE specifications have the following effects:

- Specifying DEFAULT(PRIVATE) makes all named objects in the lexical extent of the parallel region, including common block variables but excluding THREADPRIVATE variables, private to a thread as if each variable were listed explicitly in a PRIVATE clause.
- Specifying DEFAULT(SHARED) makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if each variable were listed explicitly in a SHARED clause. In the absence of an explicit DEFAULT clause, the default behavior is the same as if DEFAULT(SHARED) were specified.
- Specifying DEFAULT(NONE) declares that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION attribute of each variable used in the lexical extent of the parallel region must be specified.

Only one DEFAULT clause can be specified on a PARALLEL directive.

Variables can be exempted from a defined default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses. As a result, the following example is valid:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED(R) LASTPRIVATE(I)
```

4.7.2.4 FIRSTPRIVATE Clause

The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause.

This clause has the following format:

```
FIRSTPRIVATE(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

Variables specified are subject to PRIVATE clause semantics described in Section 4.7.2.1, page 143. In addition, private copies of the variables are initialized from the original object existing before the construct.

4.7.2.5 LASTPRIVATE Clause

The LASTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause.

When the LASTPRIVATE clause appears on a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct. Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

This clause has the following format:

```
LASTPRIVATE(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

Each *var* is subject to the PRIVATE clause semantics described in Section 4.7.2.1, page 143.

Example. Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      ENDDO
!$OMP END PARALLEL
      CALL REVERSE(I)
```

In the preceding code fragment, the value of I at the end of the parallel region will equal N+1, as in the sequential case.

4.7.2.6 REDUCTION Clause

This clause performs a reduction on the variables specified, with the operator or the intrinsic specified.

This clause has the following format:

```
REDUCTION({operator|intrinsic}:var[, var] ...)
```

operator Specify one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV.

intrinsic Specify one of the following: MAX, MIN, IAND, IOR, or IEOR.

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. Each *var* must be a named scalar variable of intrinsic type.

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each *var* is created for each thread as if the PRIVATE clause had been used. The private

copy is initialized according to the operator. For more information, see Table 6, page 148.

If a named common block is specified, its name must appear between slashes.

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value (the partial results of a subtraction reduction are added to form the final value).

The value of the shared variable becomes undefined when the first thread reaches the containing clause, and it remains so until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct; however, if the REDUCTION clause is used on a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed to ensure that all the threads have completed the REDUCTION clause.

The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in reduction statements with one of the following forms:

```
x = x operator expr

x = expr operator x   (except for subtraction)

x = intrinsic (x,expr)

x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a MAX reduction might be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. You must ensure that the operator specified in the REDUCTION clause matches the reduction operation.

The following table lists the operators and intrinsics that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 6. Initialization values

Operator / Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a REDUCTION clause for that directive.

Example 1. The following directive line shows use of the REDUCTION clause:

```
!$OMP DO REDUCTION(+: A, Y) REDUCTION(.OR.: AM)
```

Example 2. The following code fragment shows how to use the REDUCTION clause:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      ENDDO
!$OMP END PARALLEL DO
```

4.7.2.7 COPYIN Clause

The COPYIN clause applies only to common blocks that are declared THREADPRIVATE. A COPYIN clause on a parallel region specifies that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region.

This clause has the following format:

```
COPYIN(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

It is not necessary to specify a whole common block to be copied in.

Example. In the following example, the common blocks BLK1 and FIELDS are specified as thread private, but only one of the variables in common block FIELDS is specified to be copied in:

```
      COMMON /BLK1/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/,ZFIELD)
```

4.7.3 Data Environment Rules

The following rules and restrictions apply with respect to data scope:

1. Sequential DO loop control variables in the lexical extent of a PARALLEL region that would otherwise be SHARED based on default rules are automatically made private on the PARALLEL directive. Sequential DO loop control variables with no enclosing PARALLEL region are not classified automatically. You must guarantee that these indexes are private if the containing procedures are called from a PARALLEL region.

   All implied DO loop control variables are automatically made private at the enclosing implied DO construct.

2. Variables that are made private in a parallel region cannot be made private again on an enclosed work-sharing directive. As a result, variables that appear in the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION

clauses on a work-sharing directive have shared scope in the enclosing parallel region.

3. A variable that appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause must be definable.

4. Assumed-size and assumed-shape arrays cannot be specified as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE. Array dummy arguments that are explicitly shaped (including variably dimensioned) can be declared in any scoping clause.

5. Fortran pointers and allocatable arrays can be declared as PRIVATE or SHARED but not as FIRSTPRIVATE or LASTPRIVATE.

   Within a parallel region, the initial status of a private pointer is undefined. Private pointers that become allocated during the execution of a parallel region should be explicitly deallocated by the program prior to the end of the parallel region to avoid memory leaks.

   The association status of a SHARED pointer becomes undefined upon entry to and on exit from the parallel construct if it is associated with a target or a subobject of a target that is PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION inside the parallel construct. An allocatable array declared PRIVATE has an allocation status of *not currently allocated* on entry to and on exit from the construct.

6. PRIVATE or SHARED attributes can be declared for a Cray pointer but not for the pointee. The scope attribute for the pointee is determined at the point of pointer definition. You cannot declare a scope attribute for a pointee. Cray pointers cannot be specified in FIRSTPRIVATE or LASTPRIVATE clauses.

7. Scope clauses apply only to variables in the static extent of the directive on which the clause appears, with the exception of variables passed as actual arguments. Local variables in called routines that do not have the SAVE attribute are PRIVATE. Common blocks and modules in called routines in the dynamic extent of a parallel region always have an implicit SHARED attribute, unless they are THREADPRIVATE common blocks.

8. When a named common block is declared as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE, none of its constituent elements may be declared in another scope attribute. When individual members of a common block are privatized, the storage of the specified variables is no longer associated with the storage of the common block itself.

9. Variables that are not allowed in the PRIVATE and SHARED clauses are not affected by DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses, respectively.
10. Clauses can be repeated as needed, but each variable can appear explicitly in only one clause per directive, with the following exceptions:
    - A variable can be specified as both FIRSTPRIVATE and LASTPRIVATE.
    - Variables affected by the DEFAULT clause can be listed explicitly in a clause to override the default specification.

4.8 Directive Binding

Some directives are *bound* to other directives. A binding specifies the way in which one directive is related to another. For instance, a directive is bound to a second directive if it can appear in the dynamic extent of that second directive. The following rules apply with respect to the dynamic binding of directives:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

Example 1. The directive binding rules call for a BARRIER directive to bind to the closest enclosing PARALLEL directive.

In the following example, the call from MAIN to SUB2 is valid because the BARRIER (in SUB3) binds to the PARALLEL region in SUB2. The call from MAIN to SUB1 is valid because the BARRIER binds to the PARALLEL region in subroutine SUB2.

```
PROGRAM MAIN
CALL SUB1(2)
CALL SUB2(2)
END
```

```
      SUBROUTINE SUB1(N)
!$OMP PARALLEL PRIVATE(I) SHARED(N)
!$OMP DO
      DO I = 1, N
      CALL SUB2(I)
      END DO
!$OMP END PARALLEL
      END

      SUBROUTINE SUB2(K)
!$OMP PARALLEL SHARED(K)
      CALL SUB3(K)
!$OMP END PARALLEL
      END

      SUBROUTINE SUB3(N)
      CALL WORK(N)
!$OMP BARRIER
      CALL WORK(N)
      END
```

Example 2. The following program shows inner and outer DO directives that bind to different PARALLEL regions:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP PARALLEL SHARED(I,N)
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
!$OMP END PARALLEL
      END DO
!$OMP END PARALLEL
```

A following variation of the preceding example also shows correct binding:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL
```

```
      SUBROUTINE SOME_WORK(I,N)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
      END DO
!$OMP END PARALLEL
      RETURN
      END
```

4.9 Directive Nesting

The following rules apply to the dynamic nesting of directives:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive cannot be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Example 1. The following example is incorrect because the inner and outer DO directives are nested and bind to the same PARALLEL directive:

```
      PROGRAM WRONG1
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
```

```
      DO I = 1, N
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
      END DO
!$OMP END PARALLEL
      END
```

The following dynamically nested version of the preceding code is also incorrect:

```
      PROGRAM WRONG2
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL

      SUBROUTINE SOME_WORK(I,N)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
      END DO
      RETURN
      END
```

Example 2. The following example is incorrect because the DO and SINGLE directives are nested, and they bind to the same PARALLEL region:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP SINGLE
      CALL WORK(I)
!$OMP END SINGLE
      END DO
!$OMP END PARALLEL
      END
```

Example 3. The following example is incorrect because a BARRIER directive inside a SINGLE or a DO directive can result in deadlock:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL WORK(I)
!$OMP BARRIER
        CALL MORE_WORK(I)
      END DO
!$OMP END PARALLEL
      END
```

Example 4. The following example is incorrect because the BARRIER results in deadlock due to the fact that only one thread at a time can enter the critical section:

```
      PROGRAM WRONG4
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP CRITICAL
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END CRITICAL
!$OMP END PARALLEL
      END
```

Example 5. The following example is incorrect because the BARRIER results in deadlock due to the fact that only one thread executes the SINGLE section:

```
      PROGRAM WRONG5
!$OMP PARALLEL DEFAULT(SHARED)
      CALL SETUP(N)
!$OMP SINGLE
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END SINGLE
      CALL FINISH(N)
!$OMP END PARALLEL
      END
```

4.10 Analyzing Data Dependencies for Multiprocessing

The essential condition required to parallelize a loop correctly is that each iteration of the loop must be independent of all other iterations. If a loop meets this condition, then the order in which the iterations of the loop execute is not important. They can be executed backward or at the same time, and the answer is still the same. This property is captured by the notion of *data independence*.

For a loop to be data independent, no iterations of the loop can write a value into a memory location that is read or written by any other iteration of that loop. It is all right if the same iteration reads and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location as long as none of them write to it.

In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left-hand side of assignment statements. If a variable is neither modified nor passed to a function or subroutine, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: `SHARED`, `PRIVATE`, `LASTPRIVATE`, and `REDUCTION`. If a variable is declared as `SHARED`, all iterations of the loop use the same copy. If a variable is declared as `PRIVATE`, each iteration is given its own uninitialized copy. A variable is declared `SHARED` if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be `PRIVATE` if its value does not depend on any other iteration and if its value is used only within a single iteration. The `PRIVATE` variable is essentially temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the last value of a variable computed on the last iteration is used outside the loop (but would otherwise qualify as a `PRIVATE` variable), the loop can be multiprocessed by declaring the variable to be `LASTPRIVATE`.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to determine if it fulfills the criteria for `PRIVATE`, `LASTPRIVATE`, `SHARED`, or `REDUCTION`. If all of the uses conform, the loop can be parallelized. If not, the loop cannot be parallelized as written, but can possibly be rewritten into an equivalent parallel form.

When `-O task3` is specified in the `f90`(1) command line, the compiler analyzes loops for data dependence. If the compiler determines that a loop is data-independent, it automatically inserts the required compiler directives. You

can specify the `-O msgs` or `-O negmsgs` options to obtain messages from the compiler that indicate why certain loops were candidates for tasking and why certain others were not. For more information on these compiler options, see Section 2.2.20, page 20.

4.10.1 Dependency Analysis Examples

This section contains examples that show dependency analysis.

Example 1. Simple independence. In this example, each iteration writes to a different location in A, and none of the variables appearing on the right-hand side are ever written to; they are only read from. This loop can be correctly run in parallel. All the variables are SHARED except for I, which is either PRIVATE or LASTPRIVATE, depending on whether the last value of I is used later in the code.

```
  DO I = 1,N
  A(I) = X + B(I)*C(I)
END DO
```

Example 2. Data dependence. The following code fragment contains A(I) on the left-hand side and A(I-1) on the right. This means that one iteration of the loop writes to a location in A and the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

```
   DO I = 2,N
   A(I) = B(I) - A(I-1)
END DO
```

Example 3. Stride not 1. This example is similar to the previous example. The difference is that the stride of the DO loop is now 2 rather than 1. A(I) now references every other element of A, and A(I-1) references exactly those elements of A that are not referenced by A(I). None of the data locations on the right-hand side is ever the same as any of the data locations written to on the left-hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
DO I = 2,N,2
   A(I) = B(I) - A(I-1)
END DO
```

Example 4. Local variable. In the following loop, each iteration of the loop reads and writes the variable X. However, no loop iteration ever needs the value of X from any other iteration. X is used as a temporary variable; its value does not survive from one iteration to the next.

This loop can be parallelized by declaring X to be a PRIVATE variable within the loop. Note that B(I) is both read and written by the loop. This is not a problem because each iteration has a different value for I, so each iteration uses a different B(I). The same B(I) is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
DO I = 1, N
     X = A(I)*A(I) + B(I)
     B(I) = X + B(I)*X
END DO
```

Example 5. Function call. The value of X in any iteration of the following loop is independent of the value of X in any other iteration, so X can be made a PRIVATE variable. The loop can be run in parallel. Arrays A, B, C, and D can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
     DO I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
      END DO
```

This loop invokes an intrinsic function, SQRT. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, verify that the parallel invocations of the routine do not interfere with one another. In particular, SQRT returns a value that depends only on its input argument, does not modify global data, and does not use static storage (it has no side effects).

The Fortran 90 intrinsic functions have no side effects. The intrinsic functions can be used safely within a parallel loop. The intrinsic subroutines, however, can have side effects. Most Fortran library functions cannot be included in a parallel loop. In particular, rand is not safe for multiprocessing. For user-written routines, it is your responsibility to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the -e v option on the f90(1) command line when compiling routines called within a parallel loop.

Example 6. Rewritable data dependence. Here, the value of INDX survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making INDX a PRIVATE variable does not work; you need the value of INDX computed in the previous iteration. It is possible to rewrite this loop to make it parallel. See Section 4.10.2, page 160, for an example.

```
INDX = 0
DO I = 1, N
     INDX = INDX + I
     A(I) = B(I) + C(INDX)
END DO
```

Example 7. Exit branch. The following loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The compiler cannot parallelize loops containing exit branches.

```
       DO I = 1, N
          IF (A(I) .LT. EPSILON) EXIT
          A(I) = A(I) * B(I)
       END DO
```

Example 8. Complicated independence. Initially, it appears that the following loop cannot be run in parallel because it uses both W(I) and W(I-K). However, because the value of I varies between K+1 and 2*K, then I-K goes from 1 to K. This means that the W(I-K) term varies from W(1) to W(K), while the W(I) term varies from W(K+1) to W(2*K). Therefore, W(I-K) in any iteration of the loop is never the same memory location as W(I) in any other iterations. Because there is no data overlap, there are no data dependencies. This loop can be run in parallel. Elements W, B, and K can be declared SHARED, but variable I should be declared PRIVATE or LASTPRIVATE.

```
DO I = K+1, 2*K
     W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

The preceding code illustrates a general rule: the more complex the expression used to index an array, the harder it is to analyze. If the arrays in a loop are indexed only by the loop index variable, the analysis is usually straightforward.

Example 9. Inconsequential data dependence. The data dependence in the following loop is present because it is possible that at some point that I will be the same as INDEX, so there will be a data location that is being read and written by different iterations of the loop. In this special case, you can simply ignore it. You know that when I and INDEX are equal, the value written into A(I) is exactly the same as the value that is already there. The fact that some

iterations of the loop read the value before it is written and some after it is written is not important because they all get the same value. Therefore, this loop can be parallelized. Array A can be declared SHARED, but variable I should be declared PRIVATE or LASTPRIVATE.

```
INDEX = SELECT(N)
DO I = 1, N
     A(I) = A(INDEX)
END DO
```

Example 10. Local array. In the following code fragment, each iteration of the loop uses the same locations in array D. However, closer inspection reveals that array D is being used as a temporary. This can be multiprocessed by declaring D to be PRIVATE. The Fortran compiler allows arrays (even multidimensional arrays) to be PRIVATE variables, with the following restrictions: the size of the array must be either a constant or an expression; the dimension bounds must be specified; the PRIVATE array cannot have been declared using a variable or the asterisk (*) syntax; and assumed-shape, deferred-shape, and pointer arrays are not permitted.

```
DO I = 1, N
     D(1) = A(I,1) - A(J,1)
     D(2) = A(I,2) - A(J,2)
     D(3) = A(I,3) - A(J,3)
     TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 + D(3)**2)
END DO
```

The preceding loop can be parallelized. Arrays TOTAL_DISTANCE and A can be declared SHARED, and array D and variable I can be declared PRIVATE or LASTPRIVATE.

4.10.2 Rewriting Data Dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. You must first locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

After you identify data dependencies, you can use various techniques to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to find a new parallel method of solving the problem. The following examples

show how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example 1. Loop-carried value. The following code segment is the same as the rewritable data dependence example in the previous section. INDX has its value carried from iteration to iteration. However, you can compute the appropriate value for INDX without making reference to any previous value.

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

For example, consider the following code:

```
!$OMP PARALLEL DO PRIVATE (I, INDX)
    DO I = 1, N
       INDX = (I*(I+1))/2
       A(I) = B(I) + C(INDX)
    END DO
```

In this loop, the value of INDX is computed without using any values computed on any other iteration. INDX can correctly be made a PRIVATE variable, and the loop can now be multiprocessed.

Example 2. Indirect indexing. Consider the following code:

```
     DO I = 1, N
        IX = INDEXX(I)
        IY = INDEXY(I)
        XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
        YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
        IXX = IXOFFSET(IX)
        IYY = IYOFFSET(IY)
        TOTAL(IXX, IYY) = TOTAL(IXX, IYY) + EPSILON
      END DO
```

It is the final statement that causes problems. The indexes IXX and IYY are computed in a complex way and depend on the values from the IXOFFSET and IYOFFSET arrays. It is not known if TOTAL(IXX,IYY) in one iteration of the loop will always be different from TOTAL(IXX,IYY) in every other iteration of the loop.

You can pull the statement out into its own separate loop by expanding IXX and IYY into arrays to hold intermediate values, as follows:

```
!$OMP PARALLEL DO PRIVATE(IX, IY, I)
      DO I  = 1, N
         IX = INDEXX(I)
         IY = INDEXY(I)
         XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
         YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
         IXX(I) = IXOFFSET(IX)
         IYY(I) = IYOFFSET(IY)
      END DO
      DO I = 1, N
         TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
      END DO
```

Here, IXX and IYY have been turned into arrays to hold all the values computed by the first loop. The first loop (containing most of the work) can now be run in parallel. Only the second loop must still be run serially. This is true if IXOFFSET or IYOFFSET are permutation vectors.

If you were certain that the value for IXX was always different in every iteration of the loop, then the original loop could be run in parallel. It could also be run in parallel if IYY was always different. If IXX (or IYY) is always different in every iteration, then TOTAL(IXX,IYY) is never the same location in any iteration of the loop, and so there is no data conflict.

This sort of knowledge is program-specific and should always be used with great care. It may be true for a particular data set, but to run the original code in parallel as it stands, you need to be sure it will always be true for all possible input data sets.

Example 3. Recurrence. The following example shows a *recurrence*, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

```
DO I = 1,N
     X(I) = X(I-1) + Y(I)
END DO
```

Example 4. Sum reduction. The following example shows an operation known as a *reduction*. Reductions occur when an array of values is combined and reduced into a single value.

```
      SUM  = 0.0
      DO I = 1,N
           SUM = SUM + A(I)
      END DO
```

This example is a sum reduction because the combining operation is addition. Here, the value of SUM is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of A(I), you can rewrite the loop to accumulate multiple, independent subtotals and do much of the work in parallel, as follows:

```
     NUM_THREADS = OMP_GET_NUM_THREADS()
!
!  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
!
      IPIECE_SIZE = (N + (NUM_THREADS-1)) / NUM_THREADS
      DO K = 1, NUM_THREADS
        PARTIAL_SUM(K) = 0.0
!
!  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
!  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
!  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
!  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
!  HENCE THE "MIN" EXPRESSION.
!
      DO I = K*IPIECE_SIZE - IPIECE_SIZE + 1, MIN(K*IPIECE_SIZE,N)
           PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
        END DO
      END DO
!
!  NOW ADD UP THE PARTIAL SUMS
      SUM = 0.0
      DO I = 1, NUM_THREADS
         SUM = SUM + PARTIAL_SUM(I)
      END DO
```

The outer loop K can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

Because this is an important and common transformation, automatic support is provided by the REDUCTION clause:

```
      SUM = 0.0
!$OMP PARALLEL DO PRIVATE (I), REDUCTION (+:SUM)
      DO 10 I = 1, N
         SUM = SUM + A(I)
10 CONTINUE
```

The previous code has essentially the same meaning as the much longer and more confusing code above. Adding an extra dimension to an array to permit parallel computation and then combining the partial results is an important technique for trying to break data dependencies. This technique is often useful.

Reduction transformations such as this do not produce the same results as the original code. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round-off errors accumulate slightly differently. It is probable that the final answer will be slightly different from the original loop. Both answers are equally correct. The difference is usually irrelevant, but sometimes it can be significant. If the difference is significant, neither answer is really trustworthy.

This example is a `sum` reduction because the operator is plus (+). The Fortran compiler supports the following types of reduction operations:

- sum: `p = p+a(i)`
- product: `p = p*a(i)`
- min: `m = MIN(m,a(i))`
- max: `m = MAX(m,a(i))`

For example,

```
!$OMP PARALLEL DO PRIVATE (I), REDUCTION(+:BG_SUM),
!$OMP+REDUCTION(*:BG_PROD), REDUCTION(MIN:BG_MIN), REDUCTION(MAX:BG_MAX)
          DO I = 1,N
             BG_SUM  = BG_SUM + A(I)
             BG_PROD = BG_PROD * A(I)
             BG_MIN  = MIN(BG_MIN, A(I))
             BG_MAX  = MAX(BG_MAX, A(I))
          END DO
```

The following is another example of a reduction transformation:

```
        DO I = 1, N
           TOTAL = 0.0
           DO J = 1, M
```

```
            TOTAL = TOTAL + A(J)
         END DO
         B(I) = C(I) * TOTAL
      END DO
```

Initially, it might look as if the inner loop should be parallelized with a REDUCTION clause. However, consider the outer I loop. Although TOTAL cannot be made a PRIVATE variable in the inner loop, it fulfills the criteria for a PRIVATE variable in the outer loop: the value of TOTAL in each iteration of the outer loop does not depend on the value of TOTAL in any other iteration of the outer loop. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer I loop, making TOTAL and J local variables.

4.11 Work Quantum

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.

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 uses 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
```

Source Preprocessing [5]

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` (for a file in free source form). To specify preprocessing of source files with other extensions, including lowercase ones, use the `-e P` or `-e Z` options described in Section 5.4, page 176.

5.1 General Rules

You can alter the source code through source preprocessing directives. These directives are fully explained in Section 5.2, page 168. The directives must be used according to the following rules:

- Do not use source preprocessor (#) directives within multiline compiler directives (`CDIR$`, `!DIR$`, `CMIC$`, `!MIC$`, `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 whitespace 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 90 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 31 characters.
- Source preprocessing identifier names are case sensitive.
- Numeric literal constants must be integer literal constants or real literal constants, as defined for Fortran 90.
- 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.

5.2 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.

5.2.1 `#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 90 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 `f90`(1) 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 `f90`(1) command line and then search the standard (default) sequence.

The Fortran 90 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 90 `INCLUDE` lines. For information on the source preprocessing command line options, see Section 5.4, page 176.

5.2.2 `#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 on the `#undef` directive, see Section 5.2.3, page 171.

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
```

5.2.3 #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.

5.2.4 # (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.

5.2.5 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 90 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 90 determination of which block of an `IF` construct should be executed.

5.2.5.1 `#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 90 operators. The *expression* is evaluated according to C language rules, not Fortran 90 expression evaluation rules.

Note that unlike the Fortran 90 `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).

5.2.5.2 #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 f90 -D command line option. For more information on the -D option, see Section 5.4, page 176. 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*)

5.2.5.3 #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*)

5.2.5.4 #elif Directive

The #elif directive serves the same purpose in an if-group as does the ELSE IF statement of a Fortran 90 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.

5.2.5.5 `#else` Directive

The `#else` directive serves the same purpose in an if-group as does the `ELSE` statement of a Fortran 90 `IF` construct. This directive has the following format:

```
#else
```

5.2.5.6 `#endif` Directive

The `#endif` directive serves the same purpose in an if-group as does the `END IF` statement of a Fortran 90 `IF` construct. This directive has the following format:

```
#endif
```

5.3 Predefined Macros

CF90 source preprocessing supports a number of Cray Research predefined macros. They are divided into groups as follows:

- Macros that are based on the host machine
- Macros that are based on Cray system targets

The following predefined macros are based on the host system (the system upon which the compilation is being done):

Macro	Description
`unix`, `___unix`	Always defined. (The leading characters in the second form consist of 2 consecutive underscores.)
`_UNICOS`	Defined only when the compilation is being done on a UNICOS system. Its value is the major release level of UNICOS installed on the system.

The following predefined macros are based on Cray systems as targets:

Macro	Description
cray, CRAY, _CRAY	These macros are defined for UNICOS and UNICOS/mk systems as targets.
CRAY1, _CRAY1	These macros are defined for UNICOS systems as targets.
YMP	Defined if the addr32 component of the TARGET environment variable is set.
_CRAYC90	Defined if the target machine is a CRAY C90 system.
_CRAYT90	Defined if the target machine is a CRAY T90 system.
_CRAYT3E, _CRAYMPP	Defined if the target machine is a CRAY T3E system. **Note:** If you were using the _CRAYMPP macro, in releases prior to CF90 3.0, to differentiate the CRAY T3D system from other systems, you need to change that macro to either _CRAYT3D (for CRAY T3D systems) or _CRAYT3E (for CRAY T3E systems).
_CRAYIEEE	Defined if the target machine uses IEEE floating-point format real values. Undefined if the target machine uses Cray floating-point format.
_MEMSIZE	The value is obtained from the memsize component of the TARGET environment variable.
_MAXVL	Defined as the hardware vector register length (64 or 128). Defined only when the target machine has vector registers.
_ADDR32	Defined for UNICOS systems as targets. The target system must have 32–bit address registers.

_ADDR64	Defined for UNICOS and UNICOS/mk systems as targets. The target system must have 64–bit address registers.

5.4 Command Line Options

Several f90(1) command line options affect source preprocessing. They are as follows:

- The -D *identifier*[=*value*] [, *identifier*[=*value*]] ... option, which defines variables used for source preprocessing. For more information on this option, see Section 2.2.7, page 14.
- The -e P option, which performs source preprocessing on *file*.f[90] or *file*.F[90] but does not compile. The -e P option produces *file*.i. For more information on this option, see Section 2.2.6, page 9.
- The -e Z option, which performs source preprocessing and compilation on *file*.f[90] or *file*.F[90]. The -e Z option produces *file*.i. For more information on this option, see Section 2.2.6, page 9.
- The -F option, which enables macro expansion throughout the source file. For more information on this option, see Section 2.2.9, page 15.
- The -U *identifier* [, *identifier*] ... option, which undefines variables used for source preprocessing. For more information on this option, see Section 2.2.28, page 54.
- The -Wp"*srcpp_opt*" option, which passes *srcpp_opt* to the source preprocessor. For more information on this option, see Section 2.2.33, page 55.

The -D *identifier*[=*value*] [, *identifier*[=*value*]] ..., -F, -U *identifier* [, *identifier*] ..., and -Wp"*srcpp_opt*" options are ignored unless one of the following is true:

- The Fortran input source file is specified as either *file*.F or *file*.F90.
- The -e P or -e Z options have been specified.

Autotasking Directives (UNICOS systems only) (Outmoded) [A]

If your system includes multiple central processing units (CPUs), your program may be able to make use of *multitasking*, or running simultaneously on more than one CPU. This technology speeds up program execution by decreasing elapsed time. You can determine the number of CPUs on your system by entering the (1) command.

Note: The directives in this appendix are outmoded. They are supported for older codes that require this functionality. Cray Research ecourages you to write new codes using the OpenMP Fortran API directives described in Chapter 4, page 119.

The Autotasking directives are not available on UNICOS/mk systems.

You can mix Autotasking directives and OpenMP Fortran API directives in the same compilation unit.

The CF90 compiler automatically recognizes many parallel coding constructs, and it compiles them for multitasking without requiring additional user input; this capability is called *Autotasking*.

To benefit from the Autotasking software, your program must be suitable for dividing into separate tasks. If you are uncertain whether you can make use of this capability, you can compare execution times of different Autotasked versions of your program by using ATExpert to analyze performance gains. To use ATExpert, specify `-e X` on the `f90`(1) command line.

Autotasking directives let you specify the level of parallelism desired. You can start and end parallel processing at any number of suitable points within a subprogram. These directives are useful when the compiler fails to recognize parallelism that you know exists. This can occur, for example, when you have subroutine calls that can be executed in parallel.

This section provides an overview of the Autotasking directives recognized by the CF90 compiler. For more information on the tasking capabilities available through the command line, see Section 2.2.20, page 20.

Caution: The ability to use Autotasking directives in a subprogram that host associates a variable can result in undefined behavior. This note applies only to Autotasking directives; it does not apply to parallelism detected by the compiler. For more information on host association, see the *Fortran Language Reference Manual, Volume 1*, publication SR–3902.

Autotasking directives control the way the CF90 compiler multitasks your program. You can insert tasking directive lines directly into your source code. The CF90 compiler supports the following Autotasking directives:

- `CASE`, `ENDCASE`
- `CNCALL`
- `DOALL`
- `DOPARALLEL`, `ENDDO`
- `GUARD`, `ENDGUARD`
- `MAXCPUS`
- `NUMCPUS`
- `PARALLEL`, `ENDPARALLEL`
- `PERMUTATION`
- `WAIT`, `SEND`

The following sections describe the use and syntax of the Autotasking directives.

A.1 Using Directives

The following sections describe how to use directives and the effects they have on programs.

A.1.1 Directive Lines

An Autotasking *directive line* begins with the characters `CMIC$` or `!MIC$`. 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 `CMIC$` or `!MIC$` 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 !MIC$, followed by a space, and then one or more directives. If the position following the !MIC$ contains a character other than a blank, tab, or newline character, the line is assumed to be a continuation line. The !MIC$ 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:

```
!MIC$ CN
!MIC$*CALL
```

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 CF90 directive as a comment:

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

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

A.1.2 Range and Placement of Directives

The range and placement of directives is as follows:

- The MAXCPUS directive can appear anywhere in your source code. All other directives must appear within a program unit.
- The NUMCPUS directive toggles a compiler feature at the point at which the directive appears in the code. This directive is in effect until it 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.

- The ENDDO directive must appear after the loop body of a DOPARALLEL loop, if it appears. The corresponding DOPARALLEL directive must be present.
- The following directives apply only to the next loop encountered lexically:
  - CNCALL
  - DOALL
  - DOPARALLEL
  - PERMUTATION
  - PREFERTASK
- The following Autotasking directives must appear as pairs within a program unit:
  - CASE, ENDCASE
  - GUARD, ENDGUARD
  - PARALLEL, ENDPARALLEL

A.1.3 Interaction of Directives with the -x Command Line Option

The -x option on the f90(1) 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 mic, all directives preceded by !MIC$ or CMIC$ are ignored.

For more information on the -x option, see Section 2.2.35, page 56.

A.1.4 Command Line Options and Directives

Some features activated by directives can also be specified on the f90(1) 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, 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 f90(1), the associated !*prefix*$ directives are ignored. The following list shows

CF90 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, and tasking directives are ignored.
- Specifying the `-O task0` option disables tasking and causes the compiler to ignore tasking directives and Autotasking directives.

The following sections describe directive syntax and the effects of the Autotasking directives on CF90 programs.

A.1.5 Migrating to OpenMP Fortran API Directives.

The OpenMP Fortran API is a standard that consists of a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran programs. OpenMP allows you to create and manage parallel programs while ensuring portablilty.

The following table compares the Cray Autotasking directives to the OpenMP Fortran API constructs with the same functionality.

Autotasking directive	OpenMP Construct
CASE	SECTIONS
ENDCASE	END SECTIONS
DOALL	PARALLEL DO
DOPARALLEL	DO
ENDDO	END DO
GUARD	CRITICAL
ENDGUARD	END CRITICAL
PARALLEL	PARALLEL
ENDPARALLEL	END PARALLEL

Note: Prior to the CF90 3.1 release, the `CNCALL`, `MAXCPUS`, `NUMCPUS`, and `PERMUTATION` directives were supported as `!MIC$` Autotasking directives. These directives are now accepted with both the `!MIC$` and a `!DIR$` prefix. This flexibility allows you to disable the Autotasking directives (by specifying `-x mic`) but still allow the compiler to recognize the OpenMP directives and the `CNCALL`, `MAXCPUS`, `NUMCPUS`, and `PERMUTATION` directives.

A.2 Concurrent Blocks: CASE and ENDCASE

The !MIC$ CASE directive serves as a separator between adjacent code blocks that can be executed concurrently. It marks the beginning of a control structure and signals that the code following it will be executed on a single processor.

!MIC$ ENDCASE serves as the terminator for a group of one or more parallel CASE directives. All work within the control structure must complete before execution continues with the code below the ENDCASE. The compiler does not automatically generate CASE directives.

The formats for these directives are as follows:

```
!MIC$ CASE

!MIC$ ENDCASE
```

Example 1. The following example shows how the CASE directive is most often used:

```
!MIC$ PARALLEL MAXCPUS(3)
!MIC$   CASE
        CALL ABC
!MIC$   CASE
        CALL DEF
!MIC$   CASE
        CALL GHI
!MIC$   ENDCASE
!MIC$ ENDPARALLEL
```

In the preceding code, the MAXCPUS parameter on the PARALLEL directive indicates that only three processors are necessary for this parallel region because it contains only three control structures. The CASE directives indicate that the subroutine calls in this example are concurrently executable. The code within each control structure is executed on a single processor. The work in the subroutine calls completes before execution continues with the code following the ENDCASE.

Example 2. A single CASE/ENDCASE directive pair can also be used within a parallel region to allow only one processor to execute a code block, as follows:

```
!MIC$ PARALLEL
!MIC$ CASE
      CALL XYZ
```

```
!MIC$ ENDCASE
       :
!MIC$ DOPARALLEL
      DO I = 1, IMAX
       :
      END DO
!MIC$ ENDPARALLEL
```

In the preceding code, only one processor calls XYZ, and then all available processors execute the code following the ENDCASE. The MAXCPUS parameter is not used in this example because all available processors are required for the code following the ENDCASE directive.

A.3 Declare Lack of Side Effects: CNCALL

The format for this directive is as follows:

```
!MIC$ CNCALL
```

This directive is also implemented with a !DIR$ prefix. For detailed information on this directive, see Section 3.2.1, page 78.

A.4 Mark Parallel Loop: DOALL

The !MIC$ DOALL directive indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a DOALL loop, (that is, the DOALL initiates a parallel region that contains only a DO loop with independent iterations). The loop index variable for a DOALL must be specified as a PRIVATE variable.

Every variable in a parallel region must be declared as PRIVATE or SHARED unless the AUTOSCOPE parameter appears on the directive.

The format of this directive is as follows:

```
!MIC$ DOALL parameter [ [,] parameter ] ... [ [,] work_distribution ]
```

parameter Table 7, page 184, describes parameters for the DOALL directive. More than one parameter can

appear on the directive, but they must be separated by commas or blanks.

work_distribution — Parameters that specify the work distribution policy for iterations of the parallel DO loop. Only one can be used for a given DO loop.

By default, iterations are distributed one at a time (SINGLE). Table 8, page 186, describes the work distribution parameters.

Note that the specifications for *parameter* do not have to precede the specifications for *work_distribution*. The *parameter* and the *work_distribution* items can be in any order, and they can be intermixed. Table 7, page 184, and Table 8, page 186, describe the *parameter* and *work_distribution* arguments.

Table 7. Autotasking directive *parameter*

parameter	Description
AUTOSCOPE	Specifies that all unscoped variables that have not been explicitly scoped with a PRIVATE or SHARED declaration are scoped according to the default rules for scoping variables. For more information on the default rules for scoping variables, see Section A.12, page 195.
IF(*expr*)	Performs a run-time test to choose between uniprocessing and multiprocessing. When not specified, multiprocessing is chosen if the loop is not in a routine that was called from within a parallel region. The logical expression (*expr*) determines (at run time) whether multiprocessing will occur. When *expr* is true, multiprocessing is enabled.

MAXCPUS(*n*)	Specifies the maximum number of CPUs that the parallel region can use effectively. Does not ensure that *n* processors will be assigned. This is the optimal maximum. The *n* argument must be of type integer. Argument *n* can be a constant, a variable, or an expression. Both of the following are valid specifications:
	`MAXCPUS (2)`
	`MAXCPUS (NUM)`
	For information on the MAXCPUS directive, see Section A.7, page 189.

PRIVATE(*var*[,*var*] ...)	

parameter	Description
	Specifies that the variables listed will have *private* scope; that is, each task (original or helper) will have its own private copy of these variables. The PRIVATE clause identifies those variables that are not shared between parallel processes. One variable cannot be declared both PRIVATE and SHARED. The loop control variable of the DOALL loop cannot be specified as SHARED; it must be specified as PRIVATE. Variables cannot be subobjects (that is, array elements or components of derived types).
SAVELAST	Specifies that the values of private variables, from the final iteration of a DOALL directive, will continue in the original task after execution of the iterations of the DOALL. By default, private variables are not guaranteed to retain the last iteration values. SAVELAST can be used only with DOALL, and if the full iteration set is not completed (for example, if the loop is exited early), the values of private variables are indeterminate.
SHARED(*var*[,*var*] ...)	Specifies that the variables listed will have *shared* scope; that is, they are accessible to both the original task and all helper tasks. The SHARED clause identifies those variables that are shared between parallel processes. One variable cannot be declared both PRIVATE and SHARED. The loop control variable of the DOALL loop cannot be specified as SHARED; it must be specified as PRIVATE. Variables cannot be subobjects (that is, array elements or components of derived types).

Table 8. Autotasking directive *work_distribution*

work_ distribution	Description
`CHUNKSIZE(`*n*`)`	Specifies the number of iterations to distribute to an available processor. *n* is an integer expression. For best performance, *n* should be an integer constant. For example, given 100 iterations and `CHUNKSIZE(4)`, 4 iterations at a time are distributed to each available processor until the 100 iterations are complete.
`GUIDED[(`*vl*`)]`	Specifies the use of guided self-scheduling to distribute the iterations to available processors. This mechanism minimizes synchronization overhead while providing acceptable dynamic load balancing. The *vl* argument is the vector length. *vl* must be of type integer and can be either a constant or a variable. The default *vl* is 1.
`NUMCHUNKS(`*m*`)`	Specifies that the iterations are divided into *m* chunks of equal size (with a possible smaller residual chunk) and distribute these chunks to available processors. The *m* argument must be an integer constant. For example, given 100 iterations and `NUMCHUNKS(4)`, 25 iterations at a time are distributed to each available processor until the 100 iterations are complete.
`SINGLE`	Specifies that iterations should be distributed one at a time to available processors. This is the default distribution policy.
`VECTOR`	Distributes the maximum vector length (either 64 or 128 iterations at a time) to each processor. In addition, this *work_distribution* specification has the same effect as if you had specified an `IVDEP` directive. For information on the `IVDEP` directive, see Section 3.2.3, page 80.

A.5 Mark Parallel Loop: `DOPARALLEL` and `ENDDO`

The `!MIC$ DOPARALLEL` directive indicates that the `DO` loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a `DOPARALLEL` loop.

The `!MIC$ ENDDO` directive extends a control structure beyond the `DO` loop. Without a `!MIC$ ENDDO` directive, all of the CPUs will synchronize immediately after the loop, so that no processors can continue executing until all of the iterations are done. A `!MIC$ ENDDO` directive moves this point of synchronization from the end of the loop to the line of the `!MIC$ ENDDO` directive.

This lets the CF90 compiler use parallelism in loops containing some forms of reduction computations. These directives can be used only within a parallel region bounded by the PARALLEL and ENDPARALLEL directives.

Every variable in a parallel region must be declared as PRIVATE or SHARED.

The formats for these directives are as follows:

```
!MIC$ DOPARALLEL [work_distribution]

!MIC$ ENDDO
```

The *work_distribution* parameters are described in Table 8, page 186. Only one *work_distribution* parameter can be used for a given DO loop.

In the following example, a parallel region is defined by PARALLEL and ENDPARALLEL. A reduction computation is implemented by a DOPARALLEL/ENDDO pair, which ensures that all contributions to SUM and BIG are included, and GUARD/ENDGUARD, which protects the updating of shared variables SUM and BIG.

```
      SUM = 0.0
      BIG = -1.0
!MIC$ PARALLEL PRIVATE(XSUM,XBIG,I)
!MIC$1         SHARED(SUM,BIG,AA,BB,CC)
      XSUM = 0.0
      XBIG = -1.0
!MIC$ DOPARALLEL
      DO I = 1, 2000
      :
      XSUM = XSUM + (AA(I)*(BB(I)-CC(AA(I))))
      XBIG = MAX(ABS(AA(I)*BB(I)), XBIG)
      :
      ENDDO
!MIC$ GUARD
      SUM = SUM + XSUM
      BIG = MAX(XBIG,BIG)
!MIC$ ENDGUARD
!MIC$ ENDDO
!MIC$ ENDPARALLEL
```

A.6 Critical Region: GUARD and ENDGUARD

The !MIC$ GUARD and !MIC$ ENDGUARD directives delimit a critical region, providing the necessary synchronization to protect or guard the code inside the critical region. A *critical region* is a code block that is to be executed by only one processor at a time, although all processors that enter a parallel region will execute it.

The formats for these directives are as follows:

```
!MIC$ GUARD [n]

!MIC$ ENDGUARD [n]
```

n Mutual exclusion flag; two regions with the same flag cannot be active concurrently. *n* must be of type integer and can be a variable or an expression, from which the low-order 6 bits are used. For example, GUARD 1 and GUARD 2 can be active concurrently, but two GUARD 7 directives cannot.

For optimal performance, no flag should be specified. Otherwise, *n* should be an integer constant; a general expression can be used for the unusual case that the critical region number must be passed to a lower-level routine. When *n* is not provided, the critical region blocks only other instances of itself, but no other critical regions. Critical regions may appear anywhere in a program. That is, they are not limited to parallel regions.

You may receive incorrect results from your program if a routine being inlined has a dummy argument (under control of a !MIC GUARD directive) that is a scalar and the actual argument associated with that dummy argument is an array element reference. Inlining should be disabled for the following:

- A routine that contains a GUARD directive.
- A routine that is in the calling chain to a routine containing a GUARD directive.

The following example program may produce unexpected results if compiled with inlining enabled:

```
PROGRAM IMMEDIATE_DANGER
! PROGRAMS OF THIS TYPE MAY PRODUCE UNEXPECTED
! RESULTS IF COMPILED WITH INLINING ENABLED
DIMENSION IA(10)
```

```
      IA = 0
!MIC$ DOALL SHARED(IA) PRIVATE(I)
      DO I = 1, 10000
        CALL INC(IA(1))
      END DO
      PRINT *, IA(1)
      END
      SUBROUTINE INC(J)
!MIC$ GUARD
      J = J + 1
!MIC$ ENDGUARD
      END
```

A.7 Allocate CPUs: MAXCPUS

The !MIC$ MAXCPUS directive indicates the maximum number of CPUs that a section of code can use effectively. It does not guarantee that this number of processors will actually be assigned. The MAXCPUS directive is in effect until a subsequent MAXCPUS directive is encountered or the program unit is ended. The MAXCPUS directive is only in effect for the current program unit; it does not extend to subroutines called from that program unit. This directive affects only user-declared parallel regions.

The format for this directive is as follows:

```
!MIC$ MAXCPUS (ncpus)
```

ncpus Specifies the maximum number of CPUs that a section of code can use effectively. *ncpus* must be of type integer and can be a constant, variable, or expression.

When the MAXCPUS directive is used, it has the effect of adding the MAXCPUS parameter with the specified value to each subsequent !MIC$ DOALL or !MIC$ PARALLEL directive in the program unit. Without this directive, CPUs are allocated based on the NCPUS environment variable and workload.

The number of CPUs specified with this directive (*ncpus*) must be equal to or less than the number of CPUs specified by the NCPUS environment variable. If the number requested with the MAXCPUS directive is greater than the number specified by the NCPUS environment variable, no error is issued, but the directive has no effect.

Warning: The MAXCPUS directive will be removed in the CF90 3.2 release. In its place, use the MAXCPUS parameter on the DOALL or PARALLEL directive. For more information on the MAXCPUS parameter, see Table 7, page 184.

A.8 Specify Maximum Number of CPUs for a Parallel Region: NUMCPUS

The !MIC$ NUMCPUS directive globally indicates the maximum number of CPUs that a section of code can use effectively. It does not guarantee that this number of processors will actually be assigned. The NUMCPUS directive is in effect until a subsequent NUMCPUS directive is encountered. The NUMCPUS directive differs from the MAXCPUS directive in that it stays in effect across program units. The NUMCPUS directive remains in effect for all subsequently called subroutines. Without this directive, CPUs are allocated based on the NCPUS environment variable and workload.

The format for this directive is as follows:

```
!MIC$ NUMCPUS (ncpus)
```

ncpus Globally specifies the maximum number of CPUs that a code can use effectively. *ncpus* must be of type integer and can be a constant, variable, or expression.

The number of CPUs specified with this directive should be equal to or less than the number of CPUs specified by the NCPUS environment variable. If the number requested with the NUMCPUS directive is greater than the number specified by the NCPUS environment variable, no error is issued, but the directive has no effect.

A.9 Mark Parallel Region: PARALLEL and ENDPARALLEL

The !MIC$ PARALLEL and !MIC$ ENDPARALLEL directives mark, respectively, the beginning and end of a parallel region. Parallel regions are combinations of redundant code blocks and partitioned code blocks. The formats for these directives are as follows:

```
!MIC$ PARALLEL [parameter [ [,] parameter ] ...]

!MIC$ ENDPARALLEL
```

The *parameter*s are described in Table 7, page 184.

The PARALLEL directive indicates where multiple processors enter execution. The portion of code that all processors execute until reaching a DOPARALLEL directive is called a *redundant code block*. Because the iterations of the DO loop within a DOPARALLEL directive are distributed across available processors, this portion of code is called the *partitioned code block*. The scope of a variable in a parallel region is either shared or private. Shared variables are used by all processors; private variables are unique to a processor.

A.10 Declare an Array with No Repeated Values: PERMUTATION

The format for this directive is as follows:

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

This directive is also implemented with a !DIR$ prefix. For detailed information on this directive, see Section 3.2.6, page 82.

A.11 Declare a Cross-iteration Dependency: WAIT and SEND

The !MIC$ WAIT and !MIC$ SEND directives allow tasking of a loop by delimiting code that must be executed sequentially. This is useful for loops in which a substantial portion of code can be executed in parallel.

The WAIT and SEND directives must be contained within loops controlled by !MIC$ DO ALL or !MIC$ DO PARALLEL directives. The WAIT and SEND directives use the loop control variable to ensure sequential execution of the delimited code. The block that requires sequential execution must begin with a WAIT directive and end with a SEND directive. The formats for these directives are as follows:

```
!MIC$ WAIT [ POINT(n) ] [ SPAN(m) ]

!MIC$ SEND [ POINT(n) ] [ IF(condition) ]
```

POINT(*n*) — For *n*, specify an integer constant or a variable that can be used to identify a WAIT/SEND pair within a loop. Within a loop that contains more

than one section of dependent code, all WAIT/SEND pairs except one must be numbered. The value of *n* must be unique for each WAIT/SEND pair.

SPAN(*m*) — For *m*, specify an integer constant between 1 and 64, inclusive, that indicates the dependency span. The default is 1.

The dependency span is the number of iterations across which the dependency exists. For most loops with dependencies across iterations, the dependency span is 1. When it exceeds 1, you must use the SPAN option.

Multiple spans within a parallel region are not allowed.

All of the WAIT directives within a given DO ALL or DO PARALLEL loop must be specified either without any SPAN(*m*) value or with the same SPAN(*m*) value.

IF(*condition*) — Specifies a condition that must be met before execution can proceed past the SEND directive.

Note: The atexpert(1) utility adds the overhead from WAIT and SEND directives to the iteration overhead. Because the iteration overhead is typically small, a high value for this overhead often indicates WAIT/SEND overhead.

If the execution flow reaches a WAIT directive, it must also reach a corresponding SEND directive at some time during execution of the loop. If execution flow reaches a WAIT directive without also reaching a corresponding SEND directive, the loop may never complete because it is possible for a task to stop at the WAIT directive. Conversely, if execution flow reaches a SEND directive without also reaching the corresponding WAIT directive, the directives are not being used properly and the results may be incorrect.

The number of WAIT/SEND pairs that can be executed depends on the presence of a SPAN parameter, as follows:

- If no SPAN is specified, the maximum number of WAIT/SEND pairs allowed is 65. This includes one unnumbered WAIT/SEND pair and 64 WAIT/SEND pairs uniquely numbered from 1 to 64, inclusive. (In other words, one of the WAIT/SEND pairs can be specified without a POINT clause and the rest must

have POINT values in the range 1 through (64/*m*) – 1, where *m* is the SPAN value.)

- If the SPAN(*m*) parameter is specified, the maximum number of spanned WAIT/SEND pairs that can be specified is 64 divided by *m*, where *m* is the SPAN value.

  Example 1. If SPAN(4) is specified, no more than 16 WAIT/SEND pairs can be specified. This includes one unnumbered WAIT/SEND pair and 15 WAIT/SEND pairs uniquely numbered from 1 to 15, inclusive.

  Example 2. If SPAN(10) is specified, no more than 6 WAIT/SEND pairs can be specified (one unnumbered pair and numbered pairs from 1 to 5, inclusive).

Example 3. The following program's flow is incorrect because the SEND directive is never executed when the ELSE branch is taken:

```
PROGRAM WRONG1

!MIC$ WAIT
  IF . . .
  . . .

!MIC$ SEND
  . . .
    ELSE
  . . .
    ENDIF
```

Example 4. The following program's flow is incorrect because there are 100 SEND directives for 1 WAIT directive:

```
PROGRAM WRONG2

!MIC$ WAIT
  DO I=1,100
  . . .

!MIC$ SEND
  . . .
  ENDDO
```

Example 5. In the following loop, iteration I waits at the WAIT directive until iteration I-1 executes past the SEND directive.

```
!MIC$ DO ALL SHARED(...) PRIVATE(...)
      DO 10 I=2,N
         parallel work
!MIC$ WAIT
         F(I) = F(I-1)
!MIC$ SEND
         more parallel work
 10   CONTINUE
```

Example 6. A WAIT/SEND pair must be executed only once during each iteration of the DO ALL or DO PARALLEL loop. For the following loop, the SEND directive would be executed M times for each loop iteration. This is an incorrect use of the directives:

```
      PROGRAM WRONG
!MIC$ DO ALL SHARED(...) PRIVATE(...)
      DO 10 I=2,N
         parallel work
!MIC$ WAIT
         dependent work
         DO 20 J=1,M
         work
!MIC$ SEND
         work
 20      CONTINUE
         more parallel work
 10   CONTINUE
```

Correct use of SEND:

```
      PROGRAM RIGHT
!MIC$ DO ALL SHARED(...) PRIVATE(...)
      DO 10 I=2,N
         parallel work
!MIC$ WAIT
         dependent work
         DO 20 J=1,M
            work
 20      CONTINUE
!MIC$ SEND
         more parallel work
 10   CONTINUE
```

Example 7. The following example shows that the SEND directive is executed only once for each iteration of the DO 10 loop:

```
!MIC$ DO ALL SHARED(...) PRIVATE(...)
      DO 10 I=2,N
         parallel work
!MIC$ WAIT
         dependent work
         DO 20 J=1,M
            more dependent work
!MIC$ SEND IF (J.EQ.M)
         a large amount of work that is not dependent
               across iterations of the DO 10 loop
 20       CONTINUE
 10    CONTINUE
```

Without the IF option, the SEND directive would have to be placed after the 20 CONTINUE statement and only a small amount of parallelism could be exploited.

A.12 Autoscoping Rules

When the CF90 compiler generates code for a !MIC$ DOALL or a !MIC$ PARALLEL directive, one of the following must be true:

- All the variables and arrays in the region must be defined in a SHARED or PRIVATE parameter.
- The AUTOSCOPE parameter must be specified.

A *shared variable* or *array* is one that all the processors use. A *private variable* or *array* is one for which each of the processors has it own storage. If the AUTOSCOPE parameter is specified, the compiler analyzes a variable or array to determine whether it is shared or private. The following characteristics apply to private variables and arrays:

- The variable or array is written to and read from. The write operation must occur first. For more information, see Section A.12.2.4, page 198
- The loop control variable of the task loop does not appear in the subscript expression.

If the preceding conditions are not true, the variable is treated as a shared variable.

Example. Because of the suppress of J caused by subsequent directives, autoscoping determines that the loop index of the inner serial loop is shared:

```
!MIC$ DOALL AUTOSCOPE
      DO I = 1, N
        DO J = 1, M
          A(J, I) = I+J
        END DO
      END DO
!MIC$ DOALL ...     ! IMPLICIT SUPPRESS OF J
!DIR$ SUPPRESS J    ! EXPLICIT SUPPRESS OF J
```

If the compiler determines that the value of J should be retained upon exit from the parallel loop, it stores the last value of J in shared memory. In these cases, J is treated as a shared variable. The intermediate J index values in the parallel region are not stored and do not use this shared memory location.

The SHARED and PRIVATE parameters on a !MIC$ DOALL or a !MIC$ PARALLEL statement override the autoscope determination.

For more information on the AUTOSCOPE, SHARED, and PRIVATE parameters, see Table 7, page 184.

A.12.1 User-added Scope Required

If you want the scope of arguments to subroutine calls to be private, you must explicitly declare them as private. To be certain of correct scope, all variables or arrays that occur in a function or subroutine call should be specified as shared or private.

The following example shows use of a subroutine call:

```
!MIC$ DOALL AUTOSCOPE
      DO I = N1, N2
        CALL MMP(A(I), B, C)
      END DO
```

It is indeterminate whether A(I), B, or C is read or written. The CF90 compiler assigns A as shared because it is indexed by the control variable. The compiler assumes that B and C are read; therefore, it designates them as shared variables. The compiler prints a message stating that such variables as A, B, and C require a PRIVATE or SHARED declaration, but the compiler treats them as shared.

A.12.2 Examples

The following examples show shared and private variables and arrays.

A.12.2.1 Read-only Variables

The following examples show read-only variables:

```
!MIC$ DOALL PRIVATE(I) SHARED(N1,N2,A)
      DO I = N1, N2
      ...= A
      END DO
```

A is a shared variable because it is a read-only variable. All processors share the same location for A.

```
!MIC$ DOALL SHARED(N1,N2,M1,M2,V) PRIVATE(I,J)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
      ... = V(J)
  10  CONTINUE
```

V is shared because it is a read-only array. N1, N2, M1, and M2 are also shared because they are read-only variables. I and J are written and then read, so they are private variables.

A.12.2.2 Array Indexed by Loop Index

The following example shows an array indexed by the loop index:

```
!MIC$ DOALL SHARED(N1,N2,V,U,J) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
        U(I,J) = T
      END DO
```

U and V are shared arrays because they are indexed by the loop index. All processors share the same location for V and U. T is written and then read, so it is a private variable. J is shared because it is a read-only variable.

A.12.2.3 Read-then-write Variables

The following example shows read-then-write variables:

```
      SUM = 0.0
!MIC$ DOALL SHARED(N1,N2,V,SUM) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
!MIC$ GUARD
        SUM = SUM + T
!MIC$ ENDGUARD
      END DO
```

SUM is a shared variable because it is read before it is written. Special care is needed in writing into a shared variable that is not indexed by the loop control variable.

A.12.2.4 Write-then-read Variables and Arrays

The following example shows write-then-read variables and arrays:

```
!MIC$ DOALL SHARED(N1,N2,M1,M2) PRIVATE(I,J,V)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
      V(J) = ...
      ... = V(J)
   10 CONTINUE
```

V is written to and then read. It must be a private array.

A.13 Autotasking Restrictions

The following are some general Autotasking restrictions:

- Subprograms that contain an assigned GOTO statement cannot contain Autotasking directives.
- Branches out of a parallel region are not permitted and can produce incorrect results.
- A GUARD region must be obey the same nesting rules that apply to well-formed code blocks such as blocked IF or DO loop constructs. A GUARD region cannot begin in one code block and end in a different code block.
- Incorrect results may be produced, depending on how data items are stored in memory, when the following three conditions are all present:

– Two or more data objects appear on a SHARED parameter list of a microtasking PARALLEL or DOALL directive.

– The objects themselves, or components of the objects, are written by different processors.

– All or part of two such objects are stored in the same word of memory.

If the object is scoped SHARED by the AUTOSCOPE directive, the object is treated as if it appeared in an explicit SHARED parameter list. For example:

```
        CHARACTER*(9) B(1000)
!MIC$ DOALL SHARED(B) PRIVATE(I)
        DO I = 1, 1000
            B(I) = 'ABCDEFGHI'
        END DO
```

Incorrect results may be produced because the hardware requires all load and storage operations to be performed on aligned 64-bit words. When two processors attempt to update parts of the same work without the benefit of synchronization (that is, without using GUARD and ENDGUARD directives), the following events may take place:

– Processor 1 loads word W into register 1.

– Processor 2 loads word W into register 2.

– Processors 1 and 2 update registers 1 and 2, respectively.

– Processor 1 stores register 1 to word W.

– Processor 2 stores register 2 to word W.

At the end, the effect of processor 1 is lost. To prevent the preceding *race conditions* (related to false sharing) from occuring, do the following:

– Ensure that objects updated by different processors do not occupy part of a single word in memory. This is the preferred remedy because the additional storage costs are small and the overhead for critical sections is high.

– Prevent update race conditions by inserting GUARD and ENDGUARD directives.

Glossary

argument keyword

The name of a dummy (or formal) argument. This name is used in the subprogram definition; it also may be used when the subprogram is invoked to associate an actual argument with a dummy argument. Using argument keywords allows the actual arguments to appear in any order. The Fortran 90 standard specifies argument keywords for all intrinsic procedures. Argument keywords for user-supplied external procedures may be specified in a procedure interface block.

array

(1) A data structure that contains a series of related data items arranged in rows and columns for convenient access. The C shell and the `awk`(1) command can store and process arrays. (2) In Fortran 90, an object with the `DIMENSION` attribute. It is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1, and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

association

An association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. Several kinds of association exist. The principal kinds of association are pointer association, argument association, host association, use association, and storage association.

automatic variable

A variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length).

Autotasking

A trademarked process of Cray Research that automatically divides a program into individual tasks and organizes them to make the most efficient use of the computer hardware.

bottom loading

An optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

cache

In a processing unit, a high-speed buffer storage that is continually updated to contain recently accessed contents of main storage. Its purpose is to reduce access time. In disk subsystems, a method the channel buffers use to buffer disk data during transfer between the devices and memory.

cache line

On Cray MPP systems, a cache line consists of four quad words, which is the maximum size of a hardware message.

CIV

A constant increment variable is a variable that is incremented only by a loop invariant value (for example, in a loop with index J, the statement J = J + K, in which K can be equal to 0, J is a CIV).

constant

A data object whose value cannot be changed. A named entity with the PARAMETER attribute is called a named constant. A constant without a name is called a literal constant.

construct

A sequence of statements that starts with a SELECT CASE, DO, IF, or WHERE statement and ends with the corresponding terminal statement.

control construct

An action statement that can change the normal execution sequence (such as a GO TO, STOP, or RETURN statement) or a CASE, DO, or IF construct.

critical region

On Cray MPP systems, a synchronization mechanism that enforces serial access to a piece of code. Only one PE may execute in a critical region at a time.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (also called the function result). A data entity always has a type.

data object

A constant, a variable, or a part of a constant or variable.

declaration

A nonexecutable statement that specifies the attributes of a data object (for example, it may be used to specify the type of a variable or function result or the shape of an array).

definition

This term is used in two ways. (1) A data object is said to be defined when it has a valid or predictable value; otherwise, it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances, it may subsequently become undefined. (2) Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

derived type

A type that is not intrinsic (a user-defined type); it requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function with an interface and the generic specifier `OPERATOR`. Assignment for derived type objects is defined intrinsically, but it may be redefined by a subroutine with the `ASSIGNMENT` generic specifier. Data objects of derived type may be used as procedure arguments and function results, and they may appear in input/output (I/O) lists.

designator

Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of the name of the object followed by a selector that selects a part of the object. A name followed by a selector is called a **designator**.

entity

(1) In Open Systems Interconnection (OSI) terminology, a layered protocol machine. An entity in a layer performs the functions of the layer in one computer system, accessing the layer entity below and providing services to the layer entity above at local service access points. (2) In Fortran 90, a general term used to refer to any Fortran 90 concept (for example, a program unit, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface block, a derived type, an input/output (I/O) unit, a name list group, and so on).

executable construct

A statement (such as a GO TO statement) or a construct (such as a DO or CASE construct).

expression

A set of operands, which may be function invocations, and operators that produce a value.

extent

A structure that defines a starting block and number of blocks for an element of file data.

function

Usually a type of operating-system-related function written outside a program and called in to do a specific function. Smaller and more limited in capability than a utility. In a programming language, a function is usually defined as a closed subroutine that performs some defined task and returns with an answer, or identifiable return value.

The word "function" has a more specific meaning in Fortran than it has in C. In C, it is refers to any called code; in Fortran, it refers to a subprogram that returns a value.

generic specifier

An optional component of the INTERFACE statement. It can take the form of an identifier, an OPERATOR (defined_operator) clause, or an ASSIGNMENT (=) clause.

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the HEAP directive in SR-0066.

inlining

The process of replacing a user subroutine or function call with the 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 vectorization and/or tasking.

intrinsic

Anything that the language defines is intrinsic. There are intrinsic data types, procedures, and operators. You may use these freely in any scoping unit. Fortran programmers may define types, procedures, and operators; these entities are not intrinsic.

local

(1) A type of scope in which variables are accessible only to a particular part of a program (usually one module). (2) The system initiating the request for service. This term is relative to the perspective of the user.

multitasking

(1) The parallel execution of two or more parts of a program on different CPUs; these parts share an area of memory. (2) A method in multiuser systems that incorporates multiple interconnected CPUs; these CPUs run their programs simultaneously (in parallel) and shares resources such as memory, storage devices, and printers. This term can often be used interchangeably with **parallel processing**.

name

A term that identifies many different entities of a program such as a program unit, a variable, a common block, a construct, a formal argument of a

subprogram (dummy argument), or a user-defined type (derived type). A name may be associated with a specific constant (named constant).

operator

(1) A symbolic expression that indicates the action to be performed in an expression; operator types include arithmetic, relational, and logical. (2) In Fortran 90, an operator indicates a computation that involves one or two operands. Fortran 90 defines several intrinsic operators (for example, +, -, *, /, ** are numeric operators, and .NOT., .AND., and .OR. are logical operators). Users also may define operators for use with operands of intrinsic or derived types.

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 necessarily, leads to referencing a storage location outside of the entire array.

parallel processing

Processing in which multiple processors work on a single application simultaneously.

pointer

(1) A data item that consists of the address of a desired item. (2) A symbol that moves around a computer screen under the control of the user.

procedure

(1) A named sequence of control statements and/or data that is saved in a library for processing at a later time, when a calling statement activates it; it provides the capability to replace values within the procedure. (2) In Fortran 90, procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an ENTRY statement, it defines more than one procedure.

procedure interface

In Fortran 90, a sequence of statements that specifies the name and characteristics of one or more procedures, the name and attributes of each

dummy argument, and the generic specifier by which it may be referenced if any. See **generic specifier**.

In FORTRAN 77 and Fortran 90, a **generic function** is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating "generic interface," which is like a regular procedure interface, except that it has a "generic specifier" (the name of the generic function) after the keyword INTERFACE.

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.

reference

A data object reference is the appearance of a name, designator, or associated pointer in an executable statement that requires the value of the object. A procedure reference is the appearance of the procedure name, operator symbol, or assignment symbol in an executable program that requires execution of the procedure. A module reference is the appearance of the module name in a USE statement.

scalar

(1) In Fortran 90, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) A nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scope

The region of a program in which a variable is defined and can be referenced.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure

interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

search loop

A loop that can be exited by means of an IF statement.

sequence

A set ordered by a one-to-one correspondence with the numbers 1, 2, through **n**. The number of elements in the sequence is **n**. A sequence may be empty, in which case, it contains no elements.

shared

Accessible by multiple parts of a program. Shared is a type of scope.

shell variable

A name representing a string value. Variables that are usually set only on a command line are called **parameters** (positional parameters and keyword parameters). Other variables are simply names to which a user (user-defined variables) or the shell itself may assign string values. The shell has predefined shell variables (for example, HOME). Variables are referenced by prefixing the variable name by a $ (for example, $HOME).

software pipelining

Software pipelining is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to bottom loading, but it includes additional, and more efficient, scheduling optimizations.

Cray compilers perform safe bottom loading by default. Under these conditions, code generated for a loop contains operations and stores associated with the present loop iteration and contains loads associated with the next loop iteration. Loads for the first iteration are generated in the loop preamble.

When software pipelining is performed, code generated for the loop contains loads, operations, and stores associated with various iterations of the loop. Loads and operations for first iterations are generated in the preamble to the

loop. Operations and stores for last iterations of loop are generated in the postamble to the loop.

statement keyword

A keyword that is part of the syntax of a statement. Each statement, other than an assignment statement and a statement function definition, begins with a statement keyword. Examples of these keywords are `IF`, `READ`, and `INTEGER`. Statement keywords are not reserved words; you may use them as names to identify program elements.

stripmining

A single-processor optimization technique in which arrays, and the program loops that reference them, are split into optimally-sized blocks, termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

structure

A language construct that declares a collection of one or more variables grouped together under one name for convenient handling. In C and C++, a structure is defined with the `struct` keyword. In Fortran 90, a derived type is defined first and various structures of that type are subsequently declared.

subobject

Parts of a data object may be referenced and defined separately from other parts of the object. Portions of arrays are array elements and array sections. Portions of character strings are substrings. Portions of structures are structure components. Subobjects are referenced by designators and are considered to be data objects themselves.

subroutine

A series of instructions that accomplishes a specific task for many other routines. (A subsection of a user-written program of varying size and, therefore, function. It is written within the program. It is not a subsection of a routine.) It differs from a main routine in that one of its parameters must specify the location to which to return in the main program after the function has been accomplished.

TKR

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

type parameter

Two type parameters exist for intrinsic types: kind and length. 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. The length type parameter `LEN` indicates the length of a character string.

variable

(1) A name that represents a string value. Variables that usually are set only on a command line are called parameters. Other variables are simply names to which the user or the shell may assign string values. (2) In Fortran 90, data object whose value can be defined and redefined. A variable may be a scalar or an array. (3) In the shell command language, a named parameter. See also **`shell variable`**.

Index

D

E

F

N

O

T

U

V

W

X