MIPSpro™ Fortran 90 Commands and Directives Reference Manual

007–3696–004

New Features in this Document

This manual describes the commands and directives supported by the MIPSpro Fortran 90 compiler that runs on IRIX systems. New features for this release include the following:

- Support for the OpenMP Fortran Application Program Interface, Version 2.0, is included.

Record of Revision

Version	Description
3.0	August 1997 Original Printing. This printing supports the MIPSpro 7 Fortran 90 compiler, release 7.2, running on IRIX systems.
3.0.2	March 1998 This revision supports the MIPSpro 7 Fortran 90 compiler, release 7.2.1, running on IRIX systems.
003	April 1999 This release supports the MIPSpro 7 Fortran 90 compiler, release 7.3, running on IRIX systems.
004	September 2002 This release supports the MIPSpro Fortran 90 compiler, release 7.4, running on IRIX systems version 6.5 and later.

Contents

Figures

Tables

Examples

About This Manual

This manual describes the commands and directives for using the MIPSpro Fortran 90 compiler. This book is organized into the following chapters:

- Chapter 1, "Introduction", page 1, introduces the content of the manual and provides a general description of the compiler.
- Chapter 2, "The F90 Command Line", page 5, provides an overview of the `f90`(1) command, which you use to invoke the compiler. For complete details about using the compiler, see the `f90`(1) man page.
- Chapter 3, "General Directives", page 17, introduces the compiler directives and describes the general compiler directives recognized by the compiler.
- Chapter 4, "OpenMP Fortran API Multiprocessing Directives", page 27, describes the OpenMP Fortran API multiprocessing directives.
- Chapter 5, "CF90 Directives", page 39, describes CF90 compiler directives that are also supported by the compiler.
- Chapter 6, "Source Preprocessing", page 55, describes the source preprocessor.
- Chapter 7, "Interlanguage Calling", page 65, describes the interlanguage calling conventions used when calling a C/C++ function from a Fortran procedure and a Fortran procedure from a C function.
- Chapter 8, "The Auto-Parallelizing Option (APO)", page 81, describes the Auto-Parallelizing Option (APO) and the directives that accompany this feature. APO requires an additional license from Silicon Graphics, Inc. Please contact your sales representative if you are interested in using this feature.
- Appendix A, "Libraries", page 107, describes library routines available to you from Fortran programs.

Related Publications

The following documents contain information that may be useful:

- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*

- *MIPSpro Fortran Language Reference Manual, Volume 3*
- *Application Programmer's I/O Guide*
- *ProDev WorkShop: Overview*
- *SpeedShop User's Guide*
- *ProDev WorkShop: Debugger User's Guide*
- *ProDev WorkShop: Debugger Reference Manual*
- *ProDev WorkShop: Performance Analyzer User's Guide*
- *ProDev WorkShop: Tester User's Guide*
- *dbx User's Guide*
- *Origin 2000 and Onyx2 Performance Tuning and Optimization Guide*

Compiler Messages

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

Compiler Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the 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 language itself:

- Adams, J., W. Brainerd, and J. Martin. *Fortran 95 Handbook : Complete ISO/ANSI Reference*. MIT Press, 1997. ISBN 0262510960.
- Chapman, S. *Fortran 90/95 for Scientists and Engineers*. McGraw Hill Text, 1998. ISBN 0070119384.
- Chapman, S. *Introduction to Fortran 90/95*. McGraw Hill Text, 1998. ISBN 0070119694.
- Counihan, M. *Fortran 95 : Including Fortran 90, Details of High Performance Fortran (HPF), and the Fortran Module for Variable-Length Character Strings*. UCL Press, 1997. ISBN 1857283678.
- Gehrke, W. *Fortran 95 Language Guide*. Springer Verlag, 1996. ISBN 3540760628.
- International Standards Organization. *ISO/IEC 1539–1:1997, Information technology — Programming languages — Fortran*. 1997.
- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996. ISBN 0198518889.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: `http://docs.sgi.com`. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With

an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.

- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` *title* on a command line.

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, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Introduction

This manual describes the MIPSpro Fortran 90 compiler which runs on IRIX operating systems (6.2 or later). This manual describes the command line options, the directives, and related library routines used by the compiler. It does not discuss in detail any optimization or debugging techniques. See the Preface for a list of books that describe the optimization and debugging tools.

The compiler was developed to support the Fortran standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This standard, commonly referred to as *the Fortran 95 standard*, is ISO/IEC 1539–1:1997. Because the Fortran 95 standard is a superset of previous Fortran standards, the compiler compiles code written in accordance with previous Fortran standards.

The Fortran 95 standard is a revision to the Fortran 90 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 95 standard for vendors. To maintain conformance to the Fortran 95 standard, some compiler features may be changed in in future releases based upon the outcome of the outstanding interpretations to the standard.

The `f90`(1) Command

In the following example, the `f90`(1) command is used to invoke the compiler. The `-listing` option is specified to generate a source listing and a cross reference. File `pgm.f` is the input file. After compilation, you can run this program by entering the output file name as a command. In this example, the default output file name, `a.out`, is used.

```
% f90 -listing pgm.f
% ./a.out
```

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 the `f90`(1) man page..

The Compiler Programming Environment

The compiling environment allows you to develop, debug, and run Fortran codes on your computer system. It includes the following products:

- A preprocessor. By default, files suffixed with `.F` or `.F90` are run through the Fortran source preprocessor prior to compilation. You can use the `-ftpp` or `-cpp` options on the `f90`(1) command line to invoke a preprocessor for files without the `.F` or `.F90` suffix.
- A lister. You can specify the `-listing` option on the `f90`(1) command line to obtain a source listing and a cross reference. You can also invoke a separate lister, `ftnlist`(1).
- The `ftnlint`(1) utility, which checks Fortran programs for possible errors.
- The compiler information file (CIF) tools, which include the `cifconv`(1) command and the libraries.
- The libraries, which include optimized functions and intrinsics. Information on the individual library routines can be found in the online man pages for each routine.
- The performance tools contained in SpeedShop and in the ProDev ProMP suite. For more information on these products, see the *SpeedShop User's Guide* or the *ProDev WorkShop: ProMP User's Guide*.
- An archiving tool. An *archive library* is a file that contains one or more routines in object file format (*file*`.o`). When a program calls an object file that is not explicitly included in the program, the linker, `ld`(1), looks for that object file in an archive library. The scheduler then loads only that object file, not the whole library, and loads it with the calling program.

  The archiver creates and maintains archive libraries. It allows you to copy new objects into the library, replace existing objects in the library, move objects within the library, and copy individual objects from the library into individual object files. For more information on the archive library, see the `ar`(1) man page.
- Object file tools, which allow you to disassemble object files into machine instructions, print information about archive files, and perform other tasks. For more information on these tools, see the following man pages: `dis`(1), `elfdump`(1), `file`(1), `nm`(1), `size`(1), and `strip`(1).
- `ftnchop`(1), `ftnmgen`(1), and `ftnsplit`(1). These commands invoke a program unit problem isolator, a Fortran makefile utility, and a split utility, respectively. For more information on these commands, see the man pages for each.

- Online documentation utilities. The `man`(1) command allows you to retrieve online man pages. Prose reference text, such as this manual, can be retrieved through the WWW browser supported at your site. Contact your support staff for specific information on retrieving information in this manner.
- Modules. The compiler can be installed with the modules utility. This utility allows you to access different versions of the compiler and runtime environment. For more information on using the modules utility, see the `modules`(1) man page or enter the following command:

  ```
  % relnotes modules
  ```

- The message system. This system lets you obtain more comprehensive explanations of messages generated by the compiler and tools in the compiling environment. When a message condition occurs, both a message number and a verbal summary of the problem is generated. If you need more information on the error condition described in the summary, you can enter the `explain`(1) command to retrieve a more detailed description.
- Environment variables. For more information, see the `pe_environ`(5) man page, which describes many environment variables that can be used when compiling Fortran programs.

The F90 Command Line

This chapter provides an overview to the options for the f90(1) command. For complete details about each option, see the f90(1) man page.

The f90(1) command invokes the compiler. The following syntax box shows the complete f90(1) command syntax.

```
f90 [-64 | -n32] [-alignn] [-ansi] [-apo] [-apokeep] [-apolist]
  [-auto_use module_name[,module_name]...] [-bigp_off] [-bigp_on] [-c]
  [-C] [-check_bounds] [-chunk=integer] [-cif] [-coln] [-cord] [-cpp]
  [-dn] [-Dvar[=def][,var[=def]]...] [-DEBUG] [-default64] [-E]
  [-extend_source] [-fbfile] [fb_create path][-fixedform] [-flist]
  [fb_opt path] [-FLIST] [-freeform] [-ftpp] [-fullwarn] [-Gnum]
  [-g[debug_lvl]] [ -help] [-I[dir]] [-INLINE] [-in] [-ipa] [-IPA]
  [-ignore_suffix] [-KPIC] [-keep] [-Ldirectory] [-llibrary] [-LANG]
  [-LIST] [-LNO] [-listing] [-lscs] [-lscs_mp] [-macro_expand]
  [-MDupdate[file]] [-mipsn] [-mp] [-mplist] [-MP] [-mp_schedtype=mode]
  [-noappend] [-nocpp] [-noextend_source] [-nostdinc] [-Olevel]
  [-OPT] [-oout_file] [-P] [-pad_char_literals] [-pfa] [-pfakeep]
  [-pfalist] [-rreal_spec] [-rprocessor] [-S] [-show] [-show_defaults]
  [-static] [-static_threadprivate] [-TARG] [-TENV] [-Uvar]
  [-use_command] [-use_suffix] [-version] [-Wl,opt[,arg][,opt[,arg]]...]
  [-w[arg]] [-woffnum] [-x lang] [-xdirlist] [--] file.suffix[90]
  [file.suffix[90]...]
```

In some cases, more than one option can have an effect on a single compiler feature. The following list shows some of the compiler features and the options that affect them:

- Listing control: -flist, -FLIST:, -listing, -LIST:.
- Control of suffix interpretation: -ignore_suffix, -use_command, -use_suffix, -x *lang*.
- Source preprocessing: -cpp, -D*var*[=*def*][,*var*[=*def*]]..., -E, -F, -ftpp, -macro_expand, -nocpp, -P, -U*var*.
- Setting the compilation environment: -n32, -64, -mips*n*, -r*processor*, -TARG:, -TENV:.

- Optimization: `-apo`, `-LNO:`, `-OPT:`, `-O`*level*.

Note: The Auto-Parallelizing Option is invoked when you specify the `-apo` command line option. You must be licensed for the MIPSpro Auto-Parallelizing Option in order to be able to use this command line option.

Various environment variable settings can affect your compilation. For more information on the environment variables, see the `pe_environ`(5) man page.

Some `f90`(1) command options, for example, `-LNO:...`, `-LIST:...`, `-MP:...` , `-OPT:...`, `-TARG:...`, and `-TENV:...` accept several suboptions and allow you to specify a setting for each suboption. To specify multiple suboptions, either use colons to separate each suboption or specify multiple options on the command line. For example, the following command lines are equivalent:

```
f90 -LIST:notes=ON:options=OFF b.f
f90 -LIST:notes=ON -LIST:options=OFF b.f
```

Some arguments to suboptions of this type are specified with a setting that either enables or disables the feature. To enable a feature, specify the suboption either alone or with `=1`, `=ON`, or `=TRUE`. To disable a feature, specify the suboption with either `=0`, `=OFF`, or `=FALSE`. For example, the following command lines are equivalent:

```
f90 -LNO:auto_dist:blocking=OFF:oinvar=FALSE a.f
f90 -LNO:auto_dist=1:blocking=0:oinvar=OFF a.f
```

For brevity, this manual shows only the `ON` or `OFF` settings to suboptions, but the compiler also accepts 0, 1, `TRUE`, and `FALSE` as settings.

`f90` command line options

The following list summarizes the options to the `f90` command. For complete details, see the `f90`(1) man page.

`-n32`, `-64`

> Specifies the Application Binary Interface (ABI), either `-n32` or `-64`. Specifying `-n32` generates 32–bit objects. Specifying `-64` generates 64–bit objects.

`-align`*n*

Aligns data objects on 32– or 64– bit boundaries.

`-ansi`

Causes the compiler to generate messages when it encounters source code that does not conform to the Fortran standard.

`-apo`, `-apokeep`, `-apolist`

Controls the Auto-Parallelizing Option (APO), which automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so.

Note: These options are ignored unless you are licensed for the Auto-Parallelizing Option. For more information on this product contact, your sales representative.

`-auto_use` *module_name*[`,`*module_name*] `...`

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.

`-bigp_on`

Tells the compiler to enable the use of large pages within your program.

`-bigp_off`

Tells the compiler to disable the use of large pages within your program. This is the default for all optimization levels except `-Ofast`.

`-c`

Disables the load step and writes the binary object file to *file*`.o`.

`-C`, `-check_bounds`

Performs run–time array subscript range checking. These options are equivalent to the `-DEBUG:subscript_check` option. For more information on this option, see the `debug_group`(5) man page.

`-chunk=`*integer*

When compiling a multitasked program, this option specifies the number of loop iterations per chunk.

`-cif`

Generates a compiler information file (CIF) for use by the programming tools.

`-col`*n*

Specifies the line width for fixed-format source lines. Specify 72, 80, or 120 for *n*.

`-cord`

Runs the procedure rearranger, `cord`(1), on the resulting file after loading.

`-cpp`

Runs a nondefault source preprocessor, `cpp`(1), on all input source files, regardless of suffix, before compiling. This preprocessor automatically expands macros outside of preprocessor statements.

`-d`*n*

Specifies the `KIND` specification used for objects declared `DOUBLE COMPLEX` and `DOUBLE PRECISION`.

`-D`*var*[`=`*def*][`,`*var*[`=`*def*]]`...`

Defines variables used for source preprocessing as if they had been defined by a `#define` directive. If no *def* is specified, 1 is used. For information on undefining variables, see the `-U`*var* option.

`-DEBUG:...`

Controls the compiler's attempts to detect various errors (at compile time or run time) and controls how the errors are reported. For more information on the debugging options, see the `debug_group`(5) man page.

`-default64`

Sets the sizes of default integer, real, logical, and double precision objects. This option causes the following options to go into effect: `-r8`, `-i8`, `-d16`, and `-64`.

`-E`

Run only the source preprocessor files, without considering suffixes, and writes the result to `stdout`.

`-extend_source`

Specifies a 132-character line length for fixed-format source lines. By default, fixed-format lines are 72 characters wide. For more information on controlling line length, see the `-col`*n* option

`-fb`*file*

Specifies the feedback file to be used.

`-fb_create` *path*

Generates an instrumented executable program, which is suitable for producing one or more `.instr` files for subsequent feedback compilation.

`-fb_opt` *path*

Specifies the directory that contains the instrumentation output generated by compiling with `-fb_create` and then running your program with a training input set.

`-fixedform`

Treats all input source files, regardless of suffix, as if they were written in fixed source form. By default, only input files suffixed with `.f` or `.F` are assumed to be written in fixed source form.

`-flist`

Invokes all Fortran listing control options. Shows lowering, versioning, and tilling. The effect is the same as if all `-FLIST:...` options had been enabled.

`-FLIST:...`

Invokes the Fortran listing control group, which controls production of the compiler's internal program representation back into Fortran code, after IPA inlining and loop-nest transformations. This is used primarily as a diagnostic tool, and the generated Fortran code may not always compile.

`-freeform`

Treats all input source files, regardless of suffix, as if they were written in free source form. By default, only input files suffixed with `.f90` or `.F90` are assumed to be written in free source form.

`-ftpp`

Runs the Fortran source preprocessor on input Fortran source files that are suffixed with `.f` or `.f90` before compiling. By default, only files suffixed with `.F` or `.F90` are run through the Fortran source preprocessor.

`-fullwarn`

Requests that the compiler generate comment-level messages. These messages are suppressed by default. This option can be useful during software development.

`-G`*num*

Specifies the maximum size, in bytes, of a data item that is to be accessed from the Global Pointer (GP). *num* must be a decimal number.

`-g`*debug_lvl*

Generates debugging information and establishes a debugging level.

`-help`

Lists all available options. The compiler is not invoked.

`-i`*n*

Specifies the length of default integer constants, default integer variables, and logical quantities.

`-I`*dir*

Specifies a directory to be search for `INCLUDE` files.

`-ignore_suffix`

Compiles all files as if they were Fortran source files.

`-INLINE:…`

Specifies actions for the standalone inliner. For more information on the individual options in this group, see `ipa`(5).

`-ipa`

Invokes interprocedural analysis (IPA). Specifying this option is identical to specifing `-IPA` or `-IPA:`. Default settings for the individual IPA suboptions are used.

`-IPA`[`:…`]

Controls the application of interprocedural analysis (IPA) and optimization. This includes inlining, common block array padding, constant propagation, dead function elimination, alias analysis, and other features. Specify `-IPA` with no arguments to invoke the interprocedural analysis phase with default options. For more information on the individual options in this group, see the `ipa`(5) man page.

`-keep`

Writes all intermediate compilation files.

`-KPIC`

Generates position-independent code (PIC), which is necessary for programs loaded with dynamic shared libraries. Enabled by default.

`-l`*library*

Searches the library named `lib`*library*`.a` or `lib`*library*`.so`. Libraries are searched in the order given on the command line.

`-L`*directory*

Changes the library search algorithm for the loader.

`-LANG:...`

Controls the language option group.

`-LIST:...`

Writes an assembler listing file to *file*`.l`.

`-listing`

Writes a source code listing and a cross reference listing to *file*`.L`.

`-LNO:`…

Specifies options and transformations performed on loop nests by the Loop Nest Optimizer. For details about these options, see the `lno`(5) man page.

`-lscs` and `-lscs_mp`

Loads the SCSL Scientific library. The `-lscs_mp` option loads the multi-processor version of the library. This is a link-time option; if you compile and link programs separately, you only have to specify this option on the link line.

`-macro_expand`

Enables macro expansion in preprocessed Fortran source files throughout each file.

`-MDupdate`[*file*]

Updates makefile dependencies in *file*.

`-mips`*n*

Specifies the Instruction Set Architecture (ISA).

`-mp`

Generates multiprocessing code for the files being compiled. This option causes the compiler to recognize all multiprocessing directives and enables all `-MP:...` options.

`-MP:…`

Specifies individual multiprocessing options that provide fine control over certain optimizations.

`-mplist`

Generates *file*`.w2f.f`.

`-mp_schedtype=`*mode*

Specifies a default mode for scheduling work among the participating tasks in loops. This option must be specified in conjunction with `-mp`.

`-noappend`

Prevents the compiler from appending a trailing underscore character (_) on external names.

`-nocpp`

Disables the source preprocessor.

`-noextend_source`

Restricts Fortran source code lines to columns 1 through 72.

`-nostdinc`

Directs the system to skip the standard directory, `/usr/include`, when searching for `#include` files and files named on Fortran `INCLUDE` statements.

`-o`*outfile*

Writes the executable file to *out_file* rather than to `a.out`. By default, the executable output file is written to `a.out`.

`-O`*level*

Specifies the basic optimization level.

`-OPT:…`

Controls miscellaneous optimizations. These options override defaults based on the main optimization level. For details, see the `opt`(5) man page.

`-P`

Runs only the source preprocessor and puts the results for each source file (that is, for *file*`.f[90]`, *file*`.F[90]`, and/or *file*`.s`) in a corresponding *file*`.i`. The *file*`.i` that is generated does not contain # lines.

`-pad_char_literals`

Blank pads all character literal constants that are shorter than the size of the default integer type and that are passed as actual arguments. The padding extends the length to the size of the default integer type.

`-r`*processor*

Specifies the code scheduler.

`-r`*real_spec*

Specifies the default kind specification for real values.

`-S`

Generates an assembly file, *file*`.s`, rather than an object file (*file*`.o`).

`-show`

Print the passes as they execute with their arguments and their input and output files.

`-show_defaults`

List all defaults used in the compiler environment. This option does not compile the program.

`-static`

Statically allocates all local variables. Statically allocated local variables are initialized to zero and exist for the life of the program. This option can be useful when porting programs from older systems in which all variables are statically allocated.

`-static_threadprivate`

Makes all static variables private to each thread. This option can be specified in conjunction with the `-static` option, which statically allocates all local variables.

`-TARG:…`

Cross compiling is compiling a program on one system and executing it on another. To cross compile, you can either use the `-TARG:` command line options to control the target architecture and machine for which code is generated or you can set the `COMPILER_DEFAULTS_PATH` environment variable to specify the file that contains the default processor information needed to generate executable code for the target system.

`-TENV:…`

Specifies the target environment option group. The *target environment* is the system upon which the executable code will be run. These options control the target environment assumed and/or produced by the compiler.

`-U`*var*

Undefines a variable for the source preprocessor.

`-use_command`

Use the command name to determine which compiler to invoke for recognized source files.

`-use_suffix`

Use the file suffix to determine which compiler to invoke for recognized source files.

`-version`

Writes compiler release version information to `stdout`. No input file needs to be specified when this option is used.

`-w`[*arg*]

Specifies messages.

`-Wl,`*opt*[`,`*arg*][`,`*opt*[`,`*arg*]]`...`

Specifies options to be passed directly to the linker.

`-woff`*num*

Specifies message numbers to suppress.

`-x`*lang*

Specifies the programming language, regardless of suffix.

`-x`*dirlist*

Disables specified directives or specified classes of directives.

`--`

Separates options and file names. This option, which consists of two dashes, signifies the end of the options. After this symbol, you can specify the files to be processed. This is not allowed in non-XPG4 environments.

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

File or files to be processed, where *suffix* is either an uppercase `F` or a lowercase `f` for source files.

General Directives

A *directive* is a line inserted into Fortran source code that specifies actions to be performed by the compiler. Directive lines are not Fortran statements.

Many compiler features are implemented as either command line options or directives. The features implemented as command line options are set at compile time and applied to all files in the compilation. The features implemented through directives are set within your Fortran source code, and they apply to portions of your source code.

This chapter introduces the compiler directive set and describes the general directives. The sections in this chapter are as follows:

Using Directives

All directives are of the following form:

```
prefix  directive
```

prefix Each directive begins with a prefix. The prefix needed for each directive is shown in the directive's description. The following directive prefixes are used:

- `!*$*` and `C*$*`. These prefixes are used by the loop-nest directives described in this chapter.
- `!$OMP` and `C$OMP`. These prefixes are used by the OpenMP Fortran API multiprocessing directives described in Chapter 4, "OpenMP Fortran API Multiprocessing Directives", page 27.

- `!DIR$` and `CDIR$`. These prefixes are used by the Autotasking directives described in Chapter 5, "CF90 Directives", page 39.

The prefix used also depends on which Fortran source form you are using, as follows:

- If you are using fixed source form, begin a directive line with the characters `C`*prefix* or `!`*prefix*. The `!` or `C` character must appear in column 1. Beginning the directive with a `!` or `C` character ensures that other compilers will treat directive lines as comment lines.
- If you are using free source form, begin a directive line with the characters `!`*prefix*, followed by a space, and then one or more directives. The `!`*prefix* need not start in column 1, but it must be the first text on a line.

Because both fixed source form and free source form accept directives that start with the exclamation point (`!`), that is the initial character used in all directive syntax descriptions in this manual.

directive This is the specific directive's syntax. The syntax usually consists of the directive name. Some directives accept arguments. A directive's arguments, if any, are shown in the description for the directive itself.

The following sections describe the general format for directives and explain how directives are continued across source code lines.

Note: The multiprocessing directives supported in previous MIPSpro Fortran 90 releases are outmoded, and so are the `!$PAR`, `C$PAR`, `!$`, and `C$` directive prefixes. This technology is outmoded, but it is still supported for older codes that require this functionality. You are encouraged to modify your code using the OpenMP directives described in Chapter 4, "OpenMP Fortran API Multiprocessing Directives", page 27.

Directives and Command Line Options

Some compiler features can be activated on the command line and through compiler directives. The difference is that a command line setting applies to all files in the compilation, but a directive applies to only a program unit or to another specific part of a source file.

Generally, and by default, directives override command line options. There are exceptions to this rule, however. The exceptions, if any, are noted in the introductory text to each directive group.

Directive Range

The range of a particular directive depends on the directive itself, as follows:

- If a directive appears within a program unit, it applies only to that program unit. Within a program unit, many directives apply only to the loops that they immediately precede.
- If a directive appears outside a program unit (for example, prior to program code in a file) it applies to the entire file.

The descriptions for the individual directives indicate the range of the directive.

Directive Continuation and Other Considerations

It is sometimes necessary to continue a directive across one or more source code lines. The continuation character used and its placement within the directive line depends on the type of directive you are using. The introductory text for each directive group indicates the continuation character that is appropriate for that group.

For all directives in this chapter, the prefix for a directive line that is a continuation line is `!*$*&`.

Do not use source preprocessor (#) directives within multiline compiler directives.

LNO Directives

The loop nest optimization (LNO) directives control loop nest optimizations. By default, directives override command line options. To reverse this, and have command line options override the LNO directives, specify `-LNO:ignore_pragmas`. To continue a directive, the continuation line must begin with `!*$*&`.

The LNO directives are described in detail on the `lno`(5) man page. The following list summarizes the available directives:

- The AGGRESSIVEINNERLOOPFISSION directive specifies that the following loop should be split into as many loops as possible. In a loop nest, this directive must precede an inner loop.
- The BLOCKABLE directive specifies that it is legal to cache block the subsequent loops.
- The BLOCKINGSIZE and NOBLOCKING directives assert that the loop following the directive either is (or is not) involved in a cache blocking for the primary or secondary cache.
- The fission control directives specify whether the compiler should perform loop fission on the loops that immediately follow these directives.
- The fusion control directives specify whether the compiler should perform loop fusion on the loops that immediately follow these directives. Loop iterations may be peeled as needed during loop fusion. The limit of this peeling is 5, or the number specified by the `-LNO:fusion_peeling_limit` command line option.
- The loop interchange control directives specify whether or not the order of the following two or more loops should be interchanged. These directives apply to the loops that they immediately precede.
- The PREFETCH directive controls the MIPS IV prefetch instruction. Using this directive can increase performance in program units that are likely to encounter cache misses during execution. This directive applies only to the program unit in which it appears.

  When the directive is specified, the compiler estimates the memory references that will be cache misses, inserts prefetches for the misses, and schedules the prefetches ahead of their corresponding references. You can specify different levels of prefetching aggressiveness for the primary and secondary cache.
- The PREFETCH_MANUAL directive specifies whether the PREFETCH_REF and the PREFETCH_REF_DISABLE directives, which perform manual prefetches, should be respected or ignored within a subprogram.
- The PREFETCH_REF directive requests prefetching for a specific memory reference. This directive applies only to the loop nest that includes references to *array*, and the directive must immediately precede the loop nest.

  When this directive is specified, all references to *array* in the subsequent loop nest are ignored by the automatic prefetcher (if enabled).

- The PREFETCH_REF_DISABLE directive disables prefetching for all references to an array. This directive applies to all array references within the program unit.
- The UNROLL directive specifies loop unrolling. This directive applies to the loop that immediately follows the directive.

  Inner loop unrolling occurs automatically when -O2 or -O3 are in effect. Non-inner loop unrolling (and jam) occurs when -O3 is in effect.

Symbol Storage Directives

The following directives control symbol storage:

- ALIGN_SYMBOL
- FILL_SYMBOL
- FLUSH
- SECTION_GP
- SECTION_NON_GP

Control Symbol Alignment and Padding

The ALIGN_SYMBOL and FILL_SYMBOL directives control the way symbols are stored.

The ALIGN_SYMBOL directive aligns the start of *symbol* at a specified alignment boundary.

The FILL_SYMBOL directive pads *symbol* with additional storage so that the symbol is assured not to overlap (even partially) with any other data item within the storage of the specified size. The additional padding required is divided between each end of the specified variable. For example, a FILL_SYMBOL(X,L1CACHELINE) directive guarantees that X does not suffer from false sharing for the primary cache line.

The formats for these directives are as follows:

```
!*$* ALIGN_SYMBOL (symbol [, storage])

!*$* FILL_SYMBOL (symbol [, storage])
```

symbol Specify the name of a symbol. *symbol* can be a common block variable or a module name. *symbol* cannot be a component of a derived type, an array element, a common block, or blank common.

storage Specify the storage size. Specify one of the following values for *storage*:

storage	Action
L1CACHELINE	Specifies the machine-specific first-level cache line size, typically 32 bytes.
L2CACHELINE	Specifies the machine-specific secondary cache line size, typically 128 bytes.
PAGE	Specifies a machine-specific page. Typically 16 KB.
power-of-two	An integer value that is a power of 2. This is measured in bytes.

For common block variables, these directives are required at each declaration of the common block. Because the directives modify the allocated storage and its alignment for the named *symbol*, inconsistent directives can lead to undefined results.

The ALIGN_SYMBOL directive has no effect on fixed-size local symbols, such as simple scalars or arrays of known size (for example symbols declared as REAL(N) or REAL(A(3))). The directive continues to be effective for automatic arrays (stack-allocated arrays of dynamically determined size).

You cannot specify an ALIGN_SYMBOL directive and a FILL_SYMBOL directive for the same *symbol*.

Example 3-1 Controlling symbol alignment and padding

```
! X IS A COMMON BLOCK VARIABLE
      COMMON X!
      INTEGER(KIND=4) X
!*$* ALIGN_SYMBOL (X, 32)

!   X WILL START AT A 32-BYTE BOUNDARY.
!   WARNING: THE LAYOUT OF THE COMMON BLOCK WILL BE AFFECTED

!*$* ALIGN_SYMBOL (X, 2)
!   ERROR: CANNOT REQUEST AN ALIGNMENT LOWER THAN THE NATURAL
```

```
!   ALIGNMENT OF THE SYMBOL.

      REAL(KIND=8) Y
!   Y IS A COMMON BLOCK OR LOCAL VARIABLE
!*$* FILL_SYMBOL (Y, L2CACHELINE)


!   ALLOCATE EXTRA STORAGE BOTH BEFORE AND AFTER Y SO THAT
!   Y IS WITHIN AN L2CACHELINE (128 BYTES) ALL BY ITSELF.
!   THIS CAN BE USEFUL TO AVOID FALSE-SHARING BETWEEN MULTIPLE
!   PROCESSORS FOR THE CACHE LINE CONTAINING Y.
```

Declare a Synchronization Point

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.

Note: This directive has the same effect as the FLUSH directive described in the OpenMP Fortran API.

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:

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

var Variables to be flushed.

Specify Global Pointer Use

The compiler can reference global data by using the global pointer and an offset value. Using the global pointer (gp) is more efficient than constructing the address at each occurence, but because the offset size is limited to 16 bits, only a limited set of elements can be referenced using the global pointer.

The compiler places global data in gp-relative or non-gp-relative sections, but you can use the SECTION_GP and SECTION_NON_GP directives to specify the variables to go within the gp-relative section and the variables that need to be addressed explicitly.

The formats for these directives are as follows:

```
!*$* SECTION_GP (symbol [, symbol] ...)

!*$* SECTION_NON_GP (symbol [, symbol] ...)
```

symbol Enter one or more symbols. Separate multiple symbols with commas. Valid symbols are common block names, variables specified on SAVE statements, and module names. If a module name is specified, all storage in the module is affected. If a common block name is specified, it must be of the following form: /*name*/.

Inlining and IPA Directives

The following are the inlining and interprocedural analysis (IPA) directives:

- INLINE, NOINLINE
- IPA, NOIPA

Note: Neither inlining nor IPA are enabled by default. By default, the directives in this section, if present in your source code, are ignored. To enable the directives and turn on inlining and IPA, specify the `-INLINE:` option or the `-IPA:` option on your `f90`(1) command line. For more information on the command line interaction with these features, see the `f90`(1) or `ipa`(5) man page.

Inlining is the process of replacing a procedure reference with a copy of the procedure's code. This eliminates procedure call overhead and exposes the relationships between the procedure code, the return value, and the surrounding code. The `INLINE` and `NOINLINE` directives allow you to specify procedures that should be inlined.

Interprocedural analysis (IPA) is a compiler feature that includes inlining, common block array padding, constant propagation, dead procedure elimination, dead variable elimination, and global name optimizations. For detailed information on the IPA feature, see the `ipa`(5) man page. The `IPA` and `NOIPA` directives allow you to control IPA.

The formats of these directives are as follows:

```
!*$* INLINE location [(name [,name] ...)]

!*$* NOINLINE location [(name [,name] ...)]

!*$* IPA location [(name [,name] ...)]

!*$* NOIPA location [(name [,name] ...)]
```

location Specify one of the following for *location*:

location	Action
`HERE`	Specifies that routines named on the subsequent source code line should be inlined or should undergo IPA. Default.
`ROUTINE`	Specifies that the named function should be inlined or should undergo IPA everywhere it appears within the current routine.

GLOBAL — Specifies that the named function should be inlined or should undergo IPA throughout the source file.

name — For the inlining directives, each *name* specification represents one or more routines to be inlined. If no routines are named, all routines in the program are inlined.

For the IPA directives, each *name* specification represents one or more routines to undergo IPA. If no routines are named, all routines in the program undergo IPA.

Example 3-2 Inlining

Consider the following code fragment:

```
      DO I = 1,N
!*$* INLINE HERE (BETA)
         CALL BETA(I,1)
      ENDDO
      CALL BETA(N,2)
```

Using the specifier ROUTINE rather than HERE in this example would inline both calls to BETA. Note that -INLINE:=ON must be specified on the f90(1) command line when this code is compiled in order for the inlining directive to be recognized.

OpenMP Fortran API Multiprocessing Directives

This chapter provides an overview of the supported multiprocessing directives. 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.

The complete OpenMP standard is available at `http://www.openmp.org/specs`. See that documentation for complete examples, rules of usage, and restrictions. **This chapter provides only an overview of the supported directives and does not give complete details about usage or restrictions.**

To enable recognition of the OpenMP directives, specify `-mp` on the `f90`(1) command line. The `-mp` option must be specified in order for the compiler to honor any `-MP:...` options that may also be specified on the command line. The `-MP:open_mp=ON` option is on by default and must be in effect during compilation.

The following example command line can compile program `ompprg.f`, which contains OpenMP Fortran API directives:

```
f90 -mp ompprg.f
```

In addition to directives, the OpenMP Fortran API describes several library routines and environment variables. Information on these other utilities can be found in the following locations:

Programming Utility	Information Location
Command line information	For information on the `-mp` option, and the `-MP:` option, see the `f90` man page.
Library routines	`omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages
Environment variables	`pe_environ`(5) man page

Note: If individual loops in your program contain both OpenMP directives and extensions (prefixed with `!$OMP` or `!$SGI`) **and** any of the outmoded multiprocessing directives (prefixed with `!$` or `!$PAR`), you must specify the set of directives that the compiler should use. To direct the compiler to ignore the OpenMP directives, compile with `-MP:open_mp=OFF`. To direct the compiler to ignore the outmoded multiprocessing directives, compile with `-MP:old_mp=OFF`. To direct the compiler to ignore the outmoded distributed shared memory directives, specify `-MP:dsm=OFF`.

Note: The SGI multiprocessing directives, including the Origin series distributed shared memory directives, are outmoded. Their preferred alternatives are the OpenMP Fortran API directives described in this chapter.

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 4-1 OpenMP 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 4-2 OpenMP 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)
```

One or more blanks or tabs must be used to separate adjacent keywords in directives in free source form, except in the following cases where white space is optional between the keywords:

```
END CRITICAL
END DO
END MASTER
END ORDERED
END PARALLEL
END SECTIONS
END SINGLE
END WORKSHARE
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
```

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

Comments are allowed inside directives. Comments can appear on the same line as a directive. In free source form, the exclamation point initiates a comment; in fixed source form, it initiates a comment when it appears after column 6. Regardless of form, the comment extends to the end of the source line and is ignored. If the first nonblank character after the initial prefix (or after a continuation directive line in fixed source form) is an exclamation point, the line is ignored.

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 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.

Your program must be compiled with the `-mp` option in order for the compiler to honor statements preceded by conditional compilation prefixes; without the `mp` command line option, statements preceded by conditional compilation prefixes are treated as comments.

You must define the _OPENMP symbol to be used for conditional compilation. This symbol is defined during OpenMP compilation to have the decimal value YYYYMM where YYYY and MM are the year and month designators of the version of the OpenMP Fortran API is supported.

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.
- 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 prior to any comment appearing on the conditionally compiled line. Continuation lines can have an ampersand after the prefix, with optional white space before and after the ampersand.

Parallel Region Constructs

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.

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.

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. When a work-sharing construct is not enclosed dynamically within a parallel region,

it is treated as though the thread that encounters it were a team of size one. 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.

If NOWAIT is specified on the END DO, END SECTIONS, END SINGLE, or END WORKSHARE directive, an implementation may omit any code to synchronize the threads at the end of the worksharing construct. In this case, threads that finish early may proceed straight to the instructions following the work-sharing construct without waiting for the other members of the team to finish the work-sharing construct.

The following list summarizes the work-sharing constructs:

- 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. If an END DO directive is not specified, it is assumed at the end of the DO loop.

- 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.

  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 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 WORKSHARE directive divides the work of executing the enclosed code into separate units of work, and causes the threads of the team to share the work of

executing the enclosed code such that each unit is executed only once. The units of work may be assigned to threads in any manner as long as each unit is executed exactly once.

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 list describes the combined parallel work-sharing directives:

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

  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.

- The `PARALLEL SECTIONS/END PARALLEL SECTIONS` directives provide 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 `PARALLEL WORKSHARE/END PARALLEL WORKSHARE` directive provides a shortcut form for specifying a parallel region that contains a single `WORKSHARE` directive. The semantics are identical to explicitly specifying a `PARALLEL` directive immediately followed by a `WORKSHARE` directive.

Synchronization Constructs

The following list describe the synchronization constructs:

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

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

  A thread waits at the beginning of a critical section until no other thread 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 unspecified.

- 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.

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

- The 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)

  – Variables visible through host association

  –

  – Variables that appear in an EQUIVALENCE statement with a threat-visible variable

  – Local variables that 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 the enclosing parallel region.

  – Dummy arguments

  – All pointer dereferences

- 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.

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 information on directive binding, see "Directive Binding", page 37.

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.

Data Environment Constructs

The THREADPRIVATE directive makes named common blocks and named variables private to a thread but global within the thread.

Data Scope Attribute Clauses

In addition to the THREADPRIVATE directive, 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 are allowed on all directives; usually, if no data scope clauses are specified for a directive, the default scope for variables affected by the directive is SHARED.

The following list describes the data scope attribute clauses:

- The PRIVATE clause declares variables to be private to each thread in a team.
- 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.
- 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.
- The FIRSTPRIVATE clause provides a superset of the functionality provided by the PRIVATE 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.

- The REDUCTION clause performs a reduction on the variables specified, with the operator or the intrinsic specified.

  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 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.

- The COPYPRIVATE clause uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. It is an alternative to using a shared variable, or pointer association, and is useful when providing such a shared variable would be difficult. The COPYPRIVATE clause can only appear on the END SINGLE directive.

There are several rules and restrictions that apply with respect to data scope. See the OpenMP specification at http://www.openmp.org/specs for complete details.

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:

- A parallel region is available for binding purposes, whether it is serialized or executed in parallel.
- The DO, SECTIONS, SINGLE, MASTER, BARRIER, and WORKSHARE directives bind to the dynamically enclosing PARALLEL directive, if one exists. The dynamically enclosing PARALLEL directive is the closest enclosing PARALLEL directive regardless of the value of the expression in the IF clause, should the clause be present.
- 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.

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, SINGLE, and WORKSHARE directives that bind to the same PARALLEL directive cannot be nested one inside the other.
- DO, SECTIONS, SINGLE, and WORKSHARE 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, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.

- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- ORDERED directives must appear in the dynamic extent of a DO or PARALLEL DO directive which has an ORDERED clause.
- ORDERED directives are not allowed in the dynamic extent of SECTIONS, SINGLE, WORKSHARE, CRITICAL, and MASTER directives.
- CRITICAL directives with the same name are not allowed to be nested one inside the other.
- 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.

CF90 Directives

The MIPSpro Fortran 90 compiler, running on IRIX systems, recognizes some of the directives that are supported by the CF90 compiler on UNICOS and UNICOS/mk systems. The directives themselves and the sections in which they are discussed are as follows:

- "Using Directives", page 39, describes using directives.
- "Checking Array Bounds", page 41, describes the `BOUNDS` and `NOBOUNDS` directives.
- "Specifying Source Form", page 42, describes the `FREE` and `FIXED` directives.
- "Creating Identification String", page 43, describes the `ID` directive.
- "Ignoring Dummy Argument Type, Kind, and Rank", page 45, describes the `IGNORE_TKR` directive.
- "Ignoring Vector Dependencies", page 46, describes the `IVDEP` directive.
- "Mapping External Names", page 49, describes the `NAME` directive.
- "Inhibiting Loop Interchange", page 49, describes the `NOINTERCHANGE` directive.
- , describes the `NOSIDEEFFECTS` directive.
- "Designating a Nest to Task", page 51, describes the `PREFERTASK` directive.
- "Tasking Directives", page 51, describes the `TASK` and `NOTASK` directives.
- "Unrolling Loops", page 52, describes the `UNROLL` and `NOUNROLL` directives.

Using Directives

The following sections describe how to use the CF90 directives and the effects they have on IRIX platforms.

Directive Continuation

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

```
!DIR$ NA
!DIR$*ME
```

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.

Do not use source preprocessor (#) directives within multiline compiler directives.

Directive Range and Placement

The range and placement of directives is as follows:

- The FIXED and FREE directives can appear anywhere in your source code. All other directives must appear within a program unit.
- The BOUNDS/NOBOUNDS and TASK/NOTASK directives take effect at the point at which they appear in the source code.
- The ID and NOSIDEEFFECTS directives do not apply to any particular range of code. They add information to the *file*.o generated from the input program.
- The following directives apply only to the next loop encountered lexically:
  - IVDEP
  - NOINTERCHANGE
  - PREFERTASK
  - UNROLL/NOUNROLL

- The NAME and IGNORE_TKR directives do not apply to particular ranges of code. They are declarative directives that alter the status of entities in ways that affect compilation.

Interaction of Directives with the -x Command Line Option

The -x option on the f90(1) command line accepts one or more directives as arguments. When your input is compiled, the compiler ignores directives named as arguments to the -x option. For example, if you specify -x mipspro, all directives are ignored. If you specify -x *dirname*, the particular directive named in *dirname* is ignored.

Checking Array Bounds

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.

The -C option on the f90(1) command line 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)
!  DETECTED AT COMPILE TIME:
      A(11) = X
!  DETECTED AT RUN TIME IF IFUN(M) EXCEEDS 10:
      A(IFUN(M)) = W
```

The compiler generates a 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.

Bounds checking 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.

If bounds checking detects an out-of-bounds array reference, a message is issued and the program halts.

Specifying Source Form

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 -fixedform and -freeform options, if specified, on the f90(1) 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.

Creating Identification String

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 (`" "`).

Note: This directive is active only when the `-g3` and `-DEBUG:optimize_space=off` options are used.

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 sentinel characters `@(#)`. For example, assume that `id.f` contains the following source code:

  ```
  !DIR$ ID  "@(#)file.f 01 July 1997"
        PRINT *, 'hello'
        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 01 July 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.f` contains the following:

```
!DIR$ ID  'file.f 01 July 1997'
      PRINT *, 'Hello, world'
      END
```

The `what`(1) command generates the following output:

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

- Method 2. Using the `strings`(1) or `od`(1) command. 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: 1 July 1997"

         PRINT *, 'hello'
         END
```

  The `strings`(1) command generates the following output:

```
% f90 -c -g3 -DEBUG:optimize_space=off id.f
% strings id.o
File: id.f  Date: 1 July 1997
% od -c id.o
```

... portion of dump deleted

```
0002300  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0002320   F   i   l   e   :       i   d   .   f               D   a   t
0002340   e   :       1       J   u   l   y       1   9   9   7 001  \0
0002360  \0  \0  \0  \0 024 003 240 031  \0  \0 203 031  \0  \0 205 005
```

... portion of dump deleted

Ignoring Dummy Argument Type, Kind, and Rank

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 TKR rules, see the *MIPSpro Fortran Language Reference Manual, Volume 2*.

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. Dummy arguments for assumed-shape arrays or Fortran pointers cannot be specified.

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_TYPE 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.

Ignoring Vector Dependencies

The IVDEP directive directs the compiler to perform a more liberal dependency analysis for the purpose of software pipelining and other optimizations. The format of this directive is as follows:

```
!DIR$ IVDEP
```

This directive's effects depend on command line settings. When this directive is in effect, certain dependencies are ignored depending on the state of the following f90(1) command line options:

Option **Effect**

`-OPT:cray_ivdep=OFF`

Default command line setting. IRIX semantics are used when performing dependency analysis. Non-loop-carried dependencies in the subsequent loop are ignored between any two array references whenever the location referred to by at least one of the array references varies inside the loop. For more information on this command line option, see the opt(5) man page.

`-OPT:cray_ivdep=ON`

UNICOS semantics are used when performing dependency analysis. The compiler disregards backward dependencies only. For more information on this command line option, see the opt(5) man page.

`-OPT:liberal_ivdep=ON`

All dependencies are disregarded. For more information on this command line option, see the opt(5) man page.

The IVDEP directive applies only to inner loops, and it applies to the first DO loop that follows the directive within the same program unit.

Example 5-1 Data Dependency: non-loop-carried

There are two basic types of dependencies in the loop below: loop-carried and non-loop-carried. A *loop-carried dependency* occurs across iterations of the loop. A *non-loop-carried dependency* occurs within an iteration of the loop.

```
!DIR$ IVDEP
      DO I = 1,N
         A(INDEX(1,I)) = B(I)
         A(INDEX(2,I)) = C(I)
      END DO
```

A loop-carried dependency would occur if `INDEX(1,I)` in some iteration of `I` was equal to `INDEX(1,I+K)` in some other iteration of `I`. A non-loop-carried dependency would occur if `INDEX(1,I)` was equal to `INDEX(2,I)` in any iteration of `I`.

Example 5-2 Data Dependency: `IVDEP` directive

The following loop is executed with default command line options:

```
!DIR$ IVDEP
      DO I = 1,N
         A(B(K)) = A(C(K)) + D(I)
      END DO
```

Neither the reference to `A(B(K))` nor to `A(C(K))` vary inside the loop, so the `IVDEP` directive does not break the dependence.

Example 5-3 Data Dependency: broken dependence

The following loop is executed with default command line options:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I-1) + 3.0
      END DO
```

The `IVDEP` directive breaks the dependence, but the compiler issues a message indicating that an obvious dependence is being broken.

Example 5-4 Data Dependency: `IVDEP` broken dependence

The following loop is executed with default command line options, and the `IVDEP` directive breaks the dependence:

```
!DIR$ IVDEP
      DO I = 1,N
         A(B(I)) = A(B(I)) + 3.0
      END DO
```

Example 5-5 `IVDEP` and non-loop-carried dependence

The following loop is executed with default command line options, and the `IVDEP` directive does not break the dependence on `A(I)` because the dependence is non-loop-carried:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = B(I)
         C(I) = A(I) + 3.0
      END DO
```

Example 5-6 `OPT:cray_ivdep`

The following loop is executed with `-OPT:cray_ivdep=ON` in effect:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I-1) + 3.0
      END DO
```

The `IVDEP` directive breaks all lexically backward dependencies. When the loop is executed, however, the compiler issues a message indicating that it is breaking an obvious dependence.

Example 5-7 `IVDEP` and dependence

When the following loop is executed, the `IVDEP` directive does not break the dependence. This is because the dependence is from the load to the store, and the load comes lexically before the store. Assume that the code fragment in this example was compiled with `-OPT:cray_ivdep=ON`.

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I+1) + 3.0
      END DO
```

To break all dependencies, specify `-OPT:liberal_ivdep=ON`. Both `-OPT:cray_ivdep` and `-OPT:liberal_ivdep` are disabled by default.

For vector codes being transitioned to IRIX, it is recommended that `-OPT:cray_ivdep=ON` be used.

Mapping External Names

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. This directive must appear inside a program unit. 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
```

Inhibiting Loop Interchange

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
```

Determining Register Storage

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.

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 *MIPSpro Fortran Language Reference Manual, Volume 2*.

Designating a Nest to Task

The PREFERTASK directive allows loops with large iteration counts to be considered as candidates for tasking.

The compiler analyzes loops that follow a PREFERTASK directive to determine whether the loop is suitable for Autotasking. The PREFERTASK directive disables the compiler's threshold checking.

Note: The Autotasking directives are outmoded. SGI encourages you to write new codes using the OpenMP Fortran API directives.

This directive can be used if there is more than one loop in the nest that can be autotasked. Autotasking 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
```

Tasking Directives

The NOTASK directive suppresses compiler attempts to task loops and disables recognition of 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.

Note: The Autotasking directives are outmoded. SGI encourages you to write new codes using the OpenMP Fortran API directives.

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. After a `TASK` directive is specified, the compiler again attempts to autotask loops and array syntax statements and `!MIC$` 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.

Unrolling Loops

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

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

The 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 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 a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only 1 loop, and only the innermost loop can contain work.

The NOUNROLL directive inhibits loop unrolling.

Source Preprocessing

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). Files with these suffixes are preprocessed automatically by the Fortran preprocessor.

To specify preprocessing of source files with other extensions, including lowercase ones, use the `-cpp`, `-E`, `-ftpp`, or `-P` options to the `f90` command.

General Rules

You can alter the source code through source preprocessing directives. These directives are fully explained in "Directives", page 57. The directives must be used according to the following rules:

- Do not attempt macro substitution in Fortran comments. This will cause macros beginning with a `C` in column 1 (in fixed source form) not to be substituted.
- When the Fortran preprocessor is used, you must specify `-macro_expand` on the `f90`(1) command line if you want to enable macro expansion outside of preprocessor directive lines.
- Do not use source preprocessor (#) directives within multiline compiler directives.
- 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 (up to the column limit of 132).

  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, if a source preprocessing line spans lines, it must be continued by using a backslash character (\).
- 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.
- Because source preprocessing directives are independent of source form, a directive can be up to 132 columns on a single source line.

  Any directive text that extends past column 132 is ignored. The directive text is truncated, which is likely to produce parsing errors or unexpected results. If a directive is too long to fit on a single line, you can continue the line by using the backslash character (\). It cannot be continued using standard Fortran continuation methods.
- Any user-specified identifier that is used in a directive must follow Fortran rules for identifier formation. There are two exceptions to this rule:
  - The first character in the name can be an underscore character (_).
  - Although Fortran rules state that only the first 31 characters of identifiers are significant, to the source preprocessor, the first 132 characters are significant.
- Source preprocessing identifier names are case sensitive.
- Numeric literal constants must be integer literal constants or real literal constants, as defined for Fortran.
- Comments written in the style of the C language, beginning with `/*` and ending with `*/`, can appear anywhere within a source preprocessing directive in which blanks or tabs can appear. The comment, however, must begin and end on a single source line.
- 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.

Directives

The following sections describe the source preprocessing directives.

#include Directive

The `#include` directive directs the system to use the content of a file or directory. Just as with the `INCLUDE` line processing defined by the Fortran standard, an `#include` directive effectively replaces that directive line with 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 standard prohibits recursion in `INCLUDE` files, so recursion is also prohibited in the `#include` form.

The `#include` directives can be nested.

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

#define Directive

The #define directive lets you declare a source preprocessing variable and associate a token string with the variable. It also allows you to define a function-like macro. This directive has the following formats:

```
#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	Specifies the name of the variable or macro being defined.
dummy_arg_list	Specifies a list of dummy argument identifiers.
value	Specifies the *value* as 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 "#undef Directive", page 59.

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. By default, tokens are not processed in Fortran source code. They are recognized only when used in other source preprocessing directives.

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.

The following two examples must be compiled with `-macro_expand` specified on the `f90`(1) command line:

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

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

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

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

#undef Directive

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

```
#undef identifier
```

identifier Specifies the name of the source preprocessing variable or macro being undefined.

(Null) Directive

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

Conditional Directives

Conditional directives cause lines of code to either be produced by the source preprocessor or to be skipped. The conditional directives within a source file form *if-groups*. An if-group begins with an `#if`, `#ifdef`, or `#ifndef` directive, followed by lines of source code that you may or may not want skipped. Several similarities exist between the Fortran `IF` construct and if-groups:

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

  For example, you can write the following directives:

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

Determining which group of source lines (if any) to compile in an if-group is essentially the same as the Fortran determination of which block of an `IF` construct should be executed.

`#if` Directive

The `#if` directive has the following format:

```
#if expression
```

expression An expression. The values in *expression* must be integer literal constants or previously defined preprocessor variables. The expression is an integer constant expression as defined by the C language standard. All the operators in the expression are C operators, not Fortran operators. The *expression* is evaluated according to C language rules, not Fortran expression evaluation rules.

Note that unlike the Fortran `IF` construct and `IF` statement logical expressions, the *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).

`#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 the `-D` option on the `f90`(1) command line.

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*`)`

`#ifndef` Directive

The `#ifndef` directive tests for the presence of an *identifier* that is not defined. This directive has the following format:

```
#ifndef identifier
```

This directive is equivalent to either of the following two directives:

- `#if ! defined` *identifier*
- `#if ! defined(`*identifier*`)`

`#elif` Directive

The `#elif` directive serves the same purpose in an if-group as does the `ELSE IF` statement of a Fortran `IF` construct. This directive has the following format:

```
#elif expression
```

expression The expression follows all the rules of the integer constant expression in an `#if` directive.

`#else` Directive

The `#else` directive serves the same purpose in an if-group as does the `ELSE` statement of a Fortran `IF` construct. This directive has the following format:

```
#else
```

`#endif` Directive

The `#endif` directive serves the same purpose in an if-group as does the `END IF` statement of a Fortran `IF` construct. This directive has the following format:

```
#endif
```

Predefined Macros

The source preprocessor supports a number of predefined macros. They are divided into groups as follows:

- Macros that are based on the host machine
- Macros that are based on IRIX system targets

The following predefined macros are based on the host system (the system upon which the compilation is being done):

Macro	Notes
`__unix`	Always defined. The leading characters consist of 2 consecutive underscores.

The following predefined macros are based on an IRIX system target:

Macro	Notes
`_ABI`*abi*`=`*n*	Defined when *abi* is set to `N32` or `64`. Its value is the instruction set architecture. For example, `_ABIN32=2` is set when `-n32` is specified on the `f90`(1) command line; `_ABI64=3` is set when `-64` is specified on the `f90`(1) command line. For information on the `f90`(1) command line, see the `f90`(1) man page.
`_COMPILER_VERSION`	Defined as the compiler version. For example, for the MIPSpro 7.2.1 release it is set as follows: `_COMPILER_VERSION=721`.
`LANGUAGE_FORTRAN90`, `_LANGUAGE_FORTRAN90`	
`__host_mips`	The leading characters in the second form consist of 2 consecutive underscores.

`LANGUAGE_FORTRAN`, `_LANGUAGE_FORTRAN`	
`MIPSEB`, `_MIPSEB`	
`__mips`	Set to the instruction set architecture, either `3` or `4`. The leading characters consist of 2 consecutive underscores.
`_MIPS_ISA`	Set to the instruction set architecture, either `3` or `4`.
`_MIPS_SIM`	Set to the instruction set architecture, as follows: `_MIPS_SIM=_ABIN32` when `-n32` is specified on the `f90`(1) command line; `_MIPS_SIM=_ABI64` when `-64` is specified on the `f90`(1) command line.
`_OPENMP`	
`__sgi`	The leading characters consist of 2 consecutive underscores.
`_SYSTYPE_SVR4`	

Interlanguage Calling

You may want to call external procedures written in C, C++, or some other language, or you may need to call a Fortran procedure from one of those languages. This chapter focuses on the interface between Fortran and C/C++.

If your application has source programs written in different languages, you should compile each file separately, with the appropriate compiler, and then load them in a separate step. You can create object files suitable for loading by specifying the `-c` option on the `f90`(1) command, which disables the load step and writes the binary file to *file*`.o`.

In the following example, the C/C++ compiler and the Fortran compiler produce object files that can be loaded. These files are named `main.o` and `rest.o`:

```
%   cc -c main.c
%   f90 -c rest.f
```

This chapter provides more details on compiling and loading application programs that are written in Fortran, C, and C++.

External and Public Names

When your Fortran program defines the body of a procedure, the compiler places the name of the procedure, as a character string, in the object file it generates. This is a *public name*, which is accessible to other object files.

When your Fortran program uses a procedure, the compiler places the name of the procedure in the generated object file. This is an *external name*, which is used by the object file but not defined in it. Names of common blocks and names of data and procedures declared within object files are also external names. You can use the `nm`(1) utility to display the public and external names defined in a file.

It is up to the linker to resolve each reference to an external name by finding that same name as a public name in some other module. This is the main job of the linker.

Fortran Treatment of External and Public Names

The Fortran compiler ignores the case of the input source text (other than the contents of character literals). As a result, it may change the case of the names of procedures and named common blocks while it translates the source file. The names recorded in the object file are changed in the following two ways from the way you may have written them:

- They are converted to all lowercase letters.
- They are normally extended with a final underscore (_) character.

Procedure names and common block names are translated, too.

The following declarations produce the identifiers `matrix_`, `mixedcase_`, and `cblk_` in the object file:

```
SUBROUTINE MATRIX
external function MixedCase()
COMMON /CBLK/a,b,c
```

These changes cause no problems when loading program units compiled by Fortran. The same convention is used for both the public and external names, so the names match.

Note: Some IRIX-based FORTRAN 77 compilers support the `-U` command line option, which prevents the compiler from forcing all uppercase input to lowercase. As a byproduct, it becomes possible to put mixed case public names in the object file. This option is not supported by the MIPSpro Fortran 90 compiler.

In addition, some IRIX-based FORTRAN 77 compilers take the use of the `$` character as the final letter of a procedure name as a signal to suppress the underscore in the public name. The `$` is not permitted to appear in a name if the program is to be compiled by the MIPSpro Fortran 90 compiler. Use the `noappend` option to prevent the compiler from appending a trailing underscore character.

You can override the default conventions by using the `!DIR$ NAME` directive described in "Mapping External Names", page 49.

Module and internal procedure names are connected with `.in.` to make a unique name. For example, the following code creates procedures named `MPROC.in.MMM` and `IPROC.in.MPROC.in.MMM`:

```
MODULE MMM
...
CONTAINS
  SUBROUTINE MPROC()
  ...
  CONTAINS
    SUBROUTINE IPROC()
    ...
```

Calling a Fortran Subprogram from C

To call a Fortran subprogram from a C procedure, spell the name the way the Fortran compiler spells it, using all lowercase letters and a trailing underscore.

For example, consider the following Fortran declaration:

```
SUBROUTINE HYPOT()
```

This must be declared in a C function as follows (note the use of lowercase with a trailing underscore):

```
extern int hypot_()
```

Note: You cannot call Fortran subroutines that contain assumed-shape dummy arguments or Fortran pointer arguments from C.

Calling a C Function from Fortran

To call a C function from a Fortran program, ensure that the C function's name is spelled the way the Fortran compiler expects it to be. When you control the name of the C function, the simplest solution is to give it a name that consists of lowercase letters with a terminal underscore. For example, the following C function:

```
int fromfort_() {...}
```

could be declared in a Fortran program as follows:

```
external FROMFORT
```

When you do not control the name of a C function, you must either supply a function name that the Fortran compiler can call or use the `!DIR$ NAME` directive described in

"Mapping External Names", page 49. If you choose to supply a function name that the Fortran compiler can call, you need to write a C function that accepts the same arguments and has a name composed of lowercase letters followed by an underscore. This C function can then call the function whose name contains mixed case letters. You can write such a wrapper function manually, or you can use the `mkf2c`(1) utility to do it automatically.

Correspondence of Fortran and C Data Types

When you exchange data between Fortran and C, either as arguments, as function results, or as members of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data object.

Corresponding Scalar Types

The correspondence between Fortran and C scalar data types is shown in Table 7-1. This table assumes that the default command line options that affect precision are in effect. Certain `f90`(1) command line options (such as `-i2` or `-r8`) affects storage sizes for integer, logical, real, and double precision data types. For information on the `-i2` and `-r8` options, see the `f90`(1) man page.

Table 7-1 Corresponding Fortran and C Data Types

Fortran Data Type Declaration	C Data Type
`INTEGER(KIND=1)`, `LOGICAL(KIND=1)`	`signed char`
`CHARACTER`	`unsigned char`
`INTEGER(KIND=2)`, `LOGICAL(KIND=2)`	`short`
`INTEGER`, `INTEGER(KIND=4)`, `LOGICAL`, `LOGICAL(KIND=4)`	`int`
`INTEGER(KIND=8)`, `LOGICAL(KIND=8)`	`long long`
`REAL`, `REAL(KIND=4)`	`float`

Fortran Data Type Declaration	C Data Type
DOUBLE PRECISION, REAL(KIND=8)	double
REAL(KIND=16)	long double
COMPLEX, COMPLEX(KIND=4)	struct{float real, imag;};
DOUBLE COMPLEX, COMPLEX(KIND=8)	struct{double real, imag;};
COMPLEX(KIND=16)	struct{long double real, imag;};
CHARACTER(*n*)	char fstr_*n*[*n*]

For type character, Fortran declarations with a length designator, such as CHARACTER(LEN=N) :: X, are equivalent to a C declaration of unsigned char X[N].

To set a NULL character in a Fortran string, use CHAR(0), as in this example:

```
character*4 aaa
aaa(1:3) = 'abc'
aaa(4:4) = CHAR(0)
```

Corresponding Character Types

The Fortran CHARACTER data type declaration corresponds to the C type unsigned char. However, the two languages differ in the treatment of strings of characters.

A Fortran variable can be declared as CHARACTER(*n*), where *n*>1, contains exactly *n* characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach *n* characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran string (except by chance memory alignment).

There is no terminal null byte, so most of the string library functions familiar to C programmers (strcpy()(3c), strcat()(3c), strcmp()(3c), and so on) cannot be used with Fortran string values. The strncpy()(3c), strncmp()(3c), bcopy()(3c),

and `bcmp()`(3c) functions can be used because they depend on a count rather than a delimiter.

Unsupported Array Arguments

Fortran supports assumed-shape arrays, deferred-shape arrays, and array sections. You cannot pass any of these types of array to a non-Fortran procedure because Fortran represents such arrays in memory using a descriptor containing indirect pointers and other data. The format of this descriptor is not part of the published programming interface to the compiler because it is subject to change.

How Fortran Passes Arguments

When calling non-Fortran functions, you must know how arguments are passed. When calling Fortran subprograms from other languages, you must cause the other language to pass arguments correctly.

Note: All compilers for a given version of an operating system use identical conventions for passing arguments. These conventions are documented at the machine instruction level in the Assembly Language manual for the system.

An argument passed to a subprogram, regardless of its data type, is passed as the address of the actual in memory. This rule is extended for two special cases:

- The length of each `CHARACTER(`*n*`)` declaration is passed as an implicit additional `INTEGER(KIND=4)` value, following the explicit arguments.
- When a function returns type `CHARACTER(`*n*`)`, the address of the space to receive the result is passed as the first argument to the function, and the length of the result space is passed as the second implicit argument, preceding all explicit arguments.

Example 7-1 Argument passing

Consider the following code:

```
COMPLEX(KIND=8) :: CMPLX8
CHARACTER*(16) :: CSTR1, CSTR2
EXTERNAL CPXASC
CALL CPXASC(CSTR1,CSTR2,CMPLX8)
```

The code generated from the subroutine call in this example passes the following arguments:

- The address of CSTR1
- The address of CSTR2
- The address of CMPLX8
- The length of CSTR1, an integer value of 16
- The length of CSTR2, an integer value of 16

Example 7-2 Argument passing (continued)

Consider the following code:

```
CHARACTER*(8) :: SYMBL,PICKSYM
CHARACTER*(100) :: SENTENCE
INTEGER NSYM
SYMBL = PICKSYM(SENTENCE,NSYM)
```

The code generated from the function call in the preceding example passes the following arguments:

- The address of an unnamed result variable
- The length of an unnamed result variable
- The address of SENTENCE, the first explicit argument
- The address of NSYM, the second explicit argument
- The length of SENTENCE, an integer value of 100

Calling Fortran from C

There are two types of callable Fortran subprograms: subroutines and functions. In C terminology, both types of subprograms are external functions. The difference is the use of the function return value from each.

Calling a Fortran Subroutine from C

From the standpoint of a C function, a Fortran subroutine is an external function returning `void`.

Example 7-3 Calling Fortran from C

The following example shows a simple Fortran subroutine that adds arrays of complex numbers:

```
SUBROUTINE ADDC32(Z, A, B, N)
INTEGER :: N
COMPLEX(KIND=16),DIMENSION(N) :: Z,A,B
Z = A + B
RETURN
END SUBROUTINE
```

The Fortran subroutine could be called from C using the following code fragment:

```
typedef struct{long double real, imag;} cpx32;
extern void
   addc32_(cpx32 *,cpx32 *,cpx32 *,int *);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
...
      int n = MAXARRAY;
      addc32_(&z, &a, &b, &n);
```

The preceding code fragments show how the Fortran subroutine is named in the C code using lowercase letters and a terminal underscore. This is the way the Fortran compiler spells the public name in the object file.

Example 7-4 Calling Fortran from C (continued)

The following subroutine takes assumed-length character arguments:

```
SUBROUTINE PRT(BEF, VAL, AFT)
CHARACTER*(*) :: BEF, AFT
REAL :: VAL
PRINT *, BEF, VAL, AFT
RETURN
END SUBROUTINE PRT
```

The following C code prepares `CHARACTER(16)` values and passes them to the Fortran subroutine:

```
typedef char fstr_16[16];
extern void
     prt_(fstr_16 *, float *, fstr_16 *,
                  int, int);
main()
{
      float val = 2.1828e0;
      fstr_16 bef,aft;
      strncpy(bef,"Before.........",sizeof(bef));
      strncpy(aft,".........After",sizeof(aft));
      prt_(bef, &val, aft, sizeof(bef), sizeof(aft));
}
```

Note that the subroutine call requires five actual arguments: the addresses of the three explicit arguments and the lengths of the two string arguments. In the C code, the string length arguments are generated using `sizeof()`, which returns the memory size of the typedef `fstr_16`.

When the Fortran code does not require a specific string length, the C code that calls it can pass an ordinary C character vector, as shown in the following code fragment:

```
extern int
prt_(char *, float *, char *, int, int);
main()
{
          float val = 2.1828e0;
          char *bef = "Start:";
          char *aft = ":End";
          (void)prt_(bef, &val, aft, strlen(bef), strlen(aft));
}
```

In this example, the string length implicit argument values are calculated dynamically using `strlen()`.

Calling a Fortran Function from C

A Fortran function that returns a scalar value as its result corresponds exactly to the C concept of a function with an explicit return value. When a Fortran function returns any type shown in Table 7-1, page 68, other than `CHARACTER(`*n*`)`, where *n*>1, you can call the function from C and handle its return value exactly as if it were a C function returning that data type.

Example 7-5 Calling Fortran functions

The following function accepts and returns `COMPLEX(KIND=8)` values.

```
FUNCTION FSUB8(INP)
COMPLEX(KIND=8) :: INP,FSUB8
FSUB8 = INP
END FUNCTION FSUB8
```

Although a complex value is declared as a structure in C, it can be used as the return type of a function. The following C code shows how the preceding Fortran function is declared and called:

```
typedef  struct{ double real, imag; } cpx8;
extern cpx8 fsub8_(cpx8 *);
main()
{
      cpx8 inp = { -3.333, -5.555 };
      cpx8 oup = { 0.0, 0.0 };
      printf("testing fsub8...");
      oup = fsub8_( &inp );
      if ( inp.real == oup.real && inp.imag == oup.imag )
          printf("Ok\n");
      else
          printf("Nope\n");
}
```

The arguments to a function, like the arguments to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Example 7-6 Calling functions

The following function has a `CHARACTER(16)` return value.

```
FUNCTION FS16(J, K, S)
   CHARACTER*(16) :: FS16, S
   INTEGER J, K
   FS16 = S(J:K)
RETURN
END FUNCTION FS16
```

When a Fortran function returns `CHARACTER(`*n* `)`, where *n*>1, value, the returned value is not the explicit result of the function. Instead, you must pass the address and

length of the result area as the first two arguments of the function, preceding the explicit arguments. This is demonstrated in the following C code:

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *, int, int *, int *, fstr_16 *, int);
main()
{
      char work[64];
      fstr_16 inp, oup;
      int j = 7;
      int k = 11;
      strncpy(inp,"0123456789abcdef", sizeof(inp));
      fs16_ ( oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
      strncpy(work, oup, sizeof(oup));
      work[sizeof(oup)] = '\0';
      printf("FS16 returns <%s>\n", work);
}
```

In this example, the address and length of the function result are the first two arguments of the function. Because type `fstr_16` is an array, its name, `oup`, evaluates to the address of its first element. The next three arguments are the addresses of the three named arguments. The final argument is the length of the string argument.

Calling C from Fortran

You can call units of C code from Fortran as if they were written in Fortran, provided that the C modules follow the Fortran conventions for passing arguments. For more information on this, see "How Fortran Passes Arguments", page 70.

When the C function expects arguments passed using other conventions, you normally need to build a wrapper for the C function using the `mkf2c`(1) command.

Calls to C Functions

The following C function is written to use the Fortran conventions for its name (lowercase with final underscore) and for argument passing:

```
/*
|| C functions to export the facilities of strtoll()
```

```
|| to Fortran programs.  Effective Fortran declaration:
||
|| FUNCTION ISCAN(S,J)
|| INTEGER(KIND=8) :: ISCAN
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the nonspace character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
   int  scanPos, scanLen;
   long long ret = 0;
   char wrk[1024];
   char *endpt;
   /* when J>LEN(S), do nothing, return 0 */
   if (ls >= *pj)
   {
      /* convert J to origin-0, permit J=0 */
      scanPos = (0 < *pj)? *pj-1 : 0 ;

      /* calculate effective length of S(J:) */
      scanLen = ls - scanPos;

      /* copy S(J:) and append a null for strtoll() */
      strncpy(wrk,(ps+scanPos),scanLen);
      wrk[scanLen] = '\0';

      /* scan for the integer */
      ret = strtoll(wrk, &endpt, 0);
```

```
      /*
      || Advance over any whitespace following the number.
      || Trailing spaces are common at the end of Fortran
      || fixed-length char vars.
      */
      while(isspace(*endpt)) { ++endpt; }
      *pj = (endpt - wrk)+scanPos+1;
   }
   return ret;
}
```

The following Fortran code fragment demonstrates a call to the preceding C function:

```
EXTERNAL ISCAN
INTEGER(KIND=8) ISCAN
INTEGER(KIND=8) RET
INTEGER J,K
CHARACTER*(50) INP
INP = '1  -99   3141592  0xfff  033 '
J = 0
DO WHILE (J .LT. LEN(INP))
      K = J
      RET = ISCAN(INP,J)
      PRINT *, K,': ',RET,' -->',J
END DO
END
```

Using Fortran Common Blocks in C Code

A C function can refer to the contents of a common block defined in a Fortran program. The name of the block as given in the `COMMON` statement is altered as described in "Fortran Treatment of External and Public Names", page 66. (The name is converted to lowercase and extended with an underscore). The name of the blank common is `_BLNK__`, with one leading underscore and two trailing ones.

To refer to the contents of a common block, take these steps:

1. Declare a C structure with fields that have the appropriate data types to match the successive elements of the Fortran common block. For information on corresponding data types, see Table 7-1, page 68.

2. Declare the common block name as an external structure of that type.

The following example employs this method:

```
        INTEGER STKTOP, STKLEN, STACK(100)
        COMMON /WITHC/ STKTOP, STKLEN, STACK

struct fstack {
      int stktop, stklen;
      int stack[100];
}
extern fstack withc_;
int peektop_()
{
      if (withc_.stktop) /* stack not empty */
          return withc_.stack[withc_.stktop-1];
      else...
}
```

The restrictions on this capability are as follows:

- You cannot map a common block that contains Fortran pointer-based variables.
- If the common block contains a variable of Fortran derived type (a structure), ensure that the derived type is declared with the `SEQUENCE` attribute. Otherwise, its fields may not appear in the expected sequence in memory.
- When `-O3` is in effect, the compiler may split up common blocks. For information on the `-O3` option, see the `f90`(1) man page.

Using Fortran Arrays in C Code

A C program can access arrays created in Fortran. The following example illustrates this.

The following Fortran code fragment declares a matrix in a common block and then calls a C subroutine to modify the array:

```
INTEGER IMAT(10,100), R, C
COMMON /WITHC/ IMAT
R = 74
C = 6
CALL CSUB(C, R, 746)
```

```
PRINT *, IMAT(6,74)
END
```

The following C function stores its third argument in the common array using the subscripts passed in the first two arguments. In the C function, the order of the dimensions of the array are reversed, so the subscript values are reversed to match, and decremented by 1 to provide 0-origin indexing:

```
extern struct { int imat[100][10]; } withc_;
void csub_(int *pc, int *pr, int *pval)
{
      withc_.imat[*pr-1][*pc-1] = *pval;
}
```

Calls to C Using LOC and %VAL

You can use the nonstandard intrinsic functions %VAL and LOC to pass arguments in ways other than the standard Fortran conventions described in "How Fortran Passes Arguments", page 70.

Using %VAL

The %VAL function is used in an argument list to cause an argument to be passed by value rather than by reference. Suppose that you need to call a C function having the following prototype in file `ti.c`:

```
#includevoid takesint_(int i, char *s, int len)
{
   printf("i: %d\n", i);
   printf("s: %.*s\n", len, s);
}
```

The first argument to this function is an integer value, not the address of an integer value in memory. You could call this function from the following Fortran code in file `ti_f.f`:

```
      CHARACTER(80) SENTENCE
      INTEGER(4) J
      J = 13
      SENTENCE = "Hello, there."
      CALL TAKESINT(%VAL(J), SENTENCE)
      END
```

The use of %VAL(*j*) causes the contents of *j* to be passed, rather than the address of *j*.

```
% f90 -n32 ti_f.f ti.c
ti_f.f:
ti.c:
% ./a.out
i: 13
s: Hello, there.
```

Using LOC

The LOC function returns the address of its argument. The type of the argument is determined by hardware type. It can be used with %VAL to prevent passing the length of a character value as a hidden argument. In other words, the argument %VAL(LOC(*char_var*)) passes only the address of *char_var*. It does not pass the implicit length argument.

Calling Assembly Language from Fortran

You can write modules in MIPS assembly language, following the guidelines in the *MIPSpro Assembly Language Programmer's Guide*. Procedures in these modules can be called from Fortran. There is only one special consideration.

Operating in assembly language, you can change the operating mode and the rounding mode of the CPU. When running Fortran programs that contain quad precision operations, you must run the compiler in round-to-nearest mode. This mode is in effect by default, so you usually do not need to set it. When writing programs that call your own assembly routines, ensure that this mode is set. For more information, see the swapRM(3c) man page.

The Auto-Parallelizing Option (APO)

The Auto-Parallelizing Option (APO) enables the compiler to optimize parallel codes and enhances performance on multiprocessor systems. APO is controlled with command line options and source directives.

Note: APO is licensed and sold separately from the compiler. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your SGI sales representative.

APO is integrated into the compiler; it is not a source-to-source preprocessor. Although runtime performance suffers slightly on single-processor systems, parallelized programs can be created and debugged with APO enabled.

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the compiler analyzes and restructures the program with little or no intervention by you. With APO, the compiler automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques.

As the following figure shows, APO integrates automatic parallelization with other compiler optimizations, such as interprocedural analysis (IPA), optimizations for single processors, and loop nest optimization (LNO):

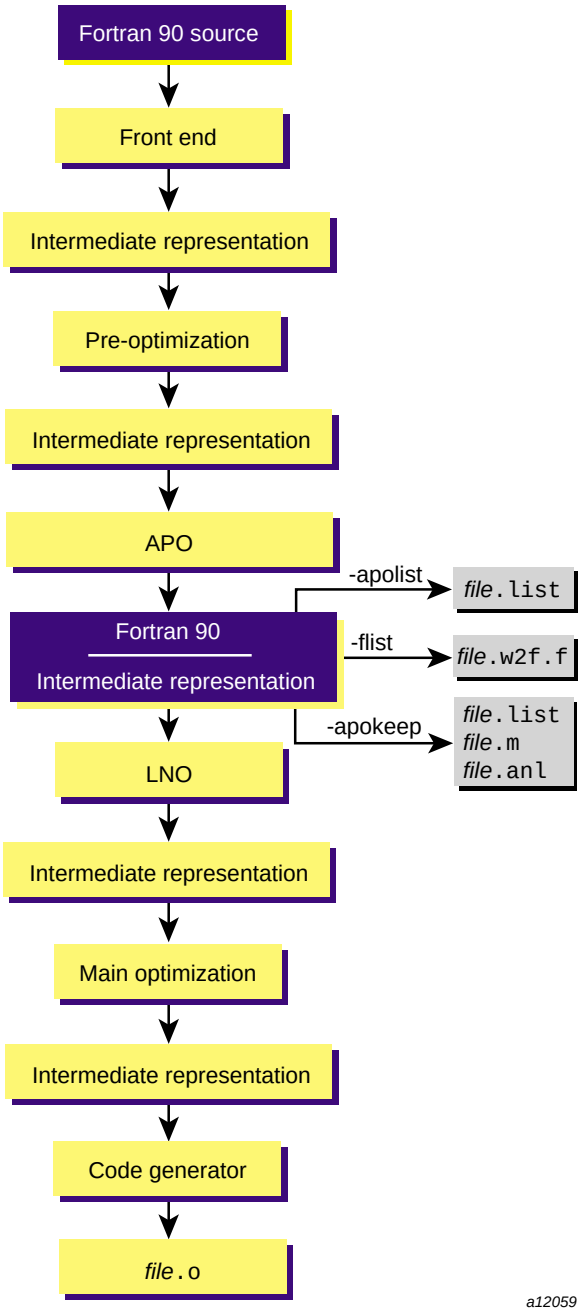

Figure 8-1 Files Generated by the ProDev Automatic Parallelization Option

`f90`(1) Command Line Options That Affect APO

Several `f90`(1) command line options control APO's effect on your program. The following command line, for example, invokes APO and requests aggressive optimization:

```
f90 -apo -O3 zebra.f
```

The following subsections describe the effects that various `f90`(1) command line options have on APO.

Note: If you invoke the linker separately, you must specify the `-apo` option on the `ld`(1) command line.

`-apo`

The `-apo` option invokes APO. When this option is enabled, the compiler automatically converts sequential code into parallel code by inserting parallel directives where it it safe and beneficial to do so. Specifying `-apo` also enables the `-mp` option, which enables recognition of the parallel directives inserted into your code.

`-apokeep` and `-apolist`

The `-apokeep` and `-apolist` options control output files. Both options generate *file*`.list`, which is a listing file that contains information on the loops that were executed in parallel and explains why others were not executed in parallel.

When `-apokeep` is specified, the compiler writes *file*`.list`, and in addition, it retains *file*`.anl` and *file*`.m`. The ProDev ProMP tools use *file*`.anl`. For more information on ProDev ProMP, see the *ProDev WorkShop: ProMP User's Guide*. *file*`.m` is an annotated version of your source code that shows the insertion of multiprocessing directives.

For more information on the content of *file*`.list`, *file*`.anl`, and *file*`.m`, see "The *file*`.w2f.f` File", page 87.

Note: Because of data conflicts, do not specify the `-mplist` or `-FLIST` options when `-apokeep` is specified.

-flist

This option generates a Fortran listing and directs the compiler to write the transformed source code and multiprocessing directives to *file*`.w2f.f`. For more information on the content of *file*`.w2f.f`, see "Files", page 86.

-IPA:...

Interprocedural analysis (IPA) is invoked by the `-IPA` command line option. It performs program optimizations that can only be done by examining the whole program rather than processing each procedure separately.

When APO is invoked with IPA, only those loops with calls determined to be safe are parallelized.

If IPA expands subroutines inline in a calling routine, the subroutines are compiled with the options of the calling routine. If the calling routine is not compiled with `-apo`, none of its inlined subroutines are parallelized. This is true even if the subroutines are compiled separately with `-apo` because with IPA, automatic parallelization is deferred until link time.

If `-apokeep` or `-pfakeep` are specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption. For that suboption, the default becomes `OFF`.

For more information on the effect of IPA, see "Loops Containing Function Calls", page 92. For more information on IPA itself, see the `ipa`(5) man page.

-LNO:...

The `-LNO` options control the Loop Nest Optimizer (LNO). LNO is enabled by default at `-O3`. LNO performs loop optimizations that better exploit caches and instruction-level parallelism. The following LNO options are of particular interest to APO users:

- `-LNO:auto_dist=on`. This option requests that APO insert data distribution directives to provide the best memory utilization on Origin2000 systems.
- `-LNO:ignore_pragmas=`*setting*. This option directs APO to ignore all of the directives and assertions described in "Compiler Directives", page 100.

- `-LNO:parallel_overhead=`*num_cycles*. This option allows you to override certain compiler assumptions regarding the efficiency to be gained by executing certain loops in parallel rather than serially. Specifically, changing this setting changes the default estimate of the cost to invoke a parallel loop in your runtime environment. This estimate varies depending on your particular runtime environment, but it is typically several thousand machine cycles.

-O3

To obtain maximum performace, specify `-O3` when compiling with APO enabled. The optimizations at this level maximize code quality even if they require extensive compile time or relax the language rules. In addition, LNO is enabled by default at this `-O` level.

The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in "`-OPT:...`", page 85.

-OPT:...

The `-OPT` command line option controls general optimizations that are not associated with a distinct compiler phase.

The `-OPT:roundoff=`*n* option controls floating-point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules.

When `-O3` is in effect, the default rounding setting is `-OPT:roundoff=2`. This setting allows transformations with extensive effects on floating-point results. It allows associative rearrangement across loop iterations and the distribution of multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round-off errors for a wide range of floating-point operands.

At `-OPT:roundoff=2` or `3`, APO can change the sequence of a loop's floating-point operations in order to parallelize it. Because floating-point operations have finite precision, this change can cause slightly different results. If you want to avoid these differences by not having such loops parallelized, you must compile with `-OPT:roundoff=0` or `-OPT:roundoff=1`.

Example 8-1 APO OPT example

APO parallelizes the following loop when compiled with the default settings of `-OPT:roundoff=2` and `-O3`:

```
REAL A, B(100)
DO I = 1, 100
    A = A + B(I)
END DO
```

At the start of the loop, each processor gets a private copy of `A` in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of `A` can be different from the value generated by a version of the loop that is not parallelized.

file

Your input file.

For information on files used and generated when APO is enabled, see "Files". For information on Fortran input files, see the `f90`(1) man page.

Files

APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. You can use these listings to identify constructs that inhibit parallelization. When you remove these constructs, you can often improve program performance dramatically.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the ProDev ProMP Parallel Analyzer View described in the *ProDev WorkShop: ProMP User's Guide*.

The following sections describe the content of the files generated by APO.

The *file*.list File

The -apolist and -apokeep options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is provided.

Example 8-2 *file*.list and APO

The following subroutine resides in file testl.f:

```
SUBROUTINE SUB(ARR, N)
REAL(KIND=8), DIMENSION(N) :: ARR
INTEGER :: N, I

ARR(2:N) = ARR(1:N-1) + ARR(2:N)

DO I = 1, N
  ARR(I) = ARR(I) + 7.0
  CALL FOO(A)
END DO

ARR = ARR + 7.0

END
```

The preceding code produces the following APO list file:

```
Parallelization Log for Subprogram sub_
5: Not Parallel
     Array dependence from ARR on line 5 to ARR on line 5.

7: Not Parallel
     Call foo_ on line 9.

12: PARALLEL (Auto) __mpdo_sub_1
```

The *file*.w2f.f File

The -flist option generates *file*.w2f.f. File *file*.w2f.f contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the

original code were not parallelized. You can use this information to change the original program.

The compiler creates *file*`.w2f.f` by invoking the appropriate translator to turn the compiler's internal representations into FORTRAN 77 (not Fortran 95). In most cases, the files contain valid code that can be recompiled, although compiling *file*`.w2f.f` without APO enabled does not produce object code that is exactly the same as that generated when APO is enabled on the original source.

By default, the parallelized program in *file*`.w2f.f` uses OpenMP directives. To generate a parallelized program that uses the outmoded MIPS multiprocessing directives specify `-FLIST:emit_omp=OFF`.

Example 8-3 APO and `.w2f.f` file

File `testw2.f` is compiled with the following command:

```
f90 -O3 -n32 -mips4 -c -apo -apokeep testw2.f
```

```
      SUBROUTINE INIT(A)
REAL(KIND=4), DIMENSION(10000) :: A

A = 0.0

END
```

Compiling `testw2.f` generates an object file, `testw2.o`, and listing file `testw2.w2f.f`, which contains the following code:

```
C ********************************
C Fortran file translated from WHIRL
C ********************************

CSGI$ start 1
        SUBROUTINE init(A)
        IMPLICIT NONE
        REAL(4) A(10000_8)
C
C       **** Temporary variables ****
C
        INTEGER(4) f90li_0_1
C
C       **** statements ****
C
```

```
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_init_1
CSGI$ start 2
C$OMP PARALLEL DO private(f90li_0_1), shared(A)
        DO f90li_0_1 = 0, 9999, 1
          A(f90li_0_1 + 1) = 0.0
        END DO
CSGI$ end 2
        RETURN
              END
CSGI$ end 1
```

Note: `WHIRL` is the name for the compiler's intermediate representation. It is written in the style of the FORTRAN 77 standard, not the Fortran 95 standard.

As explained in "The *file*`.list` File", page 87, parallel versions of loops are put in their own subroutines. In this example, that subroutine is `__mpdo_init_1`. `C$OMP PARALLEL DO` is an OpenMP directive that specifies a parallel region containing a single `DO` directive.

About the `.m` and `.anl` Files

The `-apokeep` option generates *file*`.list`. It also generates *file*`.m` and *file*`.anl`, which are used by Workshop ProMP.

file`.m` is similar to the *file*`.w2f.f` file; it is based on OpenMP and mimics the behavior of the program after automatic parallelization.

ProDev ProMP is a SGI product that provides a graphical interface to aid in both automatic and manual parallelization for Fortran. The ProDev ProMP Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. For more information, see the *ProDev WorkShop: ProMP User's Guide* and the *ProDev WorkShop: Performance Analyzer User's Guide*.

SpeedShop, another SGI product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop includes a set of commands and a number of libraries to support the commands. For more information, see the *SpeedShop User's Guide*.

Note: The code in *file*`.m` is written in the style of the FORTRAN 77 standard, not the Fortran 95 standard.

Running Your Program

You invoke a parallelized version of your program using the same command line as a sequential one. The same binary output file can be executed on various numbers of processors. The default is to have the run-time environment select the number of processors to use based on how many are available.

You can change the default behavior by setting the `OMP_NUM_THREADS` environment variable, which tells the system to use a particular number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

The `OMP_DYNAMIC` environment variable allows you to control whether the run-time environment should dynamically adjust the number of threads available for executing parallel regions to optimize use of system resources. The default value is `TRUE`. If `OMP_DYNAMIC` is set to `FALSE`, dynamic adjustment is disabled.

For more information on these and other environment variables, see the `pe_environ`(5) man page.

Troubleshooting Incomplete Optimizations

Some loops cannot be safely parallelized and others are written in ways that inhibit APO's efficiency. The following subsections describe the steps you can take to make APO more effective:

- "Constructs That Inhibit Parallelization", page 91, describes constructs that inhibit parallelization.
- "Constructs That Slow Down Parallelized Code", page 95, describes constructs that inhibit APO's effectiveness.

Constructs That Inhibit Parallelization

A program's performance can be severely constrained if APO cannot recognize that a loop is safe to parallelize. APO analyzes every loop in a program. If a loop does not appear safe, it does not parallelize that loop. The following sections describe constructs that can inhibit parallelization:

- "Loops Containing Data Dependencies", page 91, describes basic data dependencies.
- "Loops Containing Function Calls", page 92, describes function calls.
- "Loops Containing `GO TO` Statements", page 92, describes `GO TO` statements.
- "Loops Containing Problematic Array Constructs", page 93, describes problematic array subscripts.
- "Loops Containing Local Variables", page 94, describes conditionally assigned local variables.

In many instances, loops containing the previous constructs can be parallelized after minor changes. Reviewing the information generated in program *file*`.list`, described in "The *file*`.list` File", page 87, can show you if any of these constructs are in your code.

Loops Containing Data Dependencies

Generally, a loop is safe if there are no data dependencies, such as a variable being assigned in one iteration of a loop and used in another. APO does not parallelize loops for which it detects data dependencies.

For example, APO cannot parallelize loop `I` in the following subroutine because it contains a data dependence on variable `X`:

```
      SUBROUTINE SUB(N, A, B)
INTEGER :: I, N
REAL :: X, A(N), B(N)

X = 0.0
DO I = 1, N
  A(I) = X
  IF (I .GT. N / 2) X = 1.0
END DO
```

```
END
```

Many times, such dependences can be removed by making simple modifications to the source code. In this case, we can assign to X in each iteration before we read X, as follows:

```
SUBROUTINE SUB(N, A, B)
INTEGER :: I, N
REAL :: X, A(N), B(N)

DO I = 1, N
  IF (I .LE. N / 2) THEN
    X = 0.0
  ELSE
    X = 1.0
  END IF
  A(I) = X
END DO

END
```

APO now can parallelize loop I.

Loops Containing Function Calls

By default, APO does not parallelize a loop that contains a function call because the function in one iteration of the loop can modify or depend on data in other iterations.

You can, however, use interprocedural analysis (IPA) to provide APO with enough information to parallelize some loops containing subroutine calls by inlining those calls. IPA is specified by the -IPA command line option. For more information on IPA, see the ipa(5) man page.

You can also direct APO to ignore function call dependencies when analyzing the specified loops by using the !*$* ASSERT CONCURRENT CALL directive described in "!*$* ASSERT CONCURRENT CALL", page 101.

Loops Containing GO TO Statements

GO TO statements are unstructured control flows. APO converts most unstructured control flows in loops into structured flows that can be parallelized. However, GO TO statements in loops can still cause the following problems:

- Unstructured control flows. APO is unable to restructure all types of flow control in loops. You must either restructure these control flows or manually parallelize the loops containing them.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

For improved performance, remove `GO TO` statements from loops to be considered candidates for parallelization.

Loops Containing Problematic Array Constructs

The following array constructs inhibit parallelization and should be removed whenever APO is used:

- Arrays with subscripts that are indirect array references. APO cannot analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference `IB(I)` is equal to the same value for different iterations of `I`:

```
DO I = 1, N
    A(IB(I)) = ...
END DO
```

  If every element of array `IB` is unique, the loop can safely be made parallel. To achieve automatic parallelism in such cases, use the `!*$* ASSERT PERMUTATION` directive, discussed in "`!*$* ASSERT PERMUTATION (`*array_name*`)`", page 105.

- Arrays with unanalyzable subscripts. APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain the following elements:
  - Literal constants (`1`, `2`, `3`, …)
  - Variables (`I`, `J`, `K`, …)
  - The product of a literal constant and a variable, such as `N*5` or `K*32`
  - A sum or difference of any combination of the first three items, such as `N*21+K-251`

  In the following case, APO cannot analyze the division operator (`/`) in the array subscript and cannot reorder the loop:

```
DO I = 2, N, 2
    A(I/2) = ...
END DO
```

- Unknown information. In the following example there may be hidden knowledge about the relationship between variables `M` and `N`:

  ```
  DO I = 1, N
      A(I) = A(I+M)
  END DO
  ```

  The loop can be run in parallel if `M > N` because the array reference does not overlap. However, APO does not know the value of the variables and therefore cannot make the loop parallel. You can use the `!*$* ASSERT DO (CONCURRENT)` directive to have APO automatically parallelize this loop. For more information on this directive, see "`!*$* ASSERT DO (CONCURRENT)`", page 102.

Loops Containing Local Variables

When parallelizing a loop, APO often localizes (privatizes) temporary scalar and array variables by giving each processor its own nonshared copy of them. In the following example, array `TMP` is used for local scratch space:

```
DO I = 1, N
    DO J = 1, N
      TMP(J) = ...
    END DO
    DO J = 1, N
      A(J,I) = A(J,I) + TMP(J)
    END DO
END DO
```

To successfully parallelize the outer loop (`I`), APO must give each processor a distinct, private copy of array `TMP`. In this example, it is able to localize `TMP` and, thereby, to parallelize the loop.

APO cannot parallalize a loop when a conditionally assigned temporary variable might be used outside of the loop, as in the following example:

```
SUBROUTINE S1(A, B)
    COMMON T
    ...
```

```
    DO I = 1, N
      IF (B(I)) THEN
        T = ...
        A(I) = A(I) + T
      END IF
    END DO
    CALL S2()
END
```

If the loop were to be run in parallel, a problem would arise if the value of `T` were used inside subroutine `S2()` because it is not known which processor's private copy of `T` should be used by `S2()`. If `T` were not conditionally assigned, the processor that executed iteration `N` would be used. Because `T` is conditionally assigned, APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable `T` is localized. If the value of `T` is not used outside the loop, replace `T` with a local variable. Unless `T` is a local variable, APO assumes that `S2()` might use it.

Constructs That Slow Down Parallelized Code

APO parallelizes a loop by distributing its iterations among the available processors. Loop nesting, loops with low trip counts, and other program characteristics can affect the efficiency of APO. The following sections describe the effect that these and other programming constructs can have on APO's ability to parallelize:

- "Parallelizing Nested Loops", page 95, describes parallelizing nested loops.
- "Parallelizing Loops with Small or Indeterminate Trip Counts", page 97, describes parallelizing loops with small or indeterminate trip counts.
- "Parallelizing Loops with Poor Data Locality", page 98, describes parallelizing loops that exhibit poor data locality.

Parallelizing Nested Loops

APO can parallelize only one loop in a loop nest. In these cases, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from that fact that more processors end up processing larger sections of the program. This saves synchronization and other overhead costs.

Example 8-4 Parallelizing nested loops

Consider the following simple loop nest:

```
DO I = 1, L
    ...
    DO J = 1, M
      ...
      DO K = 1, N
        ...
```

When parallelizing nested loops `I`, `J`, and `K`, APO distributes only one of the loops. Effective loop nest parallelization depends on the loop that APO chooses, but it is possible for APO to choose an inferior loop to be parallelized. APO may attempt to interchange loops to make a more promising one the outermost. If the outermost loop attempt fails, APO attempts to parallelize an inner loop. Because of the potential for improved performance, it is useful for you to modify your code so that the outermost loop is the one parallelized.

"The *file*`.list` File", page 87, describes *file*`.list`. This output file contains information that tells you which loop in a nest was parallelized.

For every loop that could be parallelized, APO generates a test to determine whether the loop is being called from within either another parallel loop or from within a parallel region. In some cases, you can minimize the extra testing that APO must perform by inserting directives into your code to inhibit parallization testing. The following example demonstrates this:

Example 8-5 Parallelizing nested loops (continued)

```
SUBROUTINE CALLER
    DO I = 1, N
      CALL SUB
    END DO
    ...
END
SUBROUTINE SUB
    ...
    DO I = 1, N
      ...
    END DO
END
```

If the loop inside CALLER() is parallelized, the loop inside SUB() cannot be run in parallel when CALLER() calls SUB(). In this case, the test can be avoided.

If SUB() is always called from CALLER(), you can use the !*$* ASSERT DO (SERIAL) directive to force the sequential execution of the loop in SUB(). With the addition of the directive, the subroutine would be written as follows:

```
      SUBROUTINE CALLER
          DO I = 1, N
            CALL SUB
          END DO
          ...
      END
      SUBROUTINE SUB
          ...
!*$* ASSERT DO (SERIAL)
          DO I = 1, N
            ...
          END DO
      END
```

For more information on this compiler directive, see "!*$* ASSERT DO (SERIAL)", page 103.

Parallelizing Loops with Small or Indeterminate Trip Counts

The *trip count* is the number of times a loop is executed. Loops with large trip counts are the best candidates for parallelization. The following paragraphs show how to modify your program if your program contains loops with small trip counts or loops with indeterminate trip counts:

- Loops with small trip counts generally run faster when they are not parallelized. Consider the following loop nest:

  ```
  DO I = 1, M
      DO J = 1, N
  ```

  APO may try to parallelize loop I because it is outermost. If M is very small, it would be better to interchange the loops and make loop J outermost before parallelization. Because APO often cannot know that M is small, you can use a !*$* ASSERT DO PREFER (CONCURRENT) directive to indicate to APO that it is better to parallelize loop J, as follows:

```
      DO I = 1, M
!*$* ASSERT DO PREFER (CONCURRENT)
          DO J = 1, N
```

- If the trip count is not known (and sometimes even if it is), APO parallelizes the loop conditionally, generating code for both a parallel and a sequential version. By generating two versions, APO can avoid running a loop in parallel that may have small trip count. APO chooses the version to use based on the trip count, the code inside the loop's body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that runtime environment.

  You can avoid the overhead incurred by having APO generate both sequential and parallel versions of a loop by using the `!*$* ASSERT DO PREFER (SERIAL)` directive.

Parallelizing Loops with Poor Data Locality

Computer memory has a hierarchical organization. Higher up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is further down in the hierarchy. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor's cache, a program's performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*, which means that if a memory location is referenced, it is probable that it or a nearby location will be referenced in the near future. Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

The following examples show the effect of locality on parallelization. Assume that the loops are to be parallelized and that there are *p* processors.

Example 8-6 Distribution of iterations

```
DO I = 1, N
    ...A(I)
END DO
DO I = N, 1, -1
    ...A(I)...
END DO
```

In the first loop, the first processor accesses the first N/*p* elements of A; the second processor accesses the next N/*p* elements; and so on. In the second loop, the distribution of iterations is reversed. That is, the first processor accesses the last N/*p* elements of A, and so on. Most elements are not in the cache of the processor needing them during the second loop. This code fragment would run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 8-7 Two nests in sequence

```
DO I = 1, N
    DO J = 1, N
      A(I,J) = B(J,I) + ...
    END DO
END DO

DO I = 1, N
    DO J = 1, N
      B(I,J) = A(J,I) + ...
    END DO
END DO
```

In Example 8-7, page 99, APO may parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses the first N/*p* rows of A and the first N/*p* columns of B. In the second nest, the first processor accesses the first N/*p* columns of A and the first N/*p* rows of B. This example runs much more efficiently if you parallelize the I loop in one nest and the J loop in the other. You can instruct APO to do this with the !*$* ASSERT DO PREFER (CONCURRENT) directive, as follows:

```
      DO I = 1, N
!*$* ASSERT DO PREFER (CONCURRENT)
          DO J = 1, N
            A(I,J) = B(J,I) + ...
          END DO
      END DO

!*$* ASSERT DO PREFER (CONCURRENT)
      DO I = 1, N
          DO J = 1, N
            B(I,J) = A(J,I) + ...
          END DO
      END DO
```

Compiler Directives

APO works in conjunction with the OpenMP Fortran API directives. You can use these directives to manually parallelize some loop nests, while leaving others to APO. This approach has the following positive and negative aspects:

- As a positive aspect, the OpenMP and Origin series directives are well defined and deterministic. If you use a directive, the specified loop is run in parallel. This assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.
- The negative side to this is that you must carefully analyze the code to determine that parallelism is safe. Also, you must mark all private variables.

In addition to the OpenMP and Origin series directives, you can also use the APO-specific directives described in this section. These directives give APO more information about your code.

Note: APO also recognizes outmoded SGI multiprocessing directives. The OpenMP directive set is the preferred directive set for multiprocessing. You must include the `-mp` option on the `f90`(1) command line in order for the compiler to recognize the SGI multiprocessing directives.

The APO directives can affect certain optimizations, such as loop interchange, during the compiling process. To direct the compiler to disregard any of the preceding directives, use the `-x`*dirlist* option described in the `f90`(1) man page.

The APO directives are as follows:

- `!*$* ASSERT CONCURRENT CALL`. This directive directs APO to ignore dependencies in subroutine calls that would inhibit parallelization. For more information on this directive, see "`!*$* ASSERT CONCURRENT CALL`", page 101.
- `!*$* ASSERT DO (CONCURRENT)`. This directive asserts that APO should not let perceived dependencies between two references to the same array inhibit parallelizing. For more information on this directive, see "`!*$* ASSERT DO (CONCURRENT)`", page 102.
- `!*$* ASSERT DO (SERIAL)`. This directive requests that the following loop be executed in serial mode. For more information on this directive, see "`!*$* ASSERT DO (SERIAL)`", page 103.

- `!*$* ASSERT DO PREFER (CONCURRENT)`. This directive parallelizes the following loop if it is safe. For more information on this directive, see "`!*$* ASSERT DO PREFER (CONCURRENT)`", page 104.
- `!*$* ASSERT PERMUTATION (`*array_name*`)`. This directive asserts that array *array_name* is a permutation array. For more information on this directive, see "`!*$* ASSERT PERMUTATION (`*array_name*`)`", page 105.
- `!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE`. The `!*$* NO CONCURRENTIZE` directive inhibits either parallelization of all loops in a subroutine or parallelization of all loops in a file. The `!*$* CONCURRENTIZE` directive overrides the `!*$* NO CONCURRENTIZE` directive, and its effect varies with its placement. For more information on these directives, see "`!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE`", page 106.

Note: The compiler honors the following APO directives even if the `-apo` option is not included on your command line:

- `!*$* ASSERT CONCURRENT CALL`
- `!*$* ASSERT DO (CONCURRENT)`
- `!*$* ASSERT PERMUTATION (`*array_name*`)`

!*$* ASSERT CONCURRENT CALL

The `!*$* ASSERT CONCURRENT CALL` directive instructs APO to ignore the dependencies of subroutine and function calls contained in the loop that follows the assertion. The directive applies to the loop that immediately follows it and to all loops nested inside that loop.

Note: The directive affects the compilation even when `-apo` is not specified.

APO ignores the dependencies in subroutine `FRED()` when it analyzes the following loop:

```
!*$* ASSERT CONCURRENT CALL
         DO I = 1, N
           CALL FRED
           ...
         END DO
```

```
SUBROUTINE FRED
  ...
END
```

To prevent incorrect parallelization, make sure the following conditions are met when using `!*$* ASSERT CONCURRENT CALL`:

- A subroutine inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.
- A subroutine inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.

Example 8-8 `ASSERT CONCURRENT` (illegal use)

The following code shows an illegal use of the directive. Subroutine `FRED()` writes to variable `T`, which is also read from by `WILMA()` during other iterations:

```
!*$* ASSERT CONCURRENT CALL
         DO I = 1,M
           CALL FRED(B, I, T)
           CALL WILMA(A, I, T)
         END DO
         SUBROUTINE FRED(B, I, T)
           REAL B(*)
           T = B(I)
         END
         SUBROUTINE WILMA(A, I, T)
           REAL A(*)
           A(I) = T
         END
```

By localizing the variable `T`, you can manually parallelize the preceding example safely. However, APO does not know to localize `T`, so it illegally parallelizes the loop because of the directive.

!*$* ASSERT DO (CONCURRENT)

The `!*$* ASSERT DO (CONCURRENT)` directive instructs APO, when analyzing the loop immediately following this directive, to ignore possible dependencies between

two references to the same array. If there are real dependencies between array references, the `!*$* ASSERT DO (CONCURRENT)` directive can cause APO to generate incorrect code.

Note: This directive affects the compilation even when `-apo` is not specified.

The following example shows correct use of this directive when `M > N`:

```
!*$* ASSERT DO (CONCURRENT)
         DO I = 1, N
           A(I) = A(I+M)
```

Be aware of the following points when using this directive:

- If multiple loops in a nest can be parallelized, `!*$* ASSERT DO (CONCURRENT)` causes APO to parallelize the loop immediately following the assertion.
- Applying this directive to an inner loop can cause the loop to be made outermost by APO's loop interchange operations.
- This directive does not affect how APO analyzes `CALL` statements. For more information on APO's interaction with `CALL` statements, see "`!*$* ASSERT CONCURRENT CALL`", page 101.
- This directive does not affect how APO analyzes dependencies between two potentially aliased pointers.
- The compiler may find some obvious real dependencies. If it does so, it ignores this directive.

!*$* ASSERT DO (SERIAL)

The `!*$* ASSERT DO (SERIAL)` directive instructs APO not to parallelize the loop following the assertion; the loop is executed in serial mode. APO can, however, parallelize another loop in the same nest. The parallelized loop can be either inside or outside the designated sequential loop.

Note: This directive has the same effect as the `!*$* ASSERT DO PREFER (SERIAL)` directive. In order for the `!*$* ASSERT DO PREFER (SERIAL)` directive to be honored, however, the `-apo` option must appear on the `f90`(1) command line. The `!*$* ASSERT DO PREFER (SERIAL)` directive is outmoded.

The `!*$* ASSERT DO (SERIAL)` directive affects the compilation even when the `-apo` option is not specified.

Example 8-9 ASSERT DO

The following code fragment contains a directive that requests that loop `J` be run serially:

```
         DO I = 1, M
!*$* ASSERT DO (SERIAL)
           DO J = 1, N
             A(I,J) = B(I,J)
           END DO
           ...
         END DO
```

The directive applies only to the loop that immediately follows it. For example, APO still tries to parallelize loop `I`. This directive is useful in cases like this when the value of `N` is known to be very small.

!*$* ASSERT DO PREFER (CONCURRENT)

The `!*$* ASSERT DO PREFER (CONCURRENT)` directive instructs APO to parallelize the loop immediately following the directive if it is safe to do so.

Example 8-10 ASSERT DO PREFER

The following code fragment encourages APO to run loop `I` loop in parallel:

```
!*$* ASSERT DO PREFER (CONCURRENT)
         DO I = 1, M
           DO J = 1, N
             A(I,J) = B(I,J)
           END DO
           ...
```

```
      END DO
```

When dealing with nested loops, APO follows these guidelines:

- If the loop specified by the `!*$* ASSERT DO PREFER (CONCURRENT)` directive is safe to parallelize, APO parallelizes the specified loop even if other loops in the nest are safe.
- If the specified loop is not safe to parallelize, APO parallelizes a different loop that is safe.
- If this directive is applied to an inner loop, APO can interchange the loop and make the specified loop the outermost loop.
- If this directive is applied to more than one loop in a nest, APO parallelizes one of the specified loops.

!*$* ASSERT PERMUTATION (*array_name*)

When placed inside a subroutine, the `!*$* ASSERT PERMUTATION (`*array_name*`)` directive informs APO that *array_name* is a permutation array. A *permutation array* is one in which every element of the array has a distinct value.

This directive does not require the permutation array to be *dense*. That is, within the array, every `IB(I)` must have a distinct value, but there can be gaps between the values, such as `IB(1)` = 1, `IB(2)` = 4, `IB(3)` = 9, and so on.

Note: This directive affects compilation even when `-apo` is not specified.

Example 8-11 `ASSERT PERMUTATION`

In the following code fragment, array `IB` is declared to be a permutation array for both loops in `SUB1()`:

```
      SUBROUTINE SUB1
        DO I = 1, N
          A(IB(I)) = ...
        END DO
!*$* ASSERT PERMUTATION (IB)
        DO I = 1, N
          A(IB(I)) = ...
        END DO
```

```
      END
```

Note the following points about this directive:

- As shown in the example, you can use this directive to parallelize loops that use arrays for indirect addressing. Without this directive, APO cannot determine that the array elements used as indexes are distinct.
- `!*$* ASSERT PERMUTATION (`*array_name*`)` affects every loop in a subroutine, even those that appear ahead of it.

!*$* NO CONCURRENTIZE and !*$* CONCURRENTIZE

The `!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE` directives toggle parallelization. Their effects depend on their placement.

- When placed inside subroutines and functions, `!*$* NO CONCURRENTIZE` inhibits parallelization. In the following example, no loops inside `SUB1()` are parallelized:

```
        SUBROUTINE SUB1
!*$* NO CONCURRENTIZE
          ...
        END
```

- When placed outside of a subroutine, `!*$* NO CONCURRENTIZE` prevents the parallelization of all procedures in the file, even those that appear ahead of it in the file. Loops inside subroutines `SUB2()` and `SUB3()` are not parallelized in the following example:

```
        SUBROUTINE SUB2
          ...
        END
!*$* NO CONCURRENTIZE
        SUBROUTINE SUB3
          ...
        END
```

The `!*$* CONCURRENTIZE` directive, when placed inside a subroutine, overrides a `!*$* NO CONCURRENTIZE` directive that is placed outside of it. Thus, this directive allows you to selectively parallelize subroutines in a file that has been made sequential with a `!*$* NO CONCURRENTIZE` directive.

Libraries

The compiler works with the following other commands, intrinsic procedures, and library routines:

- The `assign`(1) command. This command can be used to alter the details of a Fortran file connection, such as device residency, alternative file names, or file space allocations. The `assign`(1) options are associated with file names, file name patterns, or unit numbers. When associated with file names or file name patterns, the options are applied whenever a matching file name is opened from a Fortran program. When associated with a unit number, the options are applied whenever that unit becomes connected.

  For complete details about the `assign` command, see the `assign`(1) man page or the *MIPSpro Fortran 90 Programmer's I/O Guide*.

- The Flexible File I/O (FFIO) system. This system lets you specify a comma-separated list of layers through which I/O data will be passed. The FFIO layers act as filters that manipulate the data file as it is being read or written. The layers include performance options and the capability to read and write files in different vendors' blocking formats. For more information on FFIO, see the `intro_ffio`(3f) man page and the *MIPSpro Fortran 90 Programmer's I/O Guide*.

- Intrinsic procedures. These procedures are predefined by the computer programming language. They are invoked in the same way that other procedures are invoked.

- POSIX library routines. The POSIX FORTRAN 77 Language Interfaces Standard IEEE Std 1003.9-1992 (POSIX.a) defines a standardized interface for accessing the system services of IEEE Std 1003.1-1990 (POSIX.1) and supports routines to access constructs not directly accessible with FORTRAN 77. These routines can also be used by Fortran 90/95 programs. For more information on these routines, see the `intro_pxf`(3f) man page.

- Miscellaneous library routines. A library is a collection of subprograms, usually grouped around a specific subject, such as input and output (I/O). You can call library routines explicitly in your program, or they can be called by the compiler. The following sections describe the library routines that are available to you.

Miscellaneous Library Routines

The following list describes the library routines that are available with the MIPSpro Fortran 90 compiler. See the individual man pages for more details.

- FFIO routines (C routines used with the FFIO layers):
  - `fffcntl`(3c)
  - `ffopen`(3c)
  - `ffpos`(3c)
  - `ffread`(3c)
  - `ffseek`(3c)
- Interface routines (job control routines that control program terminations or execute a shell command):
  - `ABORT`(3f)
  - `EXIT`(3f)
  - `ISHELL`(3f)
- I/O routines to control input and output:
  - `ASNCTL`(3f)
  - `ASNQFILE`(3f)
  - `ASSIGN`(3f)
  - `FLUSH`(3f)
  - `NUMBLKS`(3f)
  - `RNL`(3f)
  - `RNLECHO`(3f)
  - `RNLSKIP`(3f)
  - `RNLTYPE`(3f)
  - `WNL`(3f)
  - `WNLLINE`(3f)

  - WNLLONG(3f)

- Programming aids (routines for times and dates, packing and unpacking, and character argument counters):
  - SECOND(3f)
  - SECONDR(3f)
  - SYSCLOCK(3f)
  - TIMEF(3f)
- Multiprocessing routines for Fortran. There are a suite of routines developed specifically for multiprocessing. For information on these routines, see the mp(3f) man page.

Library Functions

The Fortran library routines provide an interface from Fortran programs to the IRIX system functions. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code loaded with your program.

summarizes the routines in the Fortran run-time library that can be used with the compiler. The table indicates PXF POSIX Library standard routines as recommended substitutions for IRIX system functions. See the individual man pages for details about each routine.

Table A-1 Summary of System Interface Library Routines

Function	Recommended	Purpose
abort		Abnormal termination
access	PXFACCESS	Determine accessibility of a file
acct		Enable/disable process accounting
alarm	PXFALARM	Execute a subroutine after a specified time
barrier		Perform barrier operations

Function	Recommended	Purpose
blockproc		Block processes
brk		Change data segment space allocation
chdir	PXFCHDIR	Change default directory
chmod	PXFCHMOD	Change mode of a file
chown	PXFCHOWN	Change owner
chroot	PXFCHROOT	Change root directory for a command
close		Close a file descriptor
creat	PXFCREAT	Create or rewrite a file
ctime		Return system time
dtime		Return elapsed execution time
dup		Duplicate an open file descriptor
etime		Return elapsed execution time
exit	PXFFASTEXIT	Terminate process with status
fcntl		File control
fdate		Return date and time in an ASCII string
fgetc		Get a character from a logical unit
fork	PXFFORK	Create a copy of this process
fputc		Write a character to a Fortran logical unit
free_barrier		Free barrier
fseek		Reposition a file on a logical unit
fseek64		Reposition a file on a logical unit for 64-bit architecture
fstat		Get file status
ftell		Reposition a file on a logical unit
ftell64		Reposition a file on a logical unit for 64-bit architecture
gerror		Get system error messages
getarg	PXFGETARG	Return command line arguments
getc		Get a character from a logical unit

Function	Recommended	Purpose
getcwd	PXFGETCWD	Get pathname of current working directory
getdents		Read directory entries
getegid	PXFGETEGID	Get effective group ID
gethostid		Get unique identifier of current host
getenv	PXFGETENV	Get value of environment variables
geteuid	PXFGETEUID	Get effective user ID
getgid	PXFGETGID	Get user or group ID of the caller
gethostname		Get current host ID
getlog		Get user's login name
getpgrp	PXFGETPGRP	Get process group ID
getpid	PXFGETPID	Get process ID
getppid	PXFGETPPID	Get parent process ID
getsockopt		Get options on sockets
getuid	PXFGETUID	Get user or group ID of caller
gmtime		Return system time
iargc	IPXFARGC	Return command line arguments
idate		Return date or time in numerical form
ierrno		Get system error messages
ioctl		Control device
isatty	PXFISATTY	Determine if unit is associated with tty
itime		Return date or time in numerical form
kill	PXFKILL	Send a signal to a process
link	PXFLINK	Make a link to an existing file
loc		Return the address of an object
lseek		Move read/write file pointer
lseek64		Move read/write file pointer for 64-bit architecture
lstat		Get file status

Function	Recommended	Purpose
ltime		Return system time
m_fork		Create parallel processes
m_get_myid		Get task ID
m_get_numprocs		Get number of subtasks
m_kill_procs		Kill process
m_lock		Set global lock
m_next		Return value of counter
m_park_procs		Suspend child processes
m_rele_procs		Resume child processes
m_set_procs		Set number of subtasks
m_sync		Synchronize all threads
m_unlock		Unset a global lock
mkdir		Make a directory
mknod		Make a directory/file
mount		Mount a filesystem
new_barrier		Initialize a barrier structure
nice		Lower priority of a process
open	PXFOPEN	Open a file
oserror		Get/set system error
pause	PXFPAUSE	Suspend process until signal
perror		Get system error messages
pipe		Create an interprocess channel
plock		Lock process, test, or data in memory
prctl		Control processes
profil		Execution-time profile
ptrace		Process trace
putc		Write a character to a Fortran logical unit

Function	Recommended	Purpose
putenv		Set environment variable
qsort		Quick sort
read		Read from a file descriptor
readlink		Read value of symbolic link
rename	PXFRENAME	Change the name of a file
rmdir	PXFRMDIR	Remove a directory
sbrk		Change data segment space allocation
schedctl		Call to scheduler control
send		Send a message to a socket
setblockproccnt		Set semaphore count
setgid	PXFSETGID	Set group ID
sethostid		Set current host ID
setoserror		Set system error
setpgrp	PXFSETPGRP	Set process group ID
setsockopt		Set options on sockets
setuid	PXFSETUID	Set user ID
sginap		Put process to sleep
sginap64		Put process to sleep in 64-bit environment
shmat		Attach shared memory
shmdt		Detach shared memory
sighold		Raise priority and hold signal
sigignore		Ignore signal
signal		Change the action for a signal
sigpause		Suspend until receive signal
sigrelse		Release signal and lower priority
sigset		Specify system signal handling
sleep	PXFSLEEP	Suspend execution for an interval

Function	Recommended	Purpose
socket		Create an endpoint for communication TCP
sproc		Create a new share group process
stat	PXFSTAT	Get file status
stime		Set time
symlink		Make symbolic link
sync		Update superblock
sysmp		Control multiprocessing
sysmp64		Control multiprocessing in 64-bit environment
system		Issue a shell command
taskblock		Block tasks
taskcreate		Create a new task
taskctl		Control task
taskdestroy		Kill task
tasksetblockcnt		Set task semaphore count
taskunblock		Unblock task
time	PXFTIME	Return system time (must be declared EXTERNAL)
ttynam		Find name of terminal port
uadmin		Administrative control
ulimit		Get and set user limits
ulimit64		Get and set user limits in 64-bit architecture
umask	PXFUMASK	Get and set file creation mask
umount		Dismount a file system
unblockproc		Unblock processes
unlink	PXFUNLINK	Remove a directory entry
uscalloc		Shared memory allocator
uscalloc64		Shared memory allocator in 64-bit environment
uscas		Compare and swap operator

Function	Recommended	Purpose
usclosepollsema		Detach file descriptor from a pollable semaphore
usconfig		Semaphore and lock configuration operations
uscpsema		Acquire a semaphore
uscsetlock		Unconditionally set lock
usctlsema		Semaphore control operations
usdumplock		Dump lock information
usdumpsema		Dump semaphore information
usfree		User shared memory allocation
usfreelock		Free a lock
usfreepollsema		Free a pollable semaphore
usfreesema		Free a semaphore
usgetinfo		Exchange information through an arena
usinit		Semaphore and lock initialize routine
usinitlock		Initialize a lock
usinitsema		Initialize a semaphore
usmalloc		Allocate shared memory
usmalloc64		Allocate shared memory in 64-bit environment
usmallopt		Control allocation algorithm
usnewlock		Allocate and initialize a lock
usnewpollsema		Allocate and initialize a pollable semaphore
usnewsema		Allocate and initialize a semaphore
usopenpollsema		Attach a file descriptor to a pollable semaphore
uspsema		Acquire a semaphore
usputinfo		Exchange information through an arena
usrealloc		User share memory allocation
usrealloc64		User share memory allocation in 64-bit environment
ussetlock		Set lock

Function	Recommended	Purpose
ustestlock		Test lock
ustestsema		Return value of semaphore
usunsetlock		Unset lock
usvsema		Free a resource to a semaphore
uswsetlock		Set lock
wait	PXFWAIT	Wait for a process to terminate
write		Write to a file

Compatibility with sproc(2)

The parallelism used in Fortran is implemented using the sproc(2) system call. It is recommended that programs not attempt to use both !$OMP PARALLEL DO loops and sproc calls. It is possible, but there are several restrictions:

- Any threads you create may not execute !$OMP PARALLEL DO loops; only the original thread is allowed to do this.
- The calls to routines like mp_block and mp_destroy apply only to the threads created by mp_create or to those automatically created when the Fortran job starts; they have no effect on any user-defined threads.
- Calls to routines such as m_get_numprocs do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the kill routine with the arguments 0 and sig (for example, kill(0,sig)) to signal all members of the process group might kill threads used to execute !$OMP PARALLEL DO. If you choose to intercept the SIGCLD signal, you must be prepared to receive this signal when the threads used for the !$OMP PARALLEL DO loops exit; this occurs when mp_destroy is called or at program termination.
- The m_fork call is implemented using sproc(2), so it is not legal to run m_fork on a family of processes that each subsequently executes !$OMP PARALLEL DO loops. Only the original thread can execute !$OMP PARALLEL DO loops.

Index

D

E

F

K

L

M

N

O

P

R

S