HP Fortran Programmer's Guide

Third Edition

Product Number: B3909DB
Fortran Compiler for HP-UX
Document Number: B3909-90009
September 2001

Edition: Third

Document Number: B3909-90009
Remarks: Released September 2001. Fixed documentation defects.

Edition: Second

Document Number: B3909-90005
Remarks: Released June 2001. Added new PA-RISC and Itanium options.

Edition: First

Document Number: B3909-90002
Remarks: Released October 1998. Initial release.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Itanium is a trademark of the Intel Corporation.

Parts of the Itanium compiler were generated by the iburg code-generator generator, described at http://www.cs.princeton.edu/software/iburg.

Contents

Figures

Tables

Preface

The *HP Fortran Programmer's Guide* describes how to use different features of HP Fortran to develop, compile, debug, and optimize programs in the HP-UX PA-RISC and Itanium™-based operating systems. It also describes how to migrate HP FORTRAN 77 programs to the current HP Fortran compiler and how to use the different compiler features for porting programs written for other vendors' Fortran to HP Fortran.

If you have any problems with the software, please contact your local Hewlett-Packard Sales Office or Customer Service Center.

You need not be familiar with the HP parallel architecture, programming models, or optimization concepts to understand the concepts introduced in this book.

New in HP Fortran

The most recent release of HP Fortran v2.5.1 for the PA-RISC system includes the following benefits and features:

- Full Fortran 95 compiler (based on International ANSI/ISO standards) for the PA-RISC system
- Full OpenMP implementation (described in the *Parallel Programming Guide for HP-UX Systems*, part number B3909-90008)
- Object-oriented Fortran feature optimizations
- Support for math intrinsic inlining

The HP Fortran features described in this guide are upgrades from the previous versions of HP Fortran

HP Fortran v2.5, previously introduced, is a port of the HP-UX PA-RISC Fortran product to the Itanium-based system. It is source compatible between PA-RISC and Itanium. However, Itanium Fortran will not run on PA-RISC based systems.

Scope

This guide covers programming methods for the HP Fortran compiler on machines running:

- HP-UX 11.0 and higher (PA-RISC)
- HP-UX 11i Version 1.5 (Itanium)

HP Fortran supports an extensive shared-memory programming model. HP-UX 11.0 and higher includes the required assembler, linker, and libraries.

HP Fortran fully supports the international Fortran standards informally called Fortran 90 and Fortran 95 as defined by these two standards: *ISO/IEC 1539:1991(E)* and *ISO/IEC 1539:1997(E)*.

Notational conventions

This section discusses notational conventions used in this book.

`bold monospace`	In command examples, **`bold monospace`** identifies input that must be typed exactly as shown.
`monospace`	In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages.
italic	In paragraph text, *italic* identifies titles of documents. In command syntax diagrams, *italic* identifies variables that you must provide. The following command example uses brackets to indicate that the variable *output_file* is optional: `command` *input_file* [*output_file*]
Brackets ([ ])	In command examples, square brackets designate optional entries.
Curly brackets ({}), Pipe (|)	In command syntax diagrams, text surrounded by curly brackets indicates a choice. The choices available are shown inside the curly brackets and separated by the pipe sign (|). The following command example indicates that you can enter either `a` or `b`: `command {a | b}`

Horizontal ellipses (...)	In command examples, horizontal ellipses show repetition of the preceding items.
Vertical ellipses	Vertical ellipses show that lines of code have been left out of an example.
Keycap	**Keycap** indicates the keyboard keys you must press to execute the command example.

The directives and pragmas described in this book can be used with the HP Fortran and C compilers, unless otherwise noted. The aC++ compiler does not support the pragmas, but does support the memory classes. In general discussion, these directives and pragmas are presented in lowercase type, but each compiler recognizes them regardless of their case.

References to man pages appear in the form mnpgname(1), where "mnpgname" is the name of the man page and is followed by its section number enclosed in parentheses. To view this man page, type:

```
% man 1 mnpgname
```

NOTE A Note highlights important supplemental information.

Command syntax

Consider this example:

`COMMAND` *input_file* `[...] {a | b}` [*output_file*]

- `COMMAND` must be typed as it appears.
- *input_file* indicates a file name that must be supplied by the user.
- The horizontal ellipsis in brackets indicates that additional, optional input file names may be supplied.
- Either `a` or `b` must be supplied.
- [*output_file*] indicates an optional file name.

Associated documents

The following documents are listed as additional resources to help you use the compilers and associated tools:

- *HP aC++ Online Programmer's Guide*—Presents reference and tutorial information on aC++. This manual is only available in html format.
- *HP Fortran Programmer's Reference* — Provides language reference for HP Fortran and describes the language features and requirements.
- *HP C/HP-UX Reference Manual*—Presents reference information on the C programming language, as implemented by HP.
- *CXperf for the Itanium Processor Family User's Guide*—Provides conceptual, reference information and metric set selections for profiling Itanium-based products using the CXperf performance analyzer.
- *CXperf Command Reference for PA-RISC*—Provides introductory and reference information for using the CXperf performance analyzer for PA-RISC based products.
- *CXperf User's Guide for PA-RISC*—Provides conceptual, reference information and metric set selections for profiling PA-RISC based products using the CXperf performance analyzer.
- *HP-UX Floating Point Guide*—Describes how floating-point arithmetic is implemented on HP 9000 Series 700/800 systems. It discusses how floating-point behavior affects the programmer. Additional useful includes that which assists the programmer in writing or porting floating-point intensive programs.
- *HP MLIB User's Guide VECLIB and LAPACK*—Provides usage information about mathematical software and computational kernels for engineering and scientific applications.
- *HP MPI User's Guide*—Discusses message-passing programming using HP's Message-Passing Interface library.

- *HP-UX Linker and Libraries User's Guide*—Describes how to develop software on HP-UX, using the HP compilers, assemblers, linker, libraries, and object files.
- *Parallel Programming Guide for HP-UX Systems*—Describes efficient methods for shared-memory programming using the HP-UX suite of compilers: HP Fortran, HP aC++ (ANSI C++), and HP C. This guide is intended for use by experienced Fortran, C, and C++ programmers and is intended for use on HP-UX 11.0 and higher.
- *Programming with Threads on HP-UX*—Discusses programming with POSIX threads.
- *Threadtime* by Scott J. Norton and Mark D. DiPasquale—Provides detailed guidelines on the basics of thread management, including POSIX thread structure; thread management functions; and the creation, termination and synchronization of threads.

1 An overview of HP Fortran

When you use the `f90` command to compile a Fortran program, the command invokes a number of components—and not just the compiler—to create the executable. By default, `f90` invokes different components to parse the source files for syntax errors, produce an intermediate code, optimize the intermediate code, produce object code, search a set of library files for any additional object code that may be required, and link all of the object code into an executable file that you run without further processing.

For example, consider a program that consists of three source files: `x.f90`, `y.f90`, and `z.f90`. The following command line will process the source files and, if they are syntactically correct, produce an executable file with the default name `a.out`:

```
$ f90 x.f90 y.f90 z.f90
```

After compilation is complete, you can execute the program by invoking the name of the executable, as follows:

```
$ a.out
```

However, it is likely that you'll want to control what components act on your program and what they do to it. For example, you may want to give the executable a name other than `a.out` or to link in other libraries than the default ones. The HP Fortran compiler supports a variety of command-line options that enable you to control the compilation process. This chapter provides an overview of the process and of the options that enable you to control the different components invoked by the `f90` command.

NOTE To get a summary listing of all `f90` options, refer to the *f90*(1) man page or use the command, as shown here:

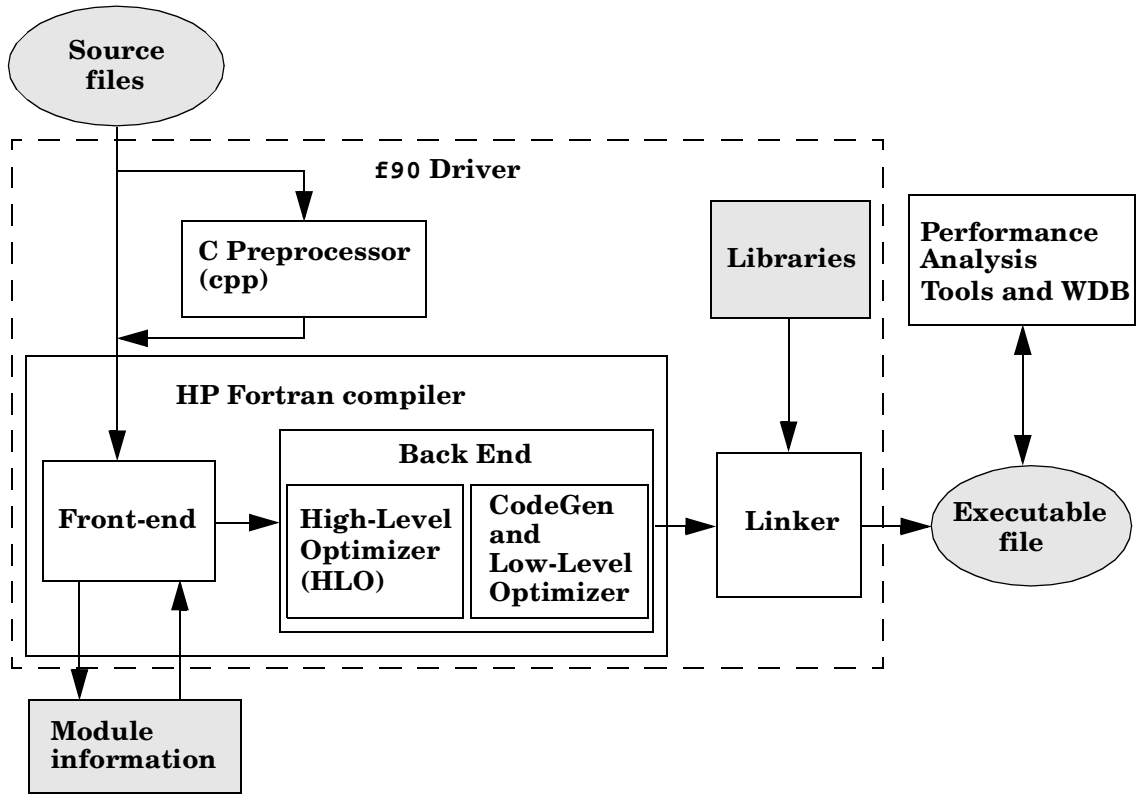
```
$ f90 +usage
```

For a full description of the options, refer to the *Parallel Programming Guide for HP-UX Systems (B3909-90008).*

The HP Fortran compiler environment

Figure 1 illustrates the compilation process, showing the different components of the HP Fortran compiler environment; active processes are unshaded and data elements are shaded. With the exception of the performance analysis tools and the debugger (WDB), all components are invoked by the `f90` command. The **C preprocessor** and linker can also be separately invoked by the `cpp` and `ld` commands; see the *cpp*(1) and *ld*(1) man pages, respectively, for more information. The remaining sections in this chapter briefly describe the different components and the command-line options that control them. Included in each section are references to other parts of this manual for more detailed information.

Figure 1 HP Fortran compiler environment



Driver

The **driver** parses the `f90` command line by which you invoke the compiler, calls each subprocess as needed to complete the compilation, and retains control throughout the compilation process.

Command-line options that control driver functions enable you to do the following:

- Call subprocesses that you want to substitute for those that `f90` calls by default
- Pass arguments to a subprocess
- Get a summary listing of all options supported by the compiler
- Display information about the version of `f90` you are using
- Control the level of information that the driver displays about the compilation process

Table 1 lists and briefly describes the options that control the driver.

Table 1 Options for controlling the `f90` driver

Option	Function
`-c`	Suppress the link phase and produce an object file (`.o`) from each source file on the command line.
`-o` *outfile*	Name the output file *outfile* instead of the default file name (`a.out` or *filename*`.o`). If linking has been suppressed, the front end uses this option to name the object file.
`+pre_include=`*file*	Process contents of *file* before all source files specified on the command line. The command line can have multiple occurrences of this option, each specifying a different *file*; they are processed in the specified order.

Option	Function
`-t`*x*`,`*name*	Substitute a private version (*name*) of one or more subprocesses (*x*) of the compilation. The values for *x* are: • `a` Assembler • `c` Compiler • `l` Linker • `p` C preprocessor • `s` Startup file (`crt0.o` library) • `e` Debug file (`end.o` library) If you compile and link separately and specify `+tl` on the compile line, you must also specify it on the link line.
`+usage`	List and briefly describe all `f90` options.
`-v`	Print verbose information to standard output as program is compiled.
`+version`	Write compiler version information to standard output, without compiling.
`-W`*x*`,`*arg1*`[,`*arg2*`,...,`*argN*`]`	Pass *arg1* through *argN* to a subprocess of the compilation, identified by *x*. The values for *x* are: • `a` Assembler • `c` Compiler • `l` Linker • `p` C preprocessor If you compile and link separately and specify `+Wl` on the compile line, you must also specify it on the link line.

C preprocessor

HP Fortran source files can include directives that the C preprocessor (`cpp`) reads and expands before the program is passed to the compiler. Among other things, `cpp` directives enable you to code parts of your program for conditional compilation. By default, the `f90` command passes source files ending in the `.F` extension to the C preprocessor.

Table 2 lists and briefly describes the options for controlling the preprocessor, including the `+cpp` option, which overrides the default behavior and passes all source files on the command line to the preprocessor. For additional information, see "Using the C preprocessor" on page 81 and the *cpp*(1) man page.

Table 2 Options for controlling the C preprocessor

Option	Function
`+cpp={yes|no|default}`	Invoke the C preprocessor. `+cpp=yes` passes all source files to the preprocessor. `+cpp=default` passes only files ending in the `.F` extension. `+cpp=no` suppresses passing any files. The default is `+cpp=default`.
`+[no]cpp_keep`	Retain [discard] output from the C preprocessor. If the source filename is *file*`.f` or *file*`.F`, output is stored in *file*`.i`; if the source filename is *file*`.f90`, the output filename is *file*`.i90`. The default, `+nocpp_keep`, is to discard the output.
`-D`*name*`[=`*def*`]`	Define the symbol *name* to the preprocessor. If *def* is specified, *name* is defined to that value.
`-I`*directory*	Add *directory* to the list of directories searched for files specified in include directives. The command line can have multiple occurrences of this option, each specifying a different directory.
`-U`*name*	Remove any initial definition of *name*, a reserved symbol that is predefined by the preprocessor.

Front-end

The **front-end** is responsible for parsing the source code and issuing warning and error messages when the parse fails. Command-line options enable you to control the front end's assumptions about the source code, including whether the source is in fixed or free format, uses implicit or explicit typing, and contains extensions. Other front-end options control the level of error messages and their language (Native Language Support), default data sizes, and search rules for **.mod** files. For a list of the options that control the front end, see Table 3.

Table 3 Options for controlling the front end

Option	Function
`+[no]autodbl`	Promote [do not promote] all integer, logical, and real items to 8 bytes, and all double-precision and complex items to 16 bytes. The default is `+noautodbl`. For information about using this option, see "Increasing default data sizes" on page 96.
`+[no]autodbl4`	Promote [do not promote] all integer, logical, and real items to 8 bytes, and complex items to 16 bytes. The `+autodbl4` option does not promote the size of double-precision and double-complex items. The default is `+noautodbl4`. For information about using this option, see "Increasing default data sizes" on page 96.
`+check={all|none}`	Enable (`+check=all`) or disable (`+check=none`) compile-time range checking of array subscripts. The default is `+check=none`. For information about using this option, see "Segmentation violation exception" on page 114.
`+[no]dlines`	Compile debug lines as source statements [comments]. Source lines must be in fixed format. The default, `+nodlines`, is to treat source lines with a `D` or `d` in column 1 as comments. For information on using this option, see "Using debugging lines" on page 117.

Option	Function
`+[no]escape`	Treat the backslash character (`\`) as a C-like escape [literal] character. The default is `+noescape`. For information on using this option when porting, see "Escape sequences" on page 232.
`+[no]extend_source`	Allow [do not allow] up to 254 characters on a single source line. The default, `+noextend_source`, is 72 characters for fixed format and 132 for free format. For information on using this option when porting, see "Source formats" on page 231.
`-I`*directory*	Add *directory* to the list of directories searched for files specified in `INCLUDE` lines and include directives, and for `.mod` files. The command line can have multiple instances of this option, each specifying a different directory. For information about using this option, see "Managing .mod files" on page 76.
`+[no]implicit_none`	Cause the types of identifiers to be implicitly undefined [defined]. The default is implicit typing (`+noimplicit_none`). For information about using this option, see "Disabling implicit typing" on page 90.
`+langlvl={90|default}`	Issue warnings for all extensions to the Fortran standard (`+langlvl=90`). The default, `+langlvl=default`, allows extensions. For information about using this option, see Chapter 11, "Porting to HP Fortran," on page 219.
`+[no]list`	Write [suppress] a program listing to standard output during compilation. The default is `+nolist`.
`+moddir=`*directory*	Write `.mod` files to *directory*. The default is to write `.mod` files to the current directory. For information about using this option, see "Managing .mod files" on page 76.
`+nls=`*lang*	Enable 16-bit Native Language Support (NLS) in strings and comments in the language specified by *lang*.

Option	Function
`+[no]onetrip`	Execute any counter-controlled `DO` loop at least once (`+onetrip`). The default is `+noonetrip`. For information about using this option when porting, see "One-trip DO loops" on page 228.
`+[no]ppu`	Postpend [do not postpend] underscores at the end of definitions of and references to externally visible symbols. The default is `+noppu`. For information about using this option when porting, see "Names with appended underscores" on page 231.
`+real_constant={single|double}`	Treat all single-precision real and complex constants as either single-precision (`+real_constant=single`) or double-precision (+real_constant=double). The default is `+real_constant=single`. This option has no effect on constants that are explicitly sized or when the `+autodbl` or `+autodbl4` option is specified. For information about using this option, see "Increasing the precision of constants" on page 94.
`+source={fixed|free|default}`	Accept source files in fixed format (`+source=fixed`) or free format (`+source=free`). The default, `+source=default`, is free for `.f90` files and fixed for `.f` and `.F` source files. For information about using this option, see "Source formats" on page 231.
`+[no]uppercase`	Use uppercase [lowercase] for all external names. The default is `+nouppercase`. For information about using this option, see "Case sensitivity" on page 170.
`-w`	Suppress warning messages.

Back-end

The two main functions of the **back-end** are:

- To optimize your program for faster performance
- To generate the code that goes into the object file

Optimization is performed by two subcomponents of the compiler's back end:

- The **High-Level Optimizer (HLO)**, which performs large-scale, high-semantic-level analyses and transformations to increase performance.
- The low-level optimizer, which performs traditional optimizations (such as common subexpression elimination and dead-code removal) as well as machine-specific optimizations.

Options for controlling optimization form the largest group of the command-line options. These options enable you to do the following:

- To set the level of optimization that is applied to your program
- To apply a package of optimizations that meet certain requirements of your application—for example, optimizations that favor compile-time speed over performance
- To apply specific optimization technologies to your program, or to specific parts of your program, for fine-tuning performance

Table 4 lists (in summary form) the options that control optimization. For information about how to use these options, see "Using options to control optimization" on page 135.

NOTE If you use the `f90` command to compile and link on separate command lines, many of the optimization options must appear on both the command line and the link line; see "Performance and optimization" on page 131. For information about using f90 to compile and link, see "Linking with f90 vs. ld" on page 65.

Table 4 Options for controlling optimization

Option	Function
`+DC7200`	Perform memory hierarchy optimizations for the PA7200 processor.
`-O[`*optlevel*`]`	Optimize program, where *optlevel* is 0 (no optimization), 1, 2, or 3 (the highest level). If *optlevel* is not specified, the program is optimized at level 2 (`-O2`).
`+O`*optlevel*	This option has the same meaning as the `-O[`*optlevel*`]` option, except that optlevel must be specified. It is provided for compatibility with makefiles.
`+O[no]info`	Provide [do not provide] feedback information about the optimization process. This option is most useful at optimization level 3 and higher. The default is `+Onoinfo`.
`+O[no]`*optimization*	Enable [disable] *optimization*, a predefined string that indicates a category of optimizations (for example, those that do not increase code size) or a specific optimization technology (for example, inlining). See the *HP Fortran Programmer's Reference*, for the different values for *optimization*.

The other component of the back end is the code generator (CodeGen), which you can control by using the command-line options in Table 5. These options allow you to specify (among other things) that the output file include debugging or profiling information or that local variables be saved in static memory.

Table 5 Options for controlling code generation

Option	Function
`+[no]asm`	Compile the named source files and leave [do not leave] the assembly language output in corresponding files whose names are suffixed with `.s`. The default is `+noasm`.
`+DA`*model*	Generate code for a specific version of the PA-RISC architecture. *model* can be one of the following: • PA-RISC version number (`1.1` or `2.0`) • A model number (for example, `750` or `870`). • One of the PA-RISC processor names (for example, `PA7000`, `PA7100`, or `PA8000`). • The word `portable` to generate code compatible across all PA-RISC 1.1 and 2.0 workstations and servers. For information about using this option, see "Compiling for different PA-RISC machines" on page 77.
`+DS`*model*	Perform instruction scheduling appropriate for a specific implementation of the PA-RISC architecture. *model* can be one of the following: • PA-RISC version number (`1.1` or `2.0`) • A model number (for example, `750` or `870`). • One of the PA-RISC processor names (for example, `PA7000`, `PA7100`, or `PA8000`). For information about using this option, see "Compiling for different PA-RISC machines" on page 77.
`-g`	Generate debugging information needed by the debugger. This option is compatible with optimization levels 0, 1, and 2. If you compile and link separately and specify `-g` on the command line, you must also specify it on the link line. For information about using this option to prepare programs for the debugger, see "Using the HP WDB debugger" on page 108.

Option	Function
`+[no]gprof`	Prepare [do not prepare] object files for profiling with `gprof`; see the *gprof*(1) man page. The default is `+nogprof`. If you compile and link separately and specify `+gprof` on the command line, you must also specify it on the link line. For information about using this option to profile programs with `gprof`, see "gprof" on page 133.
`+k`	Generate code for programs that reference a very large number of shared data items. The linker will issue a diagnostic message in the rare cases when this option is needed.
`+pic={short|long|no}`	Generate Position Independent Code (PIC) with short displacements (`+pic=short`) or long displacements (`+pic=long`) for use in shared libraries. The default is `+pic=no`. For information about using this option when creating shared libraries, see "Compiling with +pic" on page 79.
`+[no]prof`	Prepare [do not prepare] object files for profiling with `prof`; see the *prof*(1) man page. The default is `+noprof`. If you compile and link separately and specify `+prof` on the command line, you must also specify it on the link line. For information about using this option to profile programs with `prof`, see "gprof" on page 133.
`+[no]save`	Save [do not save] all local variables in all subprograms. For information about using this option when porting, see "Uninitialized variables" on page 226.

Linker

The linker (`ld`) builds an executable program from the object files produced by the back end and the libraries. An important group of options for controlling the linker specify what libraries the linker should search and where it should search for them. Other options control the type of information that the linker should or should not include in its output file, such as symbol table information used by the debugger or marks identifying the output file as shareable or demand-loadable. Table 6 lists and briefly describes options that control the linker.

NOTE If you use the `f90` command to compile and link on separate command lines and compile with any of the options (except `-c`) listed in Table 6, you must specify the same options on the link line as well.

Table 6 Options for controlling the Linker

Option	Function
`-c`	Suppress linking; produce object files only.
`+[no]demand_load`	Mark [do not mark] the output file from the linker *demand load*. If you compile and link separately and specify `+demand_load` on the command line, you must also specify it on the link line. The default is `+nodemand_load`. For information about using this option, see "Creating demand-loadable executables" on page 84.
`+FP`*flags*	Specify how the runtime environment for trapping floating-point exceptions should be initialized at program startup. If you compile and link separately and specify `+FP` on the command line, you must also specify it on the link line with the identical set of *flags*. The default is that all traps are disabled. See the *ld*(1) man page for specific values for *flags*. For information using this option, see "Floating-point exceptions" on page 113.

Option	Function
`+[no]fp_exceptions`	Enable [disable] floating-point exceptions. Enabling floating-point exceptions also causes the running program to issue a procedure traceback for runtime errors. The default is `+nofp_exceptions`. For information using this option, see "Floating-point exceptions" on page 113.
`-L`*directory*	Add *directory* to the front of the library search path. This option applies only to libraries specified by the `-l` option (see below). If you compile and link separately and specify `-L` on the command line, you must also specify it on the link line. For information about using this option, see "Library search rules" on page 71.
`-l`*x*	Link the library `lib`*x*`.a` or `lib`*x*`.sl` to the executable program. If you compile and link separately and specify `-l` on the command line, you must also specify it on the link line. For information about using this option, see "Linking to nondefault libraries" on page 68 and the *ld*(1) man page.
`-o` *outfile*	Name the output file *outfile* instead of the default `a.out`. If linking is suppressed (`-c`), this option is used instead to name the object files.
`+[no]shared`	Cause the output file from the linker to be marked *shared* [*unshared*]. If you compile and link separately and specify `+shared` on the command line, you must also specify it on the link line. The default is `+shared`. For information about using this option, see "Creating shared executables" on page 84.
`+[no]strip`	Strip [do not strip] symbol table information from the linker output. For more information, see the *ld*(1) and *strip*(1) man pages. This option is not compatible with `-g`. If you compile and link separately and specify `+strip` on the command line, you must also specify it on the link line. The default is `+nostrip`. For information using this option, see "Stripping debugging information" on page 110.

Option	Function
`+[no]ttybuf`	Use buffered [unbuffered] output to the terminal. The default is `+ttybuf`.
`+[no]U77`	Invoke [do not invoke] support for the **libU77** library (BSD 3f). If you compile and link separately and specify `+U77` on the compile line, you must also specify it on the link line. The default is `+noU77`. For information about the `libU77` library, see "Additional HP Fortran libraries" on page 69 and the *HP Fortran Programmer's Reference*.
`-Wl,`*options*	Pass a comma-separated list of *options* to the linker. For information about options supported by the linker, see the *ld*(1) man page.

Tools

The HP Fortran compiler environment includes a high-level language debugger and performance analysis tools. The debugger is HP WDB, which includes a graphical user interface. To prepare a program for debugging, you must compile it with the `-g` option. For information about this option, see "Using the HP WDB debugger" on page 108.

The performance analysis tools include the standard UNIX utilities, `prof` and `gprof`. To use `prof` and `gprof`, you must compile with the `+prof` and `+gprof` options, respectively. For more information about all of the performance analysis tools, see "Using profilers" on page 132 and the *CXperf*(1), *prof*(1), *gprof*(1), and *ttv*(1) man pages.

HP-UX operating system

Although the HP-UX operating system does not appear in Figure 1 on page 2, it provides a variety of resources for programs executing within HP-UX. For example, HP-UX captures the command line you use to invoke an executable program, breaks it up into arguments, and makes them available to your program.

HP-UX also has many callable system routines that provide low-level access to kernel-level resources. For example, your program can call HP-UX file-processing routines as alternatives to Fortran I/O.

"Writing HP-UX applications" on page 155 discusses how HP Fortran programs can take advantage of HP-UX resources. For a full description of HP-UX system routines, see the *HP-UX Reference*.

2 Compiling and linking

This chapter discusses how to compile and link HP Fortran programs and covers the following topics:

- Compiling with the f90 command
- Linking HP Fortran programs
- Special-purpose compilations
- Using environment variables

Compiling with the f90 command

The default behavior of the `f90` command is to compile source files listed on the command line and, if the compilation is successful, to pass the resulting object files to the linker. If the link stage is successful, the linker produces an executable program with the default name `a.out`.

Consider, for example, the program `hello.f90`:

hello.f90

```
PROGRAM main
  CALL hello()
END PROGRAM main

SUBROUTINE hello()
  PRINT *, 'Hello, I must be going.'
END SUBROUTINE hello
```

When compiled with the command line:

```
$ f90 hello.f90
```

`f90` produces two files, `hello.o` (object code) and `a.out` (the executable program).

If the command line contains only an object file, as in the following:

```
$ f90 hello.o
```

`f90` passes the object file to the linker, which (if successful produces the executable program `a.out`.

Here is a sample run of the executable program:

```
$ a.out
 Hello, I must be going.
```

This section provides more detailed information about using the `f90` command, including:

- Command-line syntax
- Command-line options
- Filenames recognized by `f90`

f90 command syntax

The syntax for using the `f90` command is:

`f90` [*options*] [*files*]

where *options* is a list of one or more command-line options and *files* is a list of one or more files containing HP Fortran source code to be compiled or object code to be linked. Items in *options* and *files* can be interspersed on the command line, separated by spaces. However, some options are order-sensitive. For example, the `-l` option, which is used to specify a library for linking, must follow the program file to which it will be linked.

For information about using the `-l` option, see "Linking to nondefault libraries" on page 68. For more information about the `f90` command line, see *HP Fortran Programmer's Reference*.

Command-line options

Command-line options enable you to override the default behavior of the `f90` command. Some options affect how files are compiled or linked; for example, the `-O` option requests optimization. Other options may cause the `f90` command to skip a process entirely; for example, the `-c` option compiles without linking. And still others invoke processes other than the default ones; for example, the `+cpp=yes` option causes the f90 command to send source files to the C preprocessor (`cpp`) before compiling. (For information about using `cpp`, see "Using the C preprocessor" on page 81.)

Many options are of the form +[no]*option*, where +*option* enables the option's functionality and +no*option* disables it. Other options have more than just an on or off state; these are of the form +*option*=*arg*. You can cause `f90` to list the values for *arg* on `stderr` by specifying just the option name without an argument. For example, given the command line:

```
$ f90 +langlvl= prog.f90
```

`f90` will issue the following message:

```
f90: The '+langlvl=' option requires
     one of the following sub-options:

     90        generate messages about non-FORTRAN 90 features
     default   no messages about nonstandard FORTRAN features
```

Still other options take a name as an argument. For example, the `-o`*name* option specifies the name you want to give to the output file. If you misspell an option on the `f90` command line, the driver looks for options that are similar to the one you entered and lists them as possible alternatives on `stderr`. It meanwhile compiles the program without the option in question.

For detailed information about the syntax of all the options, see the *HP Fortran Programmer's Reference*. For a brief descriptive list of the options, use the command line:

```
$ f90 +usage
```

Commonly-used options

Table 7 identifies commonly-used command-line options for when you want to change the compiler's default behavior. For a complete and full description of all HP Fortran command-line options, see "Option descriptions" on page 24.

Table 7 Commonly-used options

Option	Function
`-c`	Compile without linking. Use this option to compile and link in separate steps.
`-g`	Prepare program for debugging. Use this option to prepare your program for debugging.
`-L`*directory*	Specify where to look for libraries; applies only to succeeding `-l` options. Be careful about using this option if the `LPATH` environment variable is set.
`-l`*x*	Specify a library. Use this option to link in library routines.
`-O`	Optimize. Use this option to optimize your program at the default level of optimization.
`+save`	Give the `SAVE` attribute to local variables. Use this option when porting older Fortran programs that may contain uninitialized variables.

Option	Function
`-o` *outfile*	Name the output file *outfile*. Use this option to name the executable or object file.
`+usage`	List all compile-line options currently supported by `f90`.
`-v`	Enable verbose mode. Use this option to get a report on the compilation process.

Command-line options by category

Table 8 categorizes the `f90` command-line options. For detailed information about each of the options, see "Option descriptions" on page 24.

Table 8 Options listed by category

Category	Options
Compatibility and porting	`+autodbl`, `+autodbl4`, `+charlit77` `+[no]es`, `+extend_source`, `gformat77`, `+i8`, `+io77`, `+langlvl`, `+multi_open`, `+nocheckuf`, `+nopadsharedcommon`, `+onetrip`, `+ppu`, `+r8`, `+save`, `+[no]signedzero`, and `+U77`
Compiler configuration	`-t` and `-W`
Data storage	`+autodbl`, `+autodbl4`, `+hugecommon`, `+hugesize`, `+i8`, `+indirectcommonlist=file`, `+nopadsharedcommon`, `+r8`, `+real_constant`, and `+save`
Directory, module, and library search path	`+U77`, `-I`, `-L`, `-l`, and `+moddir`
Debugging	`+check`, `+dlines`, `-g`, and `+fp_exception`
Error control	`+FP` and `+fp_exception`

Category	Options
Industry standardized extensions	+O[no]openmp
Language features	+escape, +extend_source, +implicit_none, +langlvl, +[no]signedzero, and +source
Linking	+demand_load, -dynamic, +FP, +k, -L, -l, +shared, +sharedlibF90, +strip, and +uppercase
Listing and messages	+asm, +langlvl=90, +list, -v, +version, +what, and -w
Native language support	+nls
Performance and optimization	+cat,+DA, +DC, +DO, +DS, +fastallocatable, +O, and-O +Oparallel_intrinsics
Preprocessor	+cpp, +cpp_keep, -D, and -U
Profiling	+gprof, +prof, +pa, +pal
Miscellaneous	+asm, -c, +getarg0, +getarg1, +noalign64bitpointers, -o, +pic, +pre_include, +ttybuf, and +usage, +Z, +z

Option descriptions

The following alphabetical list describes each of the command-line options recognized by HP Fortran. The +usage option also lists and briefly describes all of the currently supported options.

+[no]asm

+asm compiles the named programs and leaves the assembler-language output in corresponding files whose names have the .s extension. The assembler-language output produced by this option is not supported as input to the assembler.

The default is +noasm.

The -S option can be used to perform the same function as +asm.

+[no]autodbl

+autodbl increases the default size of integer, logical, and real items to 8 bytes; see Table 9. It also increases the default size of double precision and complex items to 16 bytes. This option does not increase the size of the following:

- Items of character type
- Items declared with the BYTE statement
- Items declared with the DOUBLE COMPLEX statement
- Explicitly sized items

For example, the following are unaffected by +autodbl:

```
INTEGER(KIND=4)
INTEGER(4) J
REAL*8 D
3.1416_4, 113_4
```

Note, however, that constants specified with an exponent—for example, 4.0E0 and 2.3D0—are doubled.

Items promoted include constants, scalar variables, arrays, components of derived types, and record fields. This option also promotes intrinsics as well as the results and arguments of user-defined functions to the proper precision. Types specified in ON statements are also promoted.

The entire program should be compiled with this option, not just selected files.

This option is useful when porting programs that depend on the increased precision of 8 and 16 bytes. If you want to promote only single-precision items, use the +autodbl4 option. (REAL(KIND=16) arithmetic is slow.)

The default is +noautodbl.

Table 9 Data type sizes and +autodbl[4]

	Sizes in bytes of intrinsic types		
	Integer, logical, and real	**Double precision**	**Complex**
Default sizes	4	8	8
`+autodbl`	8	16	16
`+autodbl4`	8	8	16

`+[no]autodbl4`

Like `+autodbl`, `+auotdbl4` increases the default size of integer, logical, and real items to 8 bytes, and the default size of complex items to 16 bytes; see Table 9. Unlike `+autodbl`, it does not increase the default size of double precision.

This option does not increase the size of the following:

```
INTEGER(KIND=4)
INTEGER(4) J
REAL*8 D
3.1416_4, 113_4
```

Note, however, that constants specified with an exponent—for example, `4.0E0` and `2.3D0`—are doubled.

Items promoted include constants, scalar variables, arrays, components of derived types, and record fields. This option also promotes intrinsics as well as the results and arguments of user-defined functions to the proper precision. Types specified in `ON` statements are also promoted.

The entire program should be compiled with this option, not just selected files. Use this option when you want to promote only the single-precision items.

The default is `+noautodbl4`.

NOTE The `+autodbl4` option causes items declared as `REAL`, `INTEGER`, and `DOUBLE PRECISION` all to have the same size. This violates the Fortran Standard.

`-c`

`-c` compiles the specified source files but does not link them. The compiler produces a relocatable file (`.o`) for each file in the files list (these may include `.f90`, `.f`, `.F`, `.i`, `.i90`, and `.s` files). When using `-c` and `-o` together, you may specify only one source file on the command line; the resulting object file is renamed.

`+charlit77`

`+charlit77` causes character literals to be placed in writable static storage. This allows character strings passed as actual arguments to be modified by the called routine.

`+check={all|none}`

`+check=all` enables compile-time range checking for array subscripts. The `+check=all` option will also cause an executing program to halt with a runtime error if any of the following is detected:

- Integer overflow
- Out-of-bounds subscripts
- Out-of-bounds substrings

The default is `+check=none`. The `-C` option can be used to perform the same function as `+check=all`.

`+cpp={yes|no|default}`

`+cpp=yes` tells the compiler to pass the source files specified on the command line to the C preprocessor before passing them on to the compiler. This option does not apply to `.i` and `.i90` files.

The default, `+cpp=default`, is to apply the C preprocessor to files that end in the `.F` extension but not to files that end in `.f` or `.f90`.

Specifying `+cpp=no` tells the compiler not to invoke the C preprocessor for all files on the command line, including those ending in `.F`.

If you want to keep the output from the C preprocessor, use the `+cpp_keep` option.

`+[no]cpp_keep`

`+cpp_keep` causes the compiler to retain the output from the C preprocessor. If the source filename is *file*`.f` or *file*`.F`, the output filename is *file*`.i`; if the source filename is *file*`.f90`, the output filename is *file*`.i90`. The compiler will accept source files with the `.i` and `.i90` extensions.

The default, `+nocpp_keep`, is to discard the output file.

Note that this option does not pass source files to the C preprocessor. To do that, you must also specify the `+cpp=yes` option.

`-D` *name*[`=`*def*]

`-D` defines a symbol name (*name*) to the C preprocessor. If you do not provide a definition (*def*) for the symbol name, *name* is defined as `1`. This option applies only to files that are passed to the C preprocessor.

`+DA`*model*

`+DA` generates object code for a particular version of the PA-RISC architecture. By default, the compiler generates code for the machine model you are compiling on. With this option, you can override the default, causing the compiler to generate code for the machine on which the program will execute rather than for the machine on which it is compiled.

model can be one of the following:

- A PA-RISC version number (`1.1` or `2.0`)
- A model number (for example, `750` or `870`)
- One of the PA-RISC processor names (for example, `PA7000`, `PA7100`, or `PA8000`)
- The word `portable`, which causes the compiler to generate code that is compatible across all PA-RISC 1.1 and 2.0 systems

See the file `/usr/lib/sched.models` for model numbers and their architectures. Use the `uname` command to determine the model number of your system. (For information about the `uname` command, see *uname*(2).)

For best performance, use `+DA` with the model number or architecture of the machine on which you plan to execute the program.

The `+DA` option also specifies the appropriate search path for HP-UX math libraries. If your program calls mathematical functions, `+DA2.0` links in the PA2.0 version of the math library, while `+DA1.1` links in the PA1.1 library version. (For more information about using math libraries, see the *HP-UX Floating-Point Guide*.)

With +DA2.0W, memory addresses are 64-bit values. This allows common blocks and dynamically allocated memory to exceed 32-bit address limits. This feature is restricted by the available virtual memory on the system where the application is run.

NOTE You must specify +DA2.OW to generate 64-bit code. At PA64, all data types remain the same size as at PA32 except for pointers. Fortran pointers are hidden from the user and cannot be directly manipulated.

`+DC7200`

`+DC7200` performs memory hierarchy optimizations for the PA7200 processor.

`+[no]demand_load`

`+demand_load` causes the output file from the linker to be marked `demand load`. When a process is marked `demand load`, its pages are brought into physical memory only when they are accessed. The default, `+nodemand_load`, causes the output file from the linker not to be marked `demand load`.

The `-q` option performs the same function as `+demand_load`, and the `-Q` option performs the same function as `+nodemand_load`.

`+[no]dlines`

`+dlines` treats source lines with a "`D`" or "`d`" in column 1 as statements to be compiled. The default, `+nodlines`, treats lines beginning with "`D`" or "`d`" in column 1 as comments.

The `+dlines` option must be used only with source files in fixed-format.

`+DO`*osname*

`+DOosname` sets the target operating system for the compiler. The osname variable can be 11.0EP9806 (indicating the HP-UX 11.0 EXTPAK 9806 release) or 11.0 (the default). When +DO11.0EP9806 and `+Olibcalls` are both specified on an HP-UX 11.0EP9806 system, the compiler enables the fusing of library calls where applicable. This promotes instruction-level parallelism in library routines which can improve performance by concurrently computing the same function of two values.

By default, when you compile your application, it is binary compatible across the 11.x release. You only need to specify +DO when you want the latest performance features supported in the OS.

`+DS`*model*

+DS specifies an instruction scheduling algorithm for a particular implementation of the PA-RISC architecture, as specified by *model*.

model can be one of the following:

- A PA-RISC version number (1.1 or 2.0)
- A model number (for example, 750 or 870)
- One of the PA-RISC processor names (for example, PA7000, PA7100, or PA8000)

For example, specifying +DS750 performs instruction scheduling tuned for one implementation of PA-RISC 1.1. Specifying +DS2.0 or +DS1.1 performs scheduling for a representative PA-RISC 2.0 or 1.1 system, respectively. To improve performance on a particular model, use +DS with that model number.

See the file /usr/lib/sched.models for model names and numbers, as well as their architectures. Use the uname -a command to determine the model number of your system. (For more information about uname, see *uname*(2).)

Object code with scheduling tuned for a particular model will execute on other systems, although possibly less efficiently.

If you do not use this option, the compiler uses the argument specified with the +DA option. If you use neither +DS or +DA, the default instruction scheduling is for the system on which you are compiling.

-dynamic

-dynamic is used to generate dynamically-bound executables.

+[no]escape

+escape treats the backslash character (\) as a C-like escape character. The default, +noescape, treats the backslash character as a literal character.

+[no]es

`+[no]es` is similar to `+[no]extend_source` except that character literals and hollerith constants continued across a line boundary are not padded. This option provides compatibility with FORTRAN 77's `+es` option.

`+[no]extend_source`

`+extend_source` allows extended source lines, which may contain up to 254 characters. The default, `+noextend_source`, restricts fixed-format source lines to 72 characters and free-format source lines to 132 characters.

Programs that depend on the compiler's ignoring characters past column 72 will not compile correctly with the `+extend_source` option.

`+fastallocatable`

`+fastallocatable` enables a different representation for allocatable arrays in the object code produced by the compiler. This alternate representation avoids problems in optimizing code containing allocatable array references. Additionally, this alternate representation for allocatable arrays is binary compatible with the old representation.

`+FP`*flags*

`+FP` initializes the *flags* that specify how runtime floating-point exceptions should be trapped; uppercase *flags* enable traps, lowercase *flags* disable traps. *flags* can be concatenated to specify a desired behavior and may not contain spaces or tabs. Valid values for *flags* are identified in Table 10.

By default, all traps are disabled. However, you can specifically disable a behavior either by excluding the upper-case letter from *flags* or by including the equivalent lower-case letter (`v`,`z`,`o`,`u`,`i`,`d`) in *flags*. For example, the following command lines are equivalent:

```
$ f90 +FPvZI test.f90
$ f90 +FPZI test.f90
```

If you are using PA1.1 libraries, you can dynamically change these settings at run time by using the *fpsetdefaults* or *fpsetmask* routines. For more

information about these routines, see the *fpgetround*(3M) man page and the *HP-UX Floating-Point Guide*.

Enabling sudden underflow may cause the same program to compute different results on different implementations of the PA-RISC 1.1 and 2.0 architectures. This is because some hardware implementations have sudden underflow available, while others do not. The `+FPD` option enables the hardware to flush denormalized values to zero, but it does not require that it do so.

Table 10 Values for the `+FP` option

Value	Meaning
`V`	Trap on invalid floating-point operations. Examples of invalid floating-point operations include the following: • Arithmetic operation on NaNs • Operations such as `(+inf) + (-inf)` and `(+inf) - (+inf)` • Multiplication of 0 and infinity • Division operations `0/0` and `inf/inf` • Certain floating-point remainder operations • Square root of a negative value • Certain kinds of comparisons of unordered values
`Z`	Trap on floating-point divide by zero.
`O`	Trap on floating-point overflow.
`U`	Trap on floating-point underflow.
`I`	Trap on floating-point operations that produce inexact results. Inexact result traps may occur whenever roundoff is necessary to produce the result. For example, the fraction `1.0/3.0` produces an inexact trap because there is no exact floating-point representation for this fraction.

Value	Meaning
D	Enable sudden underflow (flush to zero) of denormalized values on those PA-RISC systems greater than version 1.0 that have implemented sudden underflow. (That is, +FPD enables sudden underflow only if it is available on the processor that is used at run time.) Denormalized values are those values whose absolute value is very close to zero. For IEEE single precision data types, the largest denormalized value is approximately equal to 2^{-126}. For IEEE double precision data types, such values are approximately equal to 2^{-1022}. Sudden underflow will cause some floating-point applications to run faster, with a possible loss of numerical accuracy on numbers very close to zero.

+[no]fp_exception

+fp_exception causes a descriptive message and a procedure traceback to be issued to standard error when the HP-UX signals listed in Table 11 are generated.

By default, floating-point exceptions are disabled on Series 700/800 systems, in accordance with the IEEE standard.

For a description of these signals, see *signal*(2) and *signal*(5) in the *HP-UX Reference*. For information about floating-point exceptions and error handling, see the *HP-UX Floating-Point Guide*.

You can also use the ON statement to write your own trap procedures. For information about the syntax of the ON statement, see "Using the ON statement" on page 119.

The default, +nofp_exception, disables traceback information.

Table 11 Signals recognized by the `+fp_exception` option

Signal	Meaning
SIGILL	Illegal instruction
SIGFPE	Floating-point exception
SIGBUS	Bus error instruction
SIGSEGV	Segmentation violation
SIGSYS	Bad argument to system call

`-g`

`-g` causes the compiler to generate information for use by the HP WDB debugger. The `-g` option can be used to prepare code for debugging that has been compiled with optimization options `-O`, `-O1/+O1`, and `-O2/+O2`, but not `-O3/+O3` or higher.

`+getarg`

`+getarg0 and +getarg1` control the behavior of the getarg intrinsic subroutine. `+getarg0` requests the industry standard behavior for getarg, where an index value of zero causes the program name to be returned. HP's FORTRAN 77 getarg intrinsic also implements this industry standard convention. `+getarg1` is used to request non-standard behavior, where an index value of one causes the program name to be returned (older releases of HP Fortran behaved in this manner). The default is `+getarg0`.

`gformat77`

`gformat77` requests the FORTRAN 77 style of formatting a value of zero with the G edit descriptor. Fortran 90 uses an F edit descriptor when the value being written is zero, while FORTRAN 77 uses an E edit descriptor.

`+[no]gprof`

`+gprof` prepares object code files for profiling with `gprof`. The default is `+nogprof`.

`gprof` is provided as part of the "HP-UX General Programming Tools" product; see *gprof*(1).

The `-G` option can be used to perform the same function as `+gprof`.

`+hugecommon`

+hugecommon instructs the compiler to place the specified COMMON block into a huge data segment. The format for this option is:

```
+hugecommon=name
```

where name is the name of a COMMON block. By default, only COMMON blocks larger than 2 gigabytes are placed into huge data segments.

For example:

```
% f90 +hugecommon=results pcvals.f90
```

places the COMMON block named results into a huge data segment.

`+hugecommon` is especially useful when a program contains several different COMMON blocks that together occupy more than two gigabytes but individually occupy less than two gigabytes. In this situation, the largest COMMON blocks could be placed in a huge data segment when the program is compiled by specifying their names in multiple +hugecommon options.

If a common block is specified as huge in one object file, it must be specified huge in all object files. If it is not, the program will fail to link.

NOTE PA2.0W objects cannot be combined with 32-bit object files. 64-bit applications will only execute on PA8000-based systems.

`+hugesize`

+hugesize instructs the compiler to place COMMON blocks that are larger than the specified size into a huge data segment. The format for this option is:

```
+hugesize=n
```

where n is the size in kilobytes (1024 bytes).

The default is to place COMMON blocks larger than two gigabytes (2147483648 bytes) into huge data segments; that is, +hugesize=2097152 is the default.

For example:

```
% f90 +hugesize=1024 hello.f90
```

specifies that COMMON blocks larger than 1048576 bytes (1 megabyte) should be placed into a huge data segment.

If a common block is specified as huge in one object file, it must be specified huge in all object files. If it is not, the program will fail to link.

PA2.0W objects cannot be combined with 32-bit object files. 64-bit applications will only execute on PA8000-based systems.

`-I` *directory*

`-I` specifies a directory where `.mod` files and files named in the `INCLUDE` line or in `#include` directives may be found if their name is a relative pathname—that is, does not begin with a slash (`/`). Directories are searched in the following order:

- The current source directory—that is, the directory containing the file with the `INCLUDE` line or `#include` directive.
- Directories specified by the `-I` option, in the order specified
- The current working directory
- The `/usr/include` directory

`+i8`

`+i8` changes 4-byte integer and logical constants, intrinsics, and user variables to 8-byte integers (rather than the 4-byte default).

`+[no]implicit_none`

`+implicit_none` forces the types of identifiers to be implicitly undefined. This is equivalent to specifying `IMPLICIT NONE` for each program unit in each file in the *files* list. The source code that is to be compiled with this option may contain other `IMPLICIT` statements; the statements will be honored. The default, `+noimplicit_none`, allows identifiers to be implicitly defined.

`+indirectcommonlist=file`

The common blocks listed in file (one per line, no enclosing `' / 's`) are treated as shared common blocks, but are not attached. The user must attach or otherwise allocate storage for such common blocks before they are referenced.

A C language program would typically be used to either attach a shared memory segment, or malloc a block of memory, and store that address into the external symbol for the common block. All Fortran code that references such a common block will indirect through the address in the external symbol for that indirect common block.

All source files that reference variables in such a common block must be compiled with the `+indirectcommonlist` flag, and that common block name must appear in the named file.

`+k`

`+k` generates code for programs that reference a very large number of shared data items. The linker will issue a diagnostic message in the rare case when this option is needed. By default, the compiler generates short-displacement code sequences for programs that reference global data in shared libraries. For nearly all programs, this is sufficient.

`-L` *directory*

For libraries named in `-l` operands, look in *directory* before looking in the standard places. You can specify multiple directories; each *directory* must be preceded by its own `-L` option. Directories named in `-L` options are searched in the specified order. This option must precede the `-l` option on the command line.

`-l`*x*

`-l` causes the linker to search the library named by either `/lib/lib`*x*`.a` (or `.sl`) or `/usr/lib/lib`*x*`.a` (or `.sl`); the linker searches `/lib` first. The current state of the `-a` linker option determines whether the archive (`.a`) or shared (`.sl`) version of the library is searched. See the *ld*(1) man page for information about `-a` option.

`+langlvl={90|default}`

`+langlvl=90` checks for strict compliance to the Fortran 90 Standard and issues warnings for any HP Fortran extensions to the Standard. The default, `+langlvl=default`, allows extensions.

`+[no]list`

`+list` produces a source listing on standard output. The default, `+nolist`, is not to produce a source listing.

`+moddir=`*directory*

`+moddir` directs the compiler to write `.mod` files to *directory*. If this option is not specified, the compiler writes modules in the current directory.

`+noalign64bitpointers`

`+noalign64bitpointers` disables correct alignment of pointers in derived types when compiling for wide mode (`+DA2.0W`). Earlier releases of Fortran 90 improperly aligned such pointers, occasionally leading to runtime aborts. Since this change introduces a potential binary incompatibility, the `+noalign64bitpointers` flag is provided to maintain the old behavior. Users who compile in wide mode (`+DA2.0W`)—and have derived types that contain components with the `POINTER` attribute—should recompile all source files that reference variables of that derived type. Users who have successfully used such derived type variables with older releases, and do not wish to recompile all affected source files, should always specify `+noalign64bitpointers` when compiling affected source files.

`+nocheckuf`

`+nocheckuf` disables the OPEN statement error check for opening text files with `ACCESS="sequential"`, `FORM="unformatted"`. This option is useful only when BUFFERIN/BUFFEROUT statements will be used to access the opened unit. The main program must be compiled with this option for it to have any effect, and all OPEN statements will then skip this error check.

`+nls=`*lang*

`+nls` enables 16-bit Native Language Support processing in character strings and comments for the specified language *lang*. For details on Native Language Support, refer to *Native Language Support User's Guide*.

The `-Y` option can be used to perform the same function as `+nls`.

`+nopadsharedcommon`

Do not pad shared common blocks to a multiple of 8 bytes. This option is useful when sharing shared common blocks between f77-generated programs and f90-generated programs. All source files referencing the same shared common block must be compiled with the same setting of this flag.

`-O`[*n*]

`-O` invokes the optimizer, where *n* is the level of optimization, 0 - 4. (`+O4` is recognized but not supported.) The default is optimization level 2.

This option is provided for compatibility and is functionally the same as the `+O`*n* option. The only difference between the two is that the level number is optional for the `-O` option. For more information about the levels of optimization, see the `+O`*n* option.

`+O`*n*

`+O` invokes the optimizer, where *n* is the level of optimization, 0 - 4. `+O4` is recognized but not supported and is provided for compatibility with the `f77` option, `+O4`. The `-g` option is compatible with the `+O0`, `+O1`, and `+O2` options.

Table 12 lists and describes the different levels of optimization.

NOTE See the *Parallel Programming Guide for HP-UX Systems* for a detailed description of optimization levels and methods.

`+O[no]`*optimization*

`+O[no]` options enable or disable specific optimizations or classes of optimizations (for example, optimizations that affect compilation time). For detailed information about `+O[no]`*optimization*, see “Using optimization options” on page 52.

Table 12 Levels of optimization

Level	Optimizations
0	Local optimizations, including constant folding and partial evaluation of test conditions.
1	Peephole optimizations, including: • Basic block optimizations • Branch optimizations • Instruction scheduling
2	Optimizations performed at level 1, plus the following: • Coloring register allocation • Induction variables and strength reduction • Common subexpression elimination • Loop invariant code motion • Store/copy optimization • Unused definition elimination • Dataflow analysis • Software pipelining • Scalar replacement • Sum reduction optimization
3	Optimizations performed at levels 1 and 2, plus the following: • Interprocedural optimizations, including cloning and inlining • Loop transformations to improve memory performance, including fusion and interchange

Level	Optimizations
4	Level 4 optimizations are not currently supported by the compiler. If `+O4` is specified, the compiler will issue a warning message and compile at optimization level 3.

`-o` *outfile*

`-o` names the executable file *outfile* rather than the default name of `a.out`. If not specified, `a.out` will be overwritten if it exists, or created if it does not. The *outfile* name must not end with `.f`, `.f90`, `.F`, `i`, or `.i90`. Also, it must not begin with `+` or `-`. When using `-c` and `-o` together, you may specify only one source file on the command line; the resulting object file is renamed.

`+[no]onetrip`

`+onetrip` generates code that executes any `DO` loop at least once. In accordance with the language standard, HP Fortran will not execute a `DO` loop if either of the following conditions is true:

- The increment value is greater than zero, and the initial value is greater than the limit.
- The increment value is less than zero, and the initial value is less than the limit.

However, older implementations of Fortran (for example, some FORTRAN 66 processors) always execute a `DO` loop at least once. The `+onetrip` option provides compatibility with those nonstandard implementations.

The default is `+noonetrip`.

`+O[no]openmp`

`+Oopenmp` allows users to enable the OpenMP Directives. `+Onoopenmp` will disable the OpenMP directives. `+O[no]openmp` is accepted at all opt levels. The default is `+Onoopenmp`.

`+pa`

`+pa` compiles an application for routine-level profiling (for CXperf support).

NOTE `+pa` is ignored when the HP Fortran compiler generates position-independent code (PIC). The following options cause +pa to be ignored: `+pic=short`, `+pic=long`, `+z` and `+Z`.

`+pal`

`+pal` compiles the application for routine- and loop-level profiling (for CXperf support).

NOTE `+pal` is ignored when the HP Fortran compiler generates position-independent code (PIC). The following options cause +pa to be ignored: `+pic=short`, `+pic=long`, `+z` and `+Z`.

`+pic={short|long|no}`

`+pic` generates object code that can be added to a shared library. Object code generated with this option is position-independent code (PIC). All addresses are either pc-relative or indirect references.

The argument—`short` or `long`—specifies the allocated size of the data linkage table. Normally you would specify `+pic=short` to generate PIC. Use `+pic=long` when the linker issues an error message indicating data linkage table overflow. Specifying `+pic=long` causes the compiler to allocate additional space for more imported symbols.

The default, `+pic=no`, causes the compiler to generate absolute code.

The `+z` option performs the same function as `+pic=short`, and the `+Z` option performs the same function as `+pic=long`.

`+[no]ppu`

`+ppu` appends underscores to external names, including subroutines, functions, and common blocks (for example, `int_sum_` rather than the default `int_sum`).

The default is `+noppu`.

NOTE Mixed languages programs are affected by the +ppu option. C languages references to Fortran routines and COMMON blocks require a trailing underscore when the Fortran code is compiled with +ppu. +noppu may be used in wide mode to avoid trailing underscores.

`+pre_include=`*file*

`+pre_include` causes the compiler to prepend the code in *file* before any compilation occurs. This option can appear more than once—each specifying different *files*—on the same command line.

`+[no]prof`

`+prof` prepares object files for profiling with `prof`. The default is `+noprof`.

The `-p` option can be used to perform the same function as `+prof`.

`prof` is provided as part of the "HP-UX General Programming Tools" product (see *prof*(1)).

`+r8`

`+r8` changes 4-byte real constants, intrinsics, and user variables to 8-byte reals (rather than the 4-byte default).

`+real_constant={single|double}`

`+real_constant=single` treats all single-precision numerical constants as single-precision, and the `+real_constant=double` option treats all single-precision numerical constants as double-precision. The default is `+real_constant=single`.

The `-R4` and `-R8` options can be used to perform the same function.

`+[no]save`

`+save` forces static storage for all local variables. This option provides a convenient path for porting older Fortran programs that may depend on static allocation of memory. (Variables in static storage retain their values between invocations of the program units in which they are declared). The `+save` option causes all uninitialized variables to be initialized to zero. The default is `+nosave`.

If you explicitly declare a variable with the `AUTOMATIC` attribute, the attribute overrides the `+save` option.

The `+save` command-line option inhibits many of the optimizations performed by the compiler. Generally, you will get better performance with the `+Oinitcheck`

option, which also sets uninitialized variables to zero but is more selective than `+save`; see "Using optimization options" on page 52.

The `-K` option can be used to perform the same function as `+save`.

`+[no]shared`

`+noshared` causes the output file from the linker to be marked `unshared`. The default, `+shared`, is to mark the output file as `shared`.

The `-n` option performs the same function as `+shared`, and the `-N` option performs the same function as `+noshared`.

`+sharedlibF90`

`+sharedlibF90` allows users to link the shared version of `libF90` or `libF90_parallel` from `/usr/lib`. This resolves potential issues with the Fortran 90 driver trying to link with the shared versions of `libF90`.

`+[no]signedzero`

`+[no]signedzero` enables signed-zero support. This option forces a floating point value of negative zero that appears as a formatted output list item to be represented in the output record with a leading "-". This option also changes the behavior of the `SIGN` intrinsic. The default is `+signedzero`.

`+source={fixed|free|default}`

`+source` tells the compiler that source files are in either fixed or free form. The default (`+source=default`) is free form for `.f90` source files and fixed form for `.f` and `.F` source files.

`+[no]strip`

`+strip` causes the linker to strip symbol table information from the executable program. This option is incompatible with the `-g` option. The default is `+nostrip`.

The `-s` option can be used to perform the same function as `+strip`.

`-t`*x*`,`*path*

`-t` looks in *path* for the subprocess identified by *x* and substitutes it for the default subprocess. *x* can be one or more identifiers indicating the subprocesses.

This option works in two modes:

- If *x* is a single identifier and *path* ends in with a slash (/), *path* represents the directory with the new subprocess, and the name of the subprocess is the standard name. If *path* ends in a filename, it is the name of the subprocess.
- If *x* is a set of identifiers, *path* is a directory that holds the subprocesses identified in *x*. The subprocesses in *path* have their standard names.

Table 13 lists the identifiers for *x*, the subprocesses each indicates, and the standard subprocess name.

The following example of the `-t` option tells the compiler to pass the source files to the K&R version of the C preprocessor for preprocessing:

```
-tp,/usr/ccs/lbin/cpp
```

Table 13 Values for the `-t` option x subprocesses

Value	Subprocess	Standard name
`a`	Assembler	`as`
`c`	Compiler	`f90com`
`e`	Debug file	`end.o`
`l`	Linker	`ld`
`p`	C preprocessor	`cpp`
`s`	Start-up file	`crt0.o`, `gcrt0.o`, `mcrt0.o`

`+[no]ttybuf`

`+ttybuf` controls tty buffering, using buffered output. `+nottybuf` uses unbuffered output. The default is buffered output (`+ttybuf`). The `+ttybuf` option forces buffered output even on systems whose default is unbuffered output.

The `+[no]ttybuf` option is recognized only when the main program is a Fortran program. If the main program is written in another language, use the `TTYUNBUF` environment variable (see *f90*(1)).

The +nottybuf option is incompatible with certain BSD 3F library routines. When it is used on the same command line with the +U77 option, the compiler will warn of a potential tty buffering conflict.

-U *name*

-U undefines or removes any initial definition of *name* in the C preprocessor (cpp). See the *cpp*(1) in the *HP-UX Reference* for details.

+[no]U77

+U77 option invokes support for the BSD 3F library, libU77, which provides an HP Fortran interface to some of the libc system routines. To call routines in this library, you must compile and link with +U77. For information about these routines, see the *HP Fortran Programmer's Reference*.

If +noU77 (the default) is specified or if +U77 is not specified, the compiler treats libU77 routine names as ordinary external names with no name mapping. If the name is not present in one of the libraries linked to, the linker emits an error message because of an unsatisfied symbol. If the libU77 name is the same as a libc name, the name might resolve to a libc name. This situation does not cause an error at compile time, but can produce unpredictable results.

+[no]uppercase

+uppercase uses uppercase for external names. The default, +nouppercase, is to convert external names to lowercase.

If you need to control the case of specific names, use the HP ALIAS directive, as described in "HP ALIAS" on page 190.

+usage

+usage lists and briefly describes all of the command-line options currently supported by the HP Fortran compiler. No compile occurs.

-v

-v enables the verbose mode, producing a step-by-step description of the compilation process on the standard error output.

+version

`+version` displays compiler version information only; no compilation occurs.

`-w`

`-w` suppresses warning messages. If this option is omitted, warnings are sent to standard error.

`+what`

`+what` prints what string for the Fortran 90 driver, providing version and patch numbers.

`-W`*x*,*arg1*,*arg2*,...,*argN*

`-W` causes *arg1* through *argN* to be handed off to subprocess *x*. Each *arg* takes the form:

-*option*[,*value*]

where *option* is the name of an option recognized by the subprocess and *value* is a separate argument to *option*, where necessary. The values that *x* can assume are listed in Table 14.

For example, the following option tells the linker to print a trace of each input file as `ld` processes it:

```
-Wl,-t
```

The next example passes the `-a` shared option to the linker, causing it to select shared libraries for linking.

```
-Wl,-a,shared
```

Table 14 Values for the -W option

Value	Meaning
a	Assembler
c	Compiler
l	Linker
p	C preprocessor

`+Z` *see* `+pic=long` in this chapter for a description. Note that when creating 64-bit shared executables (such as when `+DA2.0W` is specified), the `+Z` option is on by default. This is the only PIC option supported for 64-bit executables.

NOTE To not generate position-independent code for 64-bit executables, specify the -W1, -noshared option:

`+z` *see* `+pic=short` in this chapter . If +z is specified when creating 64-bit code, it instead maps to `+Z`.

Using optimization options

The options described in this section allow you to control the different optimizations that the compiler can apply to your program. These options fall into two categories:

- Options that control classes of optimization (for example, optimizations that affect code size)
- Options that control specific optimizations (for example, inlining)

The following subsections describe the options in both categories. For information about the options that control levels of optimization, see the description of the `+O`*n* option in the "Option descriptions" on page 24. The `+O[no]info` option, which provides compile-time information about the optimization process, is described in the same section.

NOTE You can insert (or remove) underscore characters in the names of any of the optimization options to improve their readability. The compiler will recognize the option name with or without underscores.

Reviewing general optimization options

The following options allow you to control how optimization affects code size, compilation time, runtime performance, and other user-visible effects. The syntax for using these options is:

`+O[no]`*optimization*

where *optimization* is a parameter that specifies the class of optimization to apply to your program. The different parameters are described below. The prefix `no` negates the effect of optimization.

Except for `+Oall`, the options do not override a specified level of optimization, nor do they imply a particular level. (The `+Oall` option automatically invokes the highest level of optimization.) To use any of these options you must also include the `+O`*n* option on the same command line, where *n* specifies the level at which the type of optimization is effective. Thus, if you wish to apply all optimizations available at level 3 except those that might significantly increase code size, you would use the command line:

```
f90 +O3 +Osize my_prog.f90
```

If an option is mistakenly used at a level at which the corresponding optimization is not performed, the compiler will issue a warning message.

The defaults specified in the following descriptions are in effect only at the specified optimization levels, unless stated otherwise.

`+O[no]aggressive`

`+Oaggressive` enables optimizations that can result in significant performance improvement but can also change a program's behavior. This option is only effective at optimization level 2 or higher.

The `+Oaggressive` option performs optimizations invoked by the following options:

- `+Oentrysched`
- `+Onofltacc`
- `+Onoinitcheck`
- `+Ovectorize`

The `+Oaggressive` option is incompatible with `+Oconservative`.

The default is `+Onoaggressive`.

`+O[no]all`

`+Oall` performs maximum optimization, including aggressive optimizations and optimizations that can significantly increase compile time and memory usage. The `+Oall` option automatically invokes the highest level of optimization.

The default is `+Onoall`.

`+O[no]conservative`

`+Oconservative` causes the optimizer to make conservative assumptions about the code when optimizing it. This option is only effective at optimization level 2 or higher.

The `+Oconservative` option sets the following options:

- `+Onofltacc`
- `+Onomoveflops`

- `+Oparmsoverlap`

  Use `+Oconservative` when conservative assumptions are necessary due to the coding style, as with nonstandard-conforming programs. Note that it is incompatible with `+Oaggressive`.

  The `+Onoconservative` option relaxes the optimizer's assumptions about the target program.

  The default is `+Onoconservative`.

`+O[no]limit`

`+Olimit` suppresses optimizations that significantly increase compilation time or that can consume large amounts of memory at compile time. This option is only effective at optimization level 2 or higher.

The `+Onolimit` option allows optimizations to be performed regardless of their effect on compilation time or memory usage.

The default is `+Olimit`.

`+O[no]size`

`+Osize` suppresses optimizations that significantly increase code size. This option is only effective at optimization level 2 or higher.

The `+Onosize` option permits optimizations that can increase code size.

The default is `+Onosize`.

Fine-tuning optimization options

The following options allow you to fine-tune the optimization process by providing control over the specific techniques that the optimizer applies to your program. The syntax for using these options is

`+O[no]`*optimization*

where *optimization* is a parameter that specifies an optimization technique to apply to your program. The different parameters are described below. The prefix `no` negates the effect of optimization.

The options do not override a specified level of optimization, nor do they imply a particular level. To use any of these options you must also include the `+O`*n* option on the same command line, where *n* specifies the level at which the type of optimization can be performed.

For example, if you find that the optimizer is causing your program to produce different floating-point results from those produced by the unoptimized program, you could use the following command line to suppress optimizations that affect floating-point calculations:

```
f90 +O3 +Onomoveflops +Ofltacc my_prog.f90
```

If an option is mistakenly used at a level for which the corresponding optimization is not performed, the compiler will issue a warning message.

The defaults given in the following descriptions are in effect only at the specified optimization levels, unless stated otherwise.

`+O[no]cache_pad_common`

`+Ocache_pad_common` can improve program performance by padding common blocks to avoid cache collisions. Cache-line collisions occur when the difference between the addresses of two data points is a multiple of the cache size. By inserting empty space between large variables (for example, arrays), the optimizer ensures that they do not start at nearby addresses, where the possibility of a cache collision is greater. This option is only effective at optimization level 3 or higher.

Note the following precautions when using this option:

- All program modules that reference the common block must be compiled with the `+Ocache_pad_common` option.
- Each common block in the program should have the same layout in all program units within which it is declared. If the layouts are different, they must be fully independent—that is, they must not pass values between them.

The default, `+Onocache_pad_common`, disables padding.

`+O[no]dataprefetch`

`+Odataprefetch` causes the optimizer to insert instructions within innermost loops to explicitly prefetch data from memory into the data cache. Data prefetch instructions will be inserted only for data structures referenced within innermost loops using

simple loop varying addresses—that is, in a simple arithmetic progression. This option is only effective at optimization level 2 or higher. It is only available for PA-RISC 2.0 targets.

Use this option for applications that have high data cache miss overhead.

The default is `+Onodataprefetch`.

`+O[no]entrysched`

`+Oentrysched` allows the optimizer to perform instruction scheduling on a subprogram's entry and exit code sequences. This option is only effective at optimization level 1 or higher.

The option can change the behavior of programs that perform exception-handling or that handle asynchronous interrupts.

The default is `+Onoentrysched`.

`+O[no]fastaccess`

`+Ofastaccess` improves execution time by speeding up access to global data items. You can use this option at any level of optimization.

Note that the `+Ofastaccess` option may increase link time.

The default is `+Onofastaccess` at optimization levels 1, 2, and 3; and `+Ofastaccess` at optimization level 4.

`+O[no]fltacc`

`+Onofltacc` enables optimizations that follow the rules of algebra but may change the order of expression evaluation. For example, if `a`, `b`, and `c` are floating-point variables, the expressions `(a + b) + c` and `a + (b + c)` may give slightly different results due to roundoff.

The `+Onofltacc` option also enables the fusion of adjacent multiply and add operations—resulting in Fused Multiply-Add (FMA). FMA is implemented by the `FMPYFADD` and `FMPYNFADD` instructions and is only available on PA-RISC 2.0 systems. (At optimization level 2 or higher, FMA occurs by default.) FMA improves performance but occasionally produces

results that may differ in accuracy from results produced by code where fusion has not occurred. In general, the differences are slight.

`+Ofltacc` disables optimizations that change the order of expression evaluation and therefore may affect the accuracy of the result. The `+Ofltacc` option also disables fusing.

Table 15 identifies the different actions taken by the optimizer, according to whether you specify `+Ofltacc`, `+Onofltacc`, or neither option. In all cases, the table assumes that you are compiling at optimization level 2 (`+O2`) or higher.

Table 15 Optimizations performed by `+O[no]fltacc`

+O[no]fltacc	Expression reordering?	FMA?
	No	Yes
`+Ofltacc`	No	No
`+Onofltacc`	Yes	Yes

`+O[no]info`

`+Oinfo` causes the compiler to display informational messages about the optimization process. The `+Oinfo` option provides feedback that can help you to determine whether the compiler optimized time-critical sections of your program. It can be used at any level of optimization but is most useful at level 3.

Currently, this option provides feedback for the following optimizations:

- Cloning, the replacement of a call to a routine by a call to a clone, which is a copy of the routine with changes specific to that call site.
- Inlining.
- Loop transformations to improve cache performance.
- Vectorization.

The default, `+Onoinfo`, disables the display of informational messages about optimization.

`+O[no]initcheck`

The initialization checking feature of the optimizer has three possible states: on, off, or unspecified. When this option is specified in the on state (`+Oinitcheck`), the optimizer initializes to zero any local, nonarray, nonstatic variables that are uninitialized with respect to at least one path leading to a use of the variable.

When `+Onoinitcheck` is specified, the optimizer issues warning messages when it discovers definitely uninitialized variables, but does not initialize them.

When this option is unspecified, the optimizer initializes to zero any local, scalar, nonstatic variables that are definitely uninitialized with respect to all paths leading to a use of the variable.

This option is only effective at optimization level 2 or higher.

`+O[no]inline`

`+Oinline` makes all subprograms eligible for inlining. This option is only effective at optimization level 3 or higher.

The `+Onoinline` option disables inlining for all subprograms in your program.

The default is `+Oinline` at optimization level 3 and `+Onoinline` at the lower levels.

`+Oinline_budget=`*n*

`+Oinline_budget` enables the optimizer to perform more aggressive inlining.

This option has the following syntax:

`+Oinline_budget=`*n*

where *n* is an integer in the range 1 - 1000000 that specifies the level of aggressiveness, as listed in Table 16 on page 59.

The `+Onolimit` and `+Osize` options also affect inlining. Specifying the `+Onolimit` option has the same effect as specifying `+Oinline_budget=200`. The `+Osize` option has the same effect as `+Oinline_budget=1`.

Note, however, that the `+Oinline_budget` option takes precedence over both of these options. This means that you can override the effect of `+Onolimit` or `+Osize` option on inlining by specifying the `+Oinline_budget` option on the same command line.

This option is only effective at optimization level 3 or higher.

Table 16 Values for the `+Oinline_budget` option

Values for *n*	Meaning
= 100	Default level of inlining.
> 100	More aggressive inlining. The optimizer is less restricted by compilation time and code size when searching for eligible routines to inline.
2 - 99	Less aggressive inlining. The optimizer gives more weight to compilation time and code size when determining whether to inline.
= 1	Only inline if it reduces code size.

`+O[no]libcalls`

invokes millicode versions of a number of frequently called intrinsic functions; see Table 17 on page 60. Millicode routines have very low call overhead and provide no error-handling. Use this option to improve the performance of selected library routines only when your program does not depend upon exception-handling.

The default is `+Onolibcalls` at optimization levels 0 and 1; at optimization level 2 or higher, the default is `+Olibcalls`.

Table 17 Millicode versions of intrinsic functions

acos	cos	pow
asin	exp	sin
atan	log	tan
atan2	log10	

`+O[no]loop_block`

`+O[no]loop_block` enables or disables blocking of eligible loops for improved cache performance. The `+Onoloop_block` option disables both automatic and directive-specified loop blocking.

`+O[no]loop_transform`

`+Oloop_transform` enables transformation of eligible loops for improved cache performance. The most important transformation is the interchange of nested loops to make the inner loop unit stride, resulting in fewer cache misses. `+Onoloop_transform` disables transformation of eligible loops. The default is `+Oloop_transform`.

`+O[no]loop_unroll[=`*factor*`]`

`+Oloop_unroll` turns on loop unrolling. *factor* is the unroll factor that controls the code expansion. The default unroll factor is 4; that is, four copies of the loop body. By experimenting with different factors, you may improve the performance of your program. This option is only effective at optimization level 2 or higher.

The default is `+Oloop_unroll=4`.

`+O[no}loop_unroll_jam`

`+loop_unroll_jam` enables loop unrolling and jamming. `+Onoloop_unroll_jam` (the default) disables both automatic and directive-specified unroll and jam. Loop unrolling and jamming increases register exploitation.

`+O[no]multiprocessor`

`+Omultiprocessor` tells the compiler to appropriately optimize several different processes on multiprocessor machines. The optimizations are those appropriate for

executables and or shared libraries. `+Onomultiprocessor`, the default, disables the optimization of more than one process running on a multiprocessor machine.

`+O[no]moveflops`

`+Omoveflops` allows the optimizer to move conditional floating-point instructions, enabling other optimizations to occur. This option is only effective at optimization level 2 or higher.

The behavior of floating-point exception handling may be altered by this option.

Using `+Onomoveflops` is recommended if floating-point traps are enabled and you do not want the behavior of floating-point exceptions to be altered by the relocation of floating-point instructions, as when your program uses the `ON` statement. The default is `+Omoveflops`.

`+O[no]parallel`

`+Oparallel` causes the compiler to transform eligible loops for parallel execution on multiprocessor machines. This option is effective only at optimization level 3 or higher.

If you link separately from the command line and compile the program with the `+Oparallel` option, you must link with the `f90` command and specify the `+Oparallel` option to link in the correct runtime support.

The `+Onoparallel` option disables parallelization for the target program. It is the default at all levels of optimization.

NOTE The `+Oparallel` option should not be used for programs that make explicit calls to the kernel threads library.

`+Oparallel_intrinsics`

`+Oparallel_intrinsics` links in the parallel version of many of the Fortran intrinsics located in libF90_parallel.

`+O[no]parmsoverlap`

`+Oparmsoverlap` causes the optimizer to assume that the actual arguments of function calls overlap in memory, thus preventing any optimizations that violate this assumption. This option is only effective at optimization level 2 or higher.

Use the `+Onoparmsoverlap` option with programs that conform to the standard requirement that parameters must not overlap.

The default is `+Onoparmsoverlap`.

`+O[no]pipeline`

`+Opipeline` enables software pipelining. This option is only effective at optimization level 2 or higher.

Use `+Onopipeline` (disable software pipelining) to conserve code space.

The default is `+Opipeline`.

`+O[no]procelim`

When `+Oprocelim` is specified, procedures that are not referenced by the application are eliminated from the output executable file. When `+Onoprocelim` is specified, procedures that are not referenced by the application are not eliminated from the output executable file. You can use this option at any level of optimization.

Use `+Oprocelim` to reduce the size of the executable file, especially when optimizing at levels 3 and 4, when inlining can remove all calls to some routines.

The default is `+Onoprocelim` at levels 0-3, and `+Oprocelim` at level 4.

`+O[no]regreassoc`

`+Onoregreassoc` disables register reassociation. This option is only effective at optimization level 2 or higher.

Use `+Onoregreassoc` to disable register reassociation in the rare case that this optimization degrades performance.

`+Oregreassoc` is the default

`+O[no]report` `+Oreport` specifies the contents of the Optimization Report.

`+O[no]vectorize`

`+Ovectorize` causes the compiler to replace certain loops with calls to the math library. This option is only effective at optimization level 3 or higher.

If you link separately from the command line and you compiled with the `+Ovectorize` option, you must ensure that the link line causes the math library to be searched.

`+Onovectorize` is the default.

Filenames

The `f90` command accepts files with any of the **filename extensions** listed in Table 18. The table also describes the meaning each name has for the `f90` command. Files with names other than those listed in the table are passed to the linker.

Table 18 Filenames recognized by f90

Filenames	Meaning
file`.f90`	Free-form Fortran source code; processed by the compiler.
file`.f`	Fixed-form Fortran source code; processed by the compiler.
file`.F`	Fixed-form Fortran source code; first processed by the C preprocessor (`cpp`), then by the compiler.
file`.i90`	Free-form output from the C preprocessor (if the source file ends in `.f90`); processed by the compiler.
file`.i`	Fixed-form output from the C preprocessor (if the source file ends in `.F` or `.f`); processed by the compiler.
file`.o`	Object code; passed to the linker (`ld`).
file`.s`	Assembly language code; passed to the assembler (`as`).

NOTE The compiler generates a **.mod** file for each file that defines a Fortran **module**. It also reads the `.mod` files when compiling source files that use modules. Do not specify `.mod` files on the command line. If you do, the compiler will pass them to the linker, which will try (and fail) to link them into the executable. For more information about `.mod` files, see “Compiling programs with modules” on page 72.

Linking HP Fortran programs

This section discusses how to link object files and covers the following topics:

- The advantages of using the `f90` command for linking as well as for compiling
- How to link libraries, including shared libraries
- How to establish the search rules used by the linker when it looks for libraries

For more information about the linker, refer to *Programming on HP-UX* and to the *ld*(1) man page.

Linking with f90 vs. ld

By default, the `f90` command both compiles and links, producing an executable program. You can modify this behavior with the `-c` option, which causes `f90` to compile only, writing the object files (if the compilation is successful) in the current working directory. If the command line contains object files only, `f90` passes them to the linker (`ld`) for linking into the executable program. In other words, you can use the `f90` command to compile and link in one command line or in separate command lines. You do not need to invoke the `ld` command separately.

In fact, we recommend that you use the `f90` command whenever you link HP Fortran object files and that you use the same command line for linking as for compiling.

When you use the `f90` command to compile and link in the same command line, the driver passes certain information—search paths, library names, and options—to the linker. If you use the `ld` command to link separately, you must specify this same information on the `ld` command line. Not doing so can cause the link to fail. Using the same `f90` command line to link as you use to compile avoids the problem of passing insufficient or incorrect information to the linker.

To see what information `f90` passes to the linker, compile with the `-v` option (verbose mode). Here is the `hello.f90` program (listed in "Compiling with the f90 command" on page 20) compiled in verbose mode. The lines are numbered for the convenience of referencing:

```
1   $ f90 -v hello.f90
2   /opt/fortran90/lbin/f90com -cm -w90 -nbs -auto
          -WB -hp\"-Oq00,al,ag,cn,Lm,sz,Ic,vo,lc,mf,po,es,rs,sp,
          in,vc,pi,fa,pe,Rr,Fl,pv,pa,nf,cp,lx,st,ap,Pg,
          ug,lu,dp,fs,bp,wp\!\" hello.f90
3   hello.f90
4      program MAIN
5      external subroutine HELLO

6   7 Lines Compiled
7   LPATH is: /opt/fortran90/lib/pa1.1:/usr/lib/pa1.1:
          /opt/fortran90/lib:/usr/lib:/opt/langtools/lib
8   /usr/ccs/bin/ld -x /opt/langtools/lib/crt0.o hello.o
          /opt/fortran90/lib/libF90.a -lcl -lc -lisamstub
```

- Line 1 is the `f90` command line.
- Line 2 is the information `f90` passes to the compiler, including the full pathname of the compiler, the name of the source file (`hello.f90`), and the internal names of the option settings as determined by the defaults and the `f90` command line.
- Lines 3 - 6 show the progress of the compilation; line 6 indicates that the compilation was successful.
- Line 7 displays the value to which `f90` has defined the `LPATH` environment variable. If you use the `ld` command to link `hello.f90`, you must define `LPATH` on the command line before invoking the linker. See "LPATH environment variable" on page 87.
- Line 8 is the command line that `f90` passes to the linker (`ld`). If you use the `ld` command to link `hello.f90`, the command line should be similar to the one shown here.

As noted in the comments on lines 7 and 8, compiling and linking `hello.f90` successfully using both the `f90` and `ld` commands requires three command lines:

```
$ f90 -c hello.f90                                    # compile

$ export LPATH=/opt/fortran90/lib/pa1.1:/usr/lib/pa1.1:\
/opt/fortran90/lib:/usr/lib:/opt/langtools/lib        # set LPATH

$ ld -x /opt/langtools/lib/crt0.o hello.o \
/opt/fortran90/lib/libF90.a -lcl -lc -lisamstub       # link
```

The command line to set LPATH in the csh is:

```
$ setenv LPATH /opt/fortran90/lib/pa1.1:/usr/lib/
pa1.1:\
/opt/fortran90/lib:/usr/lib:/opt/langtools/lib
# set LPATH
```

For more information about the linker, see the *ld*(1) man page. For a list of f90 options that you can use to control the linker, see Table 6 on page 13. To pass linker options from the f90 command line to the linker, use the -Wl option (for an example, see "Linking to shared libraries" on page 69). The *HP Fortran Programmer's Reference*, fully describes the -Wl option.

Linking to libraries

When you use the f90 command to create an executable program, the linker looks in the libraries listed in Table 19 to resolve references. By default, the linker uses the shared libraries, if available. For information about shared libraries, see "Linking to shared libraries" on page 69.

The libisamstub library is provided as a tool for **migrating** HP FORTRAN 77 programs that call ISAM routines. The ISAM library is not available with HP Fortran, but the stub library allows the linker to resolve references to ISAM routines in HP FORTRAN 77 programs.

Table 19 Libraries linked by default

Library	Contents
/usr/lib/libc1.a	Archive version of HP Fortran runtime library
/usr/lib/libc1.sl	Shared version of HP Fortran runtime library
/opt/fortran90/lib/libF90.a	Archive library of array intrinsic procedures
/usr/lib/libc.a	Archive library of intrinsic procedures and system routines
/usr/lib/libc.sl	Shared library of intrinsic procedures and system routines,
/opt/fortran90/lib/libisamstub.a /opt/fortran90/lib/libisamstubs.a	Archive libraries of stubs to satisfy ISAM references
/usr/lib/libisamstub.sl	Shared library of stubs to satisfy ISAM references

When the linker finds a reference in your program to a name that is not defined in the program (for example, the `DOT_PRODUCT` intrinsic), it looks to resolve it in the default libraries. If it cannot find the name in the default libraries, the link will fail unless the command line specifies additional, nondefault libraries. This section discusses how to link to nondefault libraries (including shared libraries) and library search rules.

Linking to nondefault libraries

The `-l` option enables you to specify other libraries for linking, in addition to the default libraries listed in Table 19. The syntax for this option is:

`-l`*x*

where *x* is a sequence of characters that completes a library name of the form `/lib/lib`*x*`.a` or `/usr/lib/lib`*x*`.a`. For example, `-lm` specifies the math library, `/usr/lib/libm.a`. (The `.a` extension indicates an **archive library**. You can also link to **shared libraries**, which have the `.sl` extension; see "Linking to shared libraries" on page 69.)

The `-l` option is order-sensitive: when the linker finds a reference in an object file that it cannot resolve in the default libraries, it searches the libraries (if any) specified *after* the file on the command line. For example, the following command line tells the linker to look for unresolved references in the math library as well as the default libraries:

```
$ f90 prog.f90 -lm
```

You can also link a library to your program by specifying its name *after* the name of the source file that references it, as follows:

```
$ f90 prog.f90 /usr/lib/libm.a
```

This form of the command line is useful for linking libraries that do not conform to the naming convention required by the `-l` option or that reside in a directory other than `/lib` or `/usr/lib`. As with the `-l` option, the library name must follow the name of the source file that references it. For example, the following command line links `prog.f90` with the library `my_routines`, both of which reside in the current working directory:

```
$ f90 prog.f90 my_routines
```

If your program calls routines in a library but the linker is unable to resolve the references, compile with the `-Wl,-v` option. The `f90` command passes `-v` to the linker, causing it to process in verbose mode. The verbose information includes:

- The names of the libraries that the linker is searching. This information can confirm that the linker is searching the correct libraries.
- The names of the object files selected by the linker to resolve the references. The linker may have found the same name in another library and resolved the reference there.

Many library-related problems are owing to a misplaced `-l` on the command line. The `-L` option (discussed in "Library search rules" on page 70) is also order-sensitive and can cause similar problems.

Additional HP Fortran libraries

HP Fortran provides the following two libraries you can link with Fortran programs:

- `/opt/fortran90/lib/libU77.a`: The BSD 3f (**libU77**) library, which provides a Fortran interface to some of the `libc` system routines. Programs that reference routines in this library must be compiled with the `+U77` option. For information about porting Fortran programs that reference `libU77` routines, see "Migrating to HP Fortran" on page 201.
- `/opt/fortran/lib/libblas.a`: The **Basic Linear Algebra Subroutine (BLAS) library**, which provides routines that perform common vector and matrix operations. Programs that reference routines in this library must be compiled with the `+lblas` option. For more information, see "Calling BLAS library routines" on page 152.

Both the `libU77` and BLAS libraries are described in the *HP Fortran Programmer's Reference*.

Linking to shared libraries

Many HP Fortran libraries as well as HP-UX libraries exist in both shared and archive versions, as indicated by the library extension name (`.sl` or `.a`). For example, there are both shared and archive versions of the HP Fortran runtime library, `/usr/lib/libcl.sl` and `/usr/lib/libcl.a`.

The difference between a shared library and an archive library is that the linker does not actually link the code in a shared library with your program. Instead, any references that your program makes to entities in

the shared library are resolved at load-time, when the library is loaded into the executable program's address space. By contrast, code in the archive library is copied to the executable program file.

The advantages of linking shared libraries are:

- The executable is smaller than it would be if linked with an archive file because the executable file is *incomplete*—it doesn't include code from the library.
- Using shared libraries ensures that you always get the most recent version of the library. If you link with an archive version, you get the version that was available at link-time. If, later on, you want a more recent version of the library, you must re-link your program with that library.

The disadvantage of linking with a shared library is that it creates a dependency between the library and the program; both the shared library and the program must always be installed together on the same system. By contrast, linking with an archive library makes the executable program independent of the library to which it was linked. Also, programs that make frequent calls to library routines may run more slowly when linked to shared libraries.

By default, the linker selects the shared version of a library, if one is available; otherwise, it selects the archive version. To force the linker to select archive libraries, specify the `-Wl,-a,archive` option on the `f90` command line. `f90` passes the arguments to the `-Wl` option (`-a` and `archive`) to the linker. This option must appear *before* the names of any libraries also specified on the command line. The following command line compiles `prog.f90` and links it with the archive versions of the default libraries as well as with the math library (as specified by the `-lm` option):

```
$ f90 -Wl,-a,archive prog.f90 -lm
```

For information about the linker's `-a` option, see the *ld*(1) man page. For more information about shared libraries, see "Creating shared libraries" on page 78.

Library search rules

When you use the `-l` option to specify the name of a library, the linker searches for the library in the directories specified by the `LPATH` environment variable. The `f90` command sets this variable so that the

linker looks first in `/opt/fortran90/lib`, then in `/usr/lib`. You can specify another directory to search by setting `LPATH` yourself; see "LPATH environment variable" on page 87.

Alternatively, you can use the `-L`*directory* option to direct the linker to search *directory* before it looks anywhere else to resolve references. For example, the following command line:

```
$ f90 -L/my_libs prog.f90 -lstuff
```

causes the linker to search for libraries (including `libstuff.sl` and `libstuff.a`), starting with the directory `/my_libs` and then looking in `/opt/fortran90/lib` and `/usr/lib`.

Special-purpose compilations

The default behavior of the HP Fortran compiler has been designed to handle typical compilations. Most applications should require no more than a few of the `f90` options to compile successfully (see Table 7 on page 22 for a list of commonly used options).

However, the compiler can also meet the needs of more specialized compilations. This section explains how to use the `f90` command for the following purposes:

- To compile programs that contain Fortran modules.
- To compile programs that will execute on different PA-RISC machines.
- To create object files for shared libraries.
- To process source files that contain C preprocessor directives.
- To create demand-loadable programs.
- To create shareable executable programs.
- To compile 32-bit programs in 64-bit mode.

Compiling programs with modules

One of the features of standard Fortran is the *module*, a program unit that facilitates shared access to data and procedures. Modules are fully described in the *HP Fortran Programmer's Reference*.

A benefit to using modules is that they provide interface information to the compiler, allowing it to catch mismatch errors between (for example) dummy arguments and actual arguments. When the HP Fortran compiler processes a file that defines a module, it generates a `.mod` file with the interface information. Later, when the compiler processes a file that uses the module, it reads the `.mod` file and checks that module entities that are referenced in the *using* file correctly match the information in the `.mod` file.

To make the `.mod` files available to the compiler, you must therefore compile the files that define modules before the files that use modules. Likewise, if you make changes to a file that defines a module, you must recompile that file as well as any files that use the module, in that order.

Also, if a module is defined and used in the same file, the definition must lexically precede any `USE` statements that reference the module. This requirement allows the compiler to generate the `.mod` file first, so that it can resolve the references in any `USE` statements.

This section discusses the following topics:

- How to compile a program that uses modules
- How to design makefiles to work with modules
- How to use the `-I` and `+moddir` options to manage `.mod` files

Examples

Consider, for example, a program that consists of three files: `main.f90`, `code.f90`, and `data.f90`. The main program unit is in `main.f90`, as follows.

main.f90

```
PROGRAM keep_stats
  ! stats_code contains module procedures for operating
  !   on statistical database
  USE stats_code
  INTEGER :: n

  ! print prompt, using nonadvancing I/O
  WRITE (*, FMT='(A)', ADVANCE='NO') 'Enter an integer '// &
               '(hint:  77 is current average):  '
  READ *, n
  IF (n == 0) THEN
    PRINT *, 'But not that one.'
  ELSE
    CALL update_db(n)
    IF (n >= get_avg()) THEN  ! get_avg is in stats_code
        PRINT *, 'Average or better.'
    ELSE
        PRINT *, 'Below average.'
    END IF
  END IF
END PROGRAM keep_stats
```

The first specification statement (`USE`) in the main program indicates that it uses the module `stats_code`. This module is defined in `code.f90`, as follows:

code.f90

```
! stats_code:  a (partial!) package of module procedures for
!   performing statistical operations
MODULE stats_code

  ! shared data to be used by procedures declared below
  USE stats_db

  CONTAINS  ! module procedures

        ! update_db:  updates shared variables in module stats_db
        SUBROUTINE update_db (new_item)
          INTEGER :: new_item

          n_items = n_items +1
          item(n_items) = new_item
          sum = sum + new_item
        END SUBROUTINE update_db

        ! get_avg:  returns arithmetic mean
        INTEGER FUNCTION get_avg ()
          get_avg = sum / n_items
        END FUNCTION get_avg
END MODULE stats_code
```

This program unit also begins with a `USE` statement, which identifies the module it uses as `stats_db`. This module is defined in `data.f90`, as follows:

data.f90

```
! stats_db:  shared data declared here
MODULE stats_db
  INTEGER, PARAMETER :: size = 100  ! max number of items in
array

  ! n_items, sum, and item hold the data for statistical analysis
  INTEGER :: n_items, sum
  INTEGER, DIMENSION(size) :: item

  ! the initializations are just to start the program going
  DATA n_items, sum, item/3, 233, 97, 22, 114, 97*0/
END MODULE stats_db
```

The use of modules in this program creates dependencies between the files because a file that uses a module that is defined in another file is dependent on that other file. These dependencies affect the order in which the program files must be compiled. The dependencies in the example program are:

- `main.f90` is dependent upon `code.f90`.
- `code.f90` is dependent upon `data.f90`.

These dependencies require that `data.f90` be compiled before `code.f90`, and that `code.f90` be compiled before `main.f90`. This order ensures that the compiler will have created each of the `.mod` files *before* it needs to read them.

The order of the source files listed in the following command line ensures that they will compile and link successfully:

```
$ f90 -o do_stats data.f90 code.f90 main.f90
```

During compilation, `f90` will create two `.mod` files, `STATS_CODE.mod` and `STATS_DB.mod`. These will be written to the current working directory, along with the object files and the executable program, `do_stats`. Following is a sample run of the executable program:

```
$ do_stats
Enter an integer (hint:  77 is current average):  77
 Average or better.
```

If instead of the preceding command line, the program had been compiled as follows:

```
$ f90 -o do_stats main.f90 data.f90 code.f90
```

the compilation would fail and `f90` would print the error message:

```
Error FCE37 : Module STATS_CODE not found
```

The compilation would fail because the compiler cannot process `main.f90` without `STATS_CODE.mod`. But the order in which the program files appear on the command line prevents the compiler from processing `code.f90` (and thereby creating `STATS_CODE.mod`) until *after* it has processed `main.f90`.

Compiling with `make`

If you use the `make` utility to compile Fortran programs, the description file should take into account the dependencies created by modules. For example, to compile the `do_stats` program using the `make` utility, the description file should express the dependencies as follows:

makefile

```
# description for building do_stats
do_stats :      main.o code.o data.o
        f90 -o do_stats main.o code.o data.o

# main.o is dependent on main.f90 and code.f90
main.o :        main.f90 code.o
        f90 -c main.f90
# code.o is dependent on code.f90 and data.f90
code.o :        code.f90 data.o
        f90 -c code.f90
# data.o is dependent only its source, data.f90
data.o :        data.f90
        f90 -c data.f90
```

Note that the dependencies correspond to the order in which the source files are specified in the following `f90` command line:

```
$ f90 -o do_stats data.f90 code.f90 main.f90
```

Assuming that you name the description file `makefile`, the command line to compile the program with `make` is:

```
$ make
```

Managing .mod files

By default, the compiler writes `.mod` files to the current working directory and looks there when it has to read them. The `+moddir=`*directory* and `-I` *directory* options enable you to specify different directories. The `+moddir` option causes the compiler to write `.mod` files in *directory*, and the `-I` option causes the compiler to search *directory* for `.mod` files to read. (The space character between `-I` and *directory* is optional.)

Using the example of the `do_stats` program, the following command line compiles (without linking) `data.f90` and writes a `.mod` file to the subdirectory `mod_files`:

```
$ f90 -c +moddir=mod_files data.f90
```

The command line:

```
$ f90 -c +moddir=mod_files -I mod_files code.f90
```

uses both the `+moddir` and `-I` options, as follows:

- The `+moddir` option causes `f90` to write the `.mod` file for `code.f90` in the subdirectory `mod_files`.
- The `-I` option causes `f90` to look in the same subdirectory for the `.mod` file to read when compiling `code.f90`.

The command line:

```
$ f90 -odo_stats -I mod_files main.f90 code.o
data.o
```

causes `f90` to compile `main.f90`, look for the `.mod` file in the subdirectory `mod_files`, and link all of the object files into an executable program named `do_stats`.

Compiling for different PA-RISC machines

When you compile an HP Fortran 90 program, the object code that the compiler generates by default is based on the PA-RISC model of the machine that is running the compiler. If your program will execute on a different PA-RISC model machine, the code may run less efficiently or (in the case of PA2.0 code that attempts to run on a PA1.1 machine) may not run at all.

Also, some libraries (for example, the math library) are available in different PA-RISC versions. By default, the compiler selects the version that is based on the PA-RISC model of the compiling machine. If your program will execute on a different model machine, it may not be linked with the appropriate libraries.

Compiling with the `+DA`*model* option ensures that the compiler generates code that is based on the architecture specified by *model* and that the linker selects libraries that are compatible with *model*. *model* must be one of the following:

- A PA-RISC version number—`1.1`, `2.0`, or `2.0W`. Use `+DA2.0W` to compile in 64-bit mode; see “Compiling in 64-bit mode” on page 85.
- A model number—for example, `750` or `870`.
- A PA-RISC processor name—for example, `PA7100` or `PA8000`.

- `portable`—code that is compatible across all models. Use `+DAportable` only if you want to ensure that your program will run on different models.

Use the `uname -m` command to learn the model of your machine, as follows:

```
$ uname -m
9000/879
```

Alternatively, you can use the `grep` command to look up the model number in the file `/opt/langtools/lib/sched.models` and find its architecture type, as follows:

```
$ grep 879 /opt/langtools/lib/sched.models
879     2.0     PA8000
```

You can also use the `+DS`*model* option to specify an architecture-specific instruction scheduler, where *model* has the same meaning as it does for the `+DA` option. Like the `+DA` option, the `+DS` option is unnecessary if the program will run on the same machine as you use to compile it. Also, if you compile with `+DA`*model*, the compiler will select the scheduling algorithm based on the same architecture—unless you use the `+DS` option to specify a different architecture.

NOTE Code generated for PA1.1 systems will execute PA2.0 systems, but the reverse is not true: the loader will not allow PA2.0 code to run on a PA1.1 system.

Creating shared libraries

As mentioned in "Linking to shared libraries" on page 69, many of the HP-UX as well as HP Fortran libraries are available in shared as well as archive versions. Linking with shared libraries can make the executable program smaller and can ensure that it always has the most current version of the library.

You can make shared versions of your own libraries, using the `+pic` command-line option and the `-b` linker option. The following sections describe how to use these options and show an example of how to create a shared library.

Compiling with +pic

The `+pic` option causes the compiler to generate **Position-Independent Code (PIC)** for use in a shared library. PIC contains no absolute addresses and can therefore be placed anywhere in a process's address space without addresses having to be relocated. This characteristic of PIC makes it shareable by multiple processes.

The syntax of the `+pic` option is:

```
+pic={short|long|no}
```

Although compiling with either `+pic=short` or `+pic=long` will generate PIC, in general you should use the `+pic=short` option. If the linker issues an error message saying that the number of referenced symbols in the shared library exceeds its limit, recompile with `+pic=long`, which will cause the compiler to allocate space for a longer symbol table.

The `+pic=no` is the default, which causes the compiler to generate absolute code, such as you would want for executable programs.

The following command line creates three object files—`x.o`, `y.o`, and `z.o`; the code in each file will be PIC:

```
$ f90 -c +pic=short x.f90 y.f90 z.f90
```

For more information about the `+pic` option, see the *HP Fortran Programmer's Reference*.

Linking with -b

The `-b` option is a linker option. It causes the linker to bind PIC object files into a shared library, instead of creating a normal executable file. The `-b` option must be used with the `ld` command; you cannot use the `f90` command to create a shared library. Also, the object files specified on the `ld` command line must consist of PIC; that is, they must have been created with either `+pic=short` or `+pic=long`.

The following command line links the object files `x.o`, `y.o`, and `z.o` into a shared library, named `my_lib.sl`:

```
$ ld -b -o my_lib.sl x.o y.o z.o
```

Note that this `ld` command line is much simpler than the `ld` command line required to link an executable file (for example, see "Linking with f90 vs. ld" on page 65).

Examples

This section shows an example of how to create and link to a shared library. The shared library will consist of PIC object files compiled from the source files, `hi.f90` and `bye.f90`. The library, `my_lib.sl`, will be linked to the executable program compiled from `greet.f90`. The code for three HP Fortran source files follows:

hi.f90

```
SUBROUTINE say_hi()
  PRINT *, 'Hi!'
END SUBROUTINE say_hi
```

bye.f90

```
SUBROUTINE say_bye()
  PRINT *, 'Bye!'
END SUBROUTINE say_bye
```

greet.f90

```
PROGRAM main
  CALL say_hi()
  CALL say_bye()
END PROGRAM main
```

The following command line creates the PIC object files (the `-c` option suppresses linking):

```
$ f90 -c +pic=short bye.f90 hi.f90
```

The next command line links the object files into the shared library:

```
$ ld -b -o my_lib.sl bye.o hi.o
```

The last command line compiles the source file `greet.f90` and links the object code with the shared library to produce the executable program `a.out`:

```
$ f90 greet.f90 my_lib.sl
```

The following is the output from a sample run of the executable program:

```
$ a.out
 Hi!
 Bye!
```

Using the C preprocessor

You can use the `f90` command to pass source files to the C preprocessor (`cpp`) before they are compiled. If the source files contain C preprocessor directives, `cpp` will act on the directives, modifying the source text accordingly. The `f90` driver will then pass the preprocessed source text to the compiler. Adding `cpp` directives to program source files and having the `cpp` command preprocess them is a convenient way to maintain multiple versions of a program—for example, a debugging version and a production version—in one set of files.

`cpp` directives are similar to debugging lines, a feature of many Fortran implementations (see "Using debugging lines" on page 117). Like `cpp` directives, debugging lines enable the compiler to treat source lines as either compilable statements or comments to be removed before compilation. But debugging lines are nonstandard, available only in fixed-form source, and not nearly as powerful as the `cpp` directives. Although `cpp` directives are not a standard feature of Fortran, `cpp` is a de facto standard feature of UNIX systems.

This section discusses how to do the following:

- Invoke `cpp` from the `f90` command line.
- Use the `-D` option to define `cpp` macros.
- Save the preprocessed output generated by `cpp`.

For more information about the `cpp` command and the directives it supports, see the *cpp*(1) man page.

Processing cpp directives

By default, the `f90` command passes source files ending in the `.F` extension to `cpp`. Compiling with the `+cpp=yes` option enables you to override this default and cause the `f90` driver to pass all source files to `cpp`. If you do not compile with the `+cpp=yes` option and if the source file does not have the `.F` extension, the compiler treats any `cpp` directives (but not any embedded Fortran statements) as comments and ignores them. (As a **language extension**, HP Fortran allows comments to begin with the `#` character, which is also the prefix character for all `cpp` directives.)

Consider the following program:

cpp_direct.f90

```
PROGRAM main
  REAL :: x

  WRITE (6, FMT='(A)', ADVANCE='NO') 'Enter a real number: '
  READ *, x
#ifdef DEBUG
PRINT *, 'The value of x in main: ', x
#endif
  PRINT *, 'x =', double_it(x)
END PROGRAM main

REAL FUNCTION double_it(arg)
  REAL :: arg

#ifdef DEBUG
PRINT *, 'The value of x in double_it: ', arg
#endif
  double_it = 2.0 * arg
END FUNCTION double_it
```

The program uses the `#ifdef` and `#endif` directives around `PRINT` statements. If the macro `DEBUG` is defined, `cpp` will leave the `PRINT` statements in the source text that is passed to the compiler; if it is not defined, `cpp` will remove the statements. You can define the macro in the source text, using the `#define` directive; or you can define it on the command line, using the `-D` command-line option. The advantage of the option is that it does not require editing the source file to define or undefine a macro.

The following command line uses the `-D` option to define the macro `DEBUG` (the space between `-D` and `DEBUG` is optional):

```
$ f90 +cpp=yes -D DEBUG cpp_direct.f90
```

Here is the output from a sample run of the executable program created by the preceding command line:

```
$ a.out
Enter a real number: 3
 The value of x in main:  3.0
 The value of x in double_it:  3.0
 x = 6.0
```

The next command line does not use the `-D` option, so that `DEBUG` is undefined, causing `cpp` to remove the `PRINT` statements from the source text that is passed to the compiler:

```
$ f90 +cpp=yes cpp_direct.f90
```

Here is the output from the nondebugging version of the program:

```
$ a.out
Enter a real number: 3.3
 x = 6.6
```

Saving the cpp output file

By default, the `f90` command discards the source text as processed by `cpp` after compilation. However, you can preserve this text by compiling with the `+cpp_keep` option. If the source file has the `.F` or `.f` extension, the output from `cpp` is written to a file with the same name but with the `.i` extension. If the source file extension is `.f90`, the output file has the `.i90` extension.

Here is the previous command line to preprocess and compile `cpp_direct.f90`, with the addition of the `+cpp_keep` option:

```
$ f90 +cpp_keep +cpp=yes cpp_direct.f90
```

After the `PRINT` statements have been removed, the resulting output file looks like this:

```
$ cat cpp_direct.i90
# 1 "cpp_direct.f90"
PROGRAM main
        REAL :: x

        WRITE (6, FMT='(A)', ADVANCE='NO') 'Enter a real number:'
        READ *, x



        PRINT *, 'x =', double_it(x)
END PROGRAM main

REAL FUNCTION double_it(arg)
        REAL :: arg



        double_it = 2.0 * arg
END FUNCTION double_it
```

Creating demand-loadable executables

By default, the loader loads the entire code for an executable program into virtual memory. For very large programs, this can increase startup time. You can override this default by causing the linker to mark your program **demand load**. A demand-loadable program is loaded into memory a page at a time, as it is accessed.

Use the `+demand_load` option to make your program demand loadable, as follows:

```
$ f90 +demand_load prog.f90
```

The `f90` command passes this option to the linker, which marks the executable program *demand load*.

Demand loading allows a program to start up faster because page loading can be spread across the execution of the program. The disadvantage of demand loading is that it can degrade performance throughout execution.

Creating shared executables

By default, the linker marks an executable program as **shared**. A **shared executable** is shareable by all processes that use the program. The first process to run the program loads its code into virtual memory. If the program is already loaded by another process, then a process shares the code with the other process.

You can override this default with the `+noshared` option, which causes the linker to mark the executable as *unshared*, making the program's code nonshareable. The following command line causes the linker to mark `prog.f90` as *unshared*:

```
$ f90 +noshared prog.f90
```

In some circumstances, it may help to debug a program or to improve its runtime performance by making it nonshareable. In general, however, it is not desirable because nonshareable executables place greater demands on memory resources.

Compiling in 64-bit mode

Compiling HP Fortran programs with the `+DA2.0W` option causes `f90` to produce 64-bit executable programs. You should consider compiling in 64-bit mode if your program does any of the following:

- Accesses a large shared memory (greater than 1.75 gigabytes) or large data spaces (greater than 1 gigabyte or, if using `EXEC_MAGIC`, greater than 1.9 gigabytes)
- Uses large data elements—greater than 32-bit words
- Provides objects or libraries that might be used in a 64-bit application

There are no HP Fortran language differences between 32-bit and 64-bit programs. Recompiling should suffice to convert a 32-bit Fortran program to run as a 64-bit program.

However, the C language has some differences in data type sizes. If your Fortran program calls functions written in C and is compiled in 64-bit mode, the size differences may require promoting the data items that are passed to or from the C functions. See Table 29 on page 163 and Table 30 on page 163 for the size differences between Fortran and C data types when compiled in 64-bit mode.

NOTE If your program does not need to run in 64-bit mode, there is no benefit to compiling it in 64-bit mode. In fact, the executable program may run slower than if compiled in 32-bit mode.

Using environment variables

Environment variables are variables that are defined in the operating environment of the system and are available to various system components. For example, when you run a program, the shell looks at the `PATH` variable to determine where the program is located. Table 20 lists and briefly describes the environment variables that control the way HP Fortran programs are compiled, linked, and run.

Table 20 HP Fortran environment variables

Environment variable	Description
`FTN_IO_BUFSIZ`	Sets the default size in bytes of the I/O library streams file buffer; equivalent to calling `setvbuf` for each logical unit that is opened; see the *setbuf*(3S) man page.
`HP_F90OPTS`	Specifies a list of command-line options that `f90` inserts in the command line that invokes the HP Fortran compiler.
`LPATH`	Specifies a list of directories that the linker is to search for libraries.
`MP_NUMBER_OF_THREADS`	Specifies the desired number of processors to be used to run HP Fortran programs that have been compiled for parallel execution.
`TMPDIR`	Specifies a directory for temporary files; used in place of the default directory `/var/tmp`.
`TTYUNBUF`	Controls **tty buffering**. To enable tty buffering, set `TTYUNBUF` to 0; to disable tty buffering, set it to a nonzero value.

The following sections describe how to use the `HP_F90_OPTS`, `LPATH`, and `MP_NUMBER_OF_THREADS` environment variables. See the *environ*(5) man page for information about system-level environment variables.

HP_F90OPTS environment variable

The HP_F90OPTS environment variable is read by the f90 driver for options to insert in the command line. This variable is useful when you want the same options and arguments each time you invoke the f90 command. For example, if HP_F90OPTS is set to the -v option, the following command line:

```
$ f90 +list hello.f90
```

is equivalent to:

```
$ f90 -v +list hello.f90
```

The syntax of the HP_F90OPTS variable allows the bar (|) character to be used to specify that options appearing before | are to be recognized before any options on the command line and that options appearing after | are to be recognized after any options on the command line. For example, the commands:

```
$ export HP_F90OPTS="-O|-lmylib"
$ f90 -v hello.f90
```

are equivalent to:

```
$ f90 -O -v hello.f90 -lmylib
```

If you are programming in the csh, the command line to define HP_F90OPTS would be:

```
% setenv HP_F90OPTS "-O|-lmylib"
```

LPATH environment variable

The LPATH environment variable is read by the linker to determine where to look for libraries to link with a program's object file. Depending on whether LPATH is set or not, one of the following actions occurs:

- If LPATH is already set, only the directories listed in LPATH are searched. This happens, for example, when LPATH is set in a user's .kshrc or .cshrc file, or after LPATH is defined from the command line.
- If LPATH is not set, the f90 command sets default LPATH settings that are used when linking the object files listed on the f90 command line.

Because the `f90` command sets `LPATH` before calling the linker, it should not be necessary to set this variable for most compilations. However, if you do need to set it (for example, you use the `ld` command to link), the following directories should be the first items in `LPATH`:

- `/opt/fortran90/lib`
- `/usr/lib`
- `/opt/langtools/lib`

The following command lines set `LPATH` to include these directories, using (respectively) the `ksh` and `csh` syntax:

```
$ export LPATH:/opt/fortran90/lib:/usr/lib:/opt/
langtools/lib
```

```
% setenv LPATH "/opt/fortran90/lib:/usr/lib:/opt/
langtools/lib"
```

To see how `f90` sets `LPATH` before calling the linker, compile with the `-v` option for verbose output. For an example, see "Linking with f90 vs. ld" on page 65.

MP_NUMBER_OF_THREADS environment variable

The `MP_NUMBER_OF_THREADS` environment variable sets the number of processors that are to execute a program that has been compiled for parallel execution. If you do not set this variable, it defaults to the number of processors on the executing machine.

The following command lines set `MP_NUMBER_OF_THREADS` to specify that programs compiled for parallel execution can execute on two processors:

```
$ export MP_NUMBER_OF_THREADS=2       # ksh syntax
% setenv MP_NUMBER_OF_THREADS 2       # csh syntax
```

For information about parallel execution, see "Compiling for parallel execution" on page 144.

3 Controlling data storage

This chapter describes the use of command-line options, directives, and other language features to control data in HP Fortran programs. In particular, it discusses the following topics:

- Disabling implicit typing
- Automatic and static variables
- Increasing the precision of constants
- Increasing default data sizes
- Sharing data among programs
- Modules vs. common blocks

NOTE For information about how HP Fortran aligns data, refer to the *HP Fortran Programmer's Reference*.

Disabling implicit typing

By default, HP Fortran uses *implicit typing* to determine the type of a variable or function that has not been declared with a type declaration statement. That is, the type of an undeclared entity is determined by the first letter of its name: if the letter is in the range `I - N`, the entity is of type integer; otherwise, it is of type real.

Although implicit typing is mandated by the Standard, its use can become a source of runtime bugs because implicit typing allows the inadvertent use of undeclared variables or functions. For the sake of illustration, consider a program that calls a nonintrinsic library function named `foo`. Assume that:

- The default typing rules are in effect.
- `foo` returns an integer.
- The programmer has not declared the return type of `foo` and has assigned its return value to a variable of type real.

Experience has shown that this is not an unlikely scenario and that it can produce bad results.

The Standard provides the `IMPLICIT NONE` statement to override implicit typing. But the `IMPLICIT NONE` statement is limited in scope to the program unit in which it appears. To force explicit typing for all files specified on the command line, use the `+implicit_none` option. This option disables implicit typing; that is, all variables, arrays, named constants, function subprograms, `ENTRY` names, and statement functions (but not intrinsic functions) must be explicitly declared.

Using this option is equivalent to specifying `IMPLICIT NONE` for each program unit in each file specified on the `f90` command line. However, the `+implicit_none` option does not override any `IMPLICIT` statements in the source file. The *HP Fortran Programmer's Reference* describes the implicit typing rules, the `IMPLICIT NONE` statement, and the `+implicit_none` option.

Automatic and static variables

By default, HP Fortran allocates stack storage for program variables. Such variables are called **automatic variables** because they are allocated at each invocation of the program unit in which they are declared.

Static variables are allocated storage from static memory when the program is first loaded into memory. They remain allocated for the life of the program.

HP Fortran allocates static storage for the following variables:

- Variables specified in a `COMMON` or `EQUIVALENCE` statement.
- Variables initialized in a type declaration statement or in a `DATA` statement.
- Variables specified in a `SAVE` or `STATIC` statement. A `SAVE` statement without a variable list specifies static storage for all variables in the scoping unit.
- Variables in program files that have been compiled with the `+save` or `+Oinitcheck` command-line option. See "Uninitialized variables" on page 226 for information about using these options when porting.

Static variables have two characteristics that are of special interest:

- They are set to 0 or null value at load-time.
- They do not require re-initialization at each invocation of their program unit.

Static variables have several disadvantages. In Fortran programs that use recursion, static variables can defeat one purpose of recursion—to provide a fresh set of local variables at each recursive call. Also, the widespread use of static variables in a program can slow its performance: static variables are ineligible for such fundamental optimizations as register allocation, and they can limit the optimization of program units that use them.

The following example program illustrates the difference between automatic and static variables. The program consists of a main program unit that calls a recursive internal subroutine. The subroutine increments two variables (`stat_val` and `auto_val`), prints the updated

variables, and then calls itself recursively. Neither of the two variables is explicitly initialized, but `stat_val` is declared with the `SAVE` attribute, which means that it is allocated static storage and is pre-initialized to 0 by the compiler.

The program is shown below.

recursive.f90

```
PROGRAM main
! This program calls a recursive internal subroutine.

    CALL recurse

    CONTAINS
        ! This subroutine calls itself four times.
        !   Each time it is called, it adds 1 to the values in
        !   stat_val and auto_val and displays the result.
        !   stat_val has the SAVE attribute and therefore is
        !   pre-initialized and retains its value between calls.
        !   auto_val is an automatic variable and therefore has
        !   an unpredictable value (plus 1) at each call.
        RECURSIVE SUBROUTINE recurse
        INTEGER(KIND=1), SAVE :: stat_val
        INTEGER(KIND=1) :: auto_val

        stat_val = stat_val + 1
        auto_val = auto_val + 1
        PRINT *, 'stat_val = ', stat_val
        PRINT *, 'auto_val = ', auto_val
        IF (stat_val < 4) THEN
            CALL recurse()
        END IF

        END SUBROUTINE recurse

END PROGRAM main
```

Following are the command lines to compile and execute this program, along with sample output. Notice that `stat_val` regularly increments at each call. The reason is that it is a static variable and therefore retains its value between calls. But `auto_val` is not actually incremented; it is an automatic variable and is given a fresh (and uninitialized) memory location at each call. In other words, the subroutine adds 1 to whatever value happened to be in the memory location that was allocated to `auto_val` at the start of the call:

```
$ f90 recursive.f90
$ a.out
 stat_val =  1
 auto_val =  124
```

```
stat_val =  2
auto_val =  1
stat_val =  3
auto_val =  65
stat_val =  4
auto_val =  65
```

NOTE HP Fortran provides the AUTOMATIC and STATIC statements as porting extensions. The STATIC statement is functionally the same as the SAVE statement, and the AUTOMATIC statement may be used to declare a variable as automatic. However, such a declaration is generally pointless because variables compiled under HP Fortran are automatic by default.

The *HP Fortran Programmer's Reference* provides detailed information about the AUTOMATIC, SAVE, and STATIC statements.

Increasing the precision of constants

By default, HP Fortran evaluates all floating-point constants as single-precision. For example, the compiler treats following constant

```
3.14159265358979323846
```

as though you had specified:

```
3.1415927
```

Although the loss of **precision** might be acceptable when assigning to single-precision variables, it is might be less acceptable when assigning to double-precision variables or when using floating-point constants in expressions where the loss in precision might result in significant round-off differences.

NOTE HP Fortran provides two ways to override the default precision of individual constants: the kind parameter and the exponent form. The kind parameter indicates the precision of floating-point constants: 4 for single-precision, 8 for double-precision, and 16 for quad-precision.

In the following example, the kind parameter `_8` specifies that the constant is to be evaluated as double-precision:

```
3.14159265358979323846_8
```

To change the precision of all floating-point constants (except those having a kind parameter), you can use the `+real_constant` option. This option takes two forms, `+real_constant=double` and `+real_constant=single`, which specify (respectively) double-precision and single-precision for floating-point constants in the files compiled with this option. The `+real_constant=single` form is the default. Neither form of the option has any affect on constants that have the kind parameter.

To promote all floating-point constants in the source files `x.f`, `y.f`, and `z.f`, compile with the command line:

```
$ f90 +real_constant=double x.f y.f z.f
```

The `+real_constant=single` option specifies that all floating-point constants in a file are to be treated as single-precision (the default). The following command line specifies single-precision for all floating-point constants in the files `a.f`, `b.f`, and `c.f`:

```
$ f90 +real_constant=single a.f b.f c.f
```

Note that `+real_constant=single` does not demote constants that use either the kind parameter or the exponent form (for example, `4.0D0`).

For information about increasing the precision of variables, see "Increasing default data sizes" on page 96. The *HP Fortran Programmer's Reference* describes the syntax of the kind parameter and the exponent form and the `+real_constant` option. For detailed information about how floating-point arithmetic is implemented on HP 9000 computers and how floating-point behavior affects the programmer, refer to the *HP-UX Floating-Point Guide*.

Increasing default data sizes

The +autodbl and +autodbl4 options enable you to increase the default sizes (that is, the number of storage bytes) for both constants and variables of default numeric and logical types. Unlike the +real_constant option, the +autodbl and +autodbl4 options affect both constants and variables of both real and integer types. (For information about using the +real_constant option, see "Increasing the precision of constants" on page 94.)

When compiled with the +autodbl and +autodbl4 options, constants are treated as though they had twice the default number of bytes (4) available for evaluating them. The effect of these options is to increase the range of default integers and the precision of default reals.

The +autodbl and +autodbl4 options have no effect on the size of entities declared with the CHARACTER, BYTE, or DOUBLE COMPLEX statements, nor on entities that are explicitly sized. That is, if a variable is declared with a kind parameter or if a constant has a kind parameter, it is unchanged by +autodbl or +autodbl4.

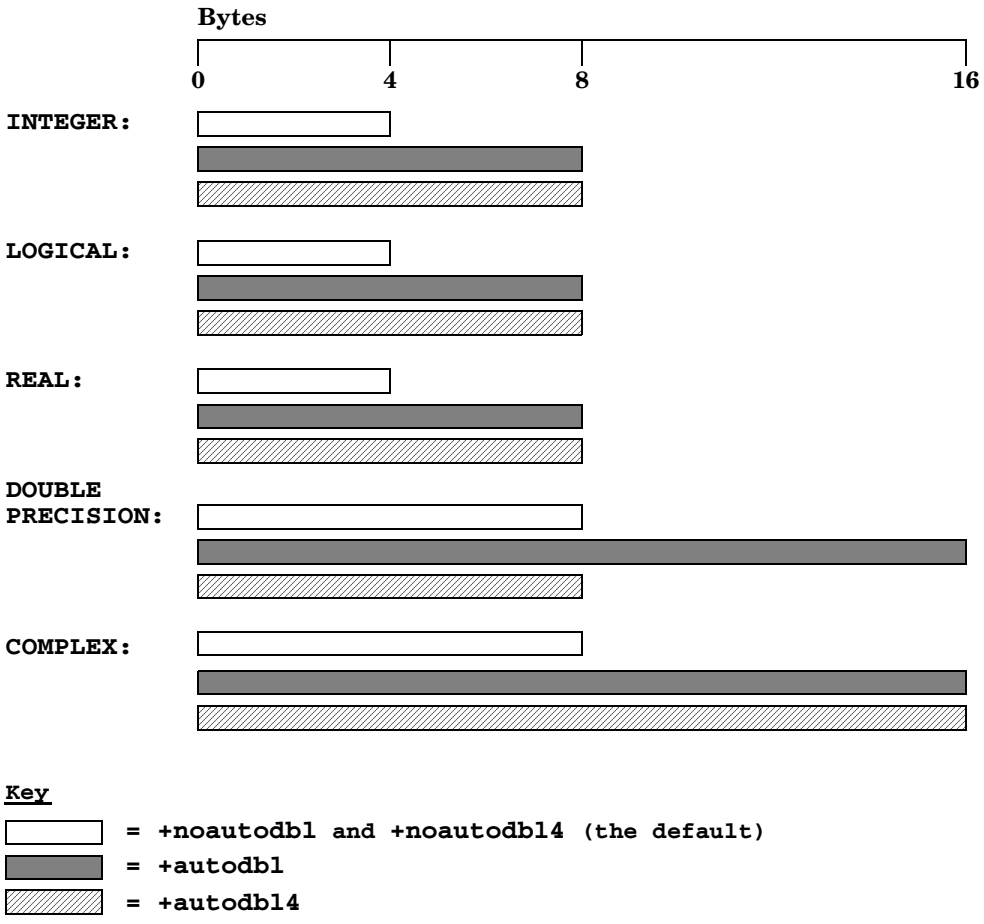
NOTE HP Fortran interprets the kind parameter as indicating the number of storage bytes to allocate for a variable. When used with variables and constants of type real, the kind parameter also indicates the precision: 4 for single-precision, 8 for double-precision, and 16 for quad-precision.

Promoting double-precision variables to quad-precision can have a severe impact on performance because the instructions to perform quad-precision operations are implemented in software. If you are concerned about performance and want to increase default data sizes, consider using the +autodbl4 option, which does not promote variables declared with the DOUBLE PRECISION statement. There is no other difference between +autodbl or +autodbl4.

These options affect all files on the command line. To increase the size or precision of selected variables and constants, use the kind parameter.

Figure 2 on page 97 shows the default data types whose sizes are changed by the +autodbl and +autodbl4 options.

Figure 2 Increasing default data sizes



The following program illustrates the different effects of the `+autodbl` and `+autodbl4` options. The program assigns the same quad-precision constant to three variables:

- `x`, a default (that is, single-precision) real
- `y`, a real that is declared as double-precision with the kind parameter
- `z`, a double-precision real that is declared with the `DOUBLE PRECISION` statement

The following program includes `PRINT` statements to show the stored values.

precision.f90

```
PROGRAM main
REAL x
REAL(KIND=16) y
DOUBLE PRECISION z

! Assign a quad-precision constant to a default real:
x = 3.14159265358979323846_16
PRINT 10, 'Stored in x:  ', x

! Assign a quad-precision constant to a variable that
! has been explicitly sized for quad-precision:
y = 3.14159265358979323846_16
PRINT 10, 'Stored in y:  ', y

! Assign a quad-precision constant to a variable
!   declared with the DOUBLE PRECISION statement:
z = 3.14159265358979323846_16
PRINT 10, 'Stored in z:  ', z


10 FORMAT (A, F22.20)

END PROGRAM main
```

Following are three different sets of command lines to compile and execute this program, including sample output from each compilation. Note that variable `y` remains the same for each compilation: the compiler does not promote variables that are sized with the kind parameter.

First, the program is compiled without any option:

```
$ f90 precision2.f90
$ a.out
Stored in x:  3.14159274101257320000
Stored in y:  3.14159265358979323846
Stored in z:  3.14159265358979310000
```

Next, the program is compiled with the `+autodbl` option. As shown in the output, `x` is promoted to double-precision and `z` to quad-precision:

```
$ f90 +autodbl precision2.f90
$ a.out
Stored in x:  3.14159265358979310000
Stored in y:  3.14159265358979323846
Stored in z:  3.14159265358979323846
```

Finally, the program is compiled with the `+autodbl4` option. As shown in the output, `x` is promoted, but `z` is not:

```
$ f90 +autodbl4 precision2.f90
$ a.out
Stored in x:  3.14159265358979310000
Stored in y:  3.14159265358979323846
Stored in z:  3.14159265358979310000
```

Though useful for increasing the range and precision of numerical data, the `+autodbl` and `+autodbl4` options are especially useful when porting; see "Large word size" on page 227. For detailed information about these options, see the *HP Fortran Programmer's Reference*. For detailed information about how floating-point arithmetic is implemented on HP 9000 computers and how floating-point behavior affects the programmer, refer to the *HP-UX Floating-Point Guide*.

Sharing data among programs

If you are designing an application that requires multiple **threads** of control that share the same data, the design can take either of two forms:

- The program makes calls to the threads library:

  `/usr/lib/libpthread.sl`

  which creates multiple threads executing in a single process and therefore all sharing the same address space.

- The application consists of several programs that run simultaneously in separate processes and that access an HP-UX shared memory segment.

The first approach is beyond the scope of this manual and requires that you have an understanding of how to call the threads library.[1] The second approach is described here.

To share data among several HP Fortran programs that are executing simultaneously in separate processes, use the `$HP$ SHARED_COMMON` directive. This directive enables you to create a common block that is accessible by HP Fortran programs executing in different processes.

The `$HP$ SHARED_COMMON` directive causes the compiler to insert HP-UX system calls to perform shared memory operations. To the programmer, the programs sharing the memory segment appear as though they were program units in the same program, accessing a set of common block variables.

Following are two programs to illustrate how the `$HP$ SHARED_COMMON` directive works:

- The first program, `go_to_sleep.f90`, must execute first. Because it executes first, it creates the shared memory segment and then enters a `DO` loop, where it waits until the second program starts to execute. You can use the `ipcs -m` command to confirm that a shared memory segment has been created.

1. Specifying the `+Oparallel` option causes the compiler to transform eligible loops in an HP Fortran program for parallel execution. For information about compiling for parallel execution, see “Compiling for parallel execution” on page 144.

- When the second program, `wake_up.f90`, starts to execute, it writes to the shared common block variables, one of which causes `go_to_sleep.f90` to break out of the `DO` loop and run to completion.

The `$HP$ SHARED_COMMON` directive must appear at the beginning of the specification part of the main program unit of each program sharing the memory segment. Also, the common block specified for sharing must have the same layout in all files in which it is declared.

You can use the `ipcs -m` command both to determine that HP-UX has created a shared memory segment and, after the programs complete execution, to confirm that it has been released.

The following two examples illustrate these concepts.

go_to_sleep.f90

```
PROGRAM main
! This program, go_to_sleep.f90, and its companion, wake_up.f90,
!   share data in a common block, using the $HP$ SHARED_COMMON
!   directive.  Execute this program first.  After it starts to
!   execute, use ipcs(1) to confirm that a shared memory segment
!   has been created.  In a separate process, run wake.f90.
!   When it executes, it assigns to alarm, ending this program.

        LOGICAL :: alarm
        CHARACTER(LEN=8) :: message

! Declare a common block, shared_data, for sharing among
!   multiple, simultaneously executing programs.  Each program
!   that shares the common block must reference it by the same
!   key, 'scb1'.
!$HP$ SHARED_COMMON KEY='scb1' /shared_data/

! Declare a common block with two variables:  alarm and message.
!   when alarm is set by wake_up.f90, this program breaks out
!   of the DO loop, prints message (which wake_up.f90 has
!   written to), and exits.
        COMMON /shared_data/ alarm, message


        alarm = .FALSE.
! Wait for alarm to be set...
        DO WHILE (alarm .EQ. .FALSE.)
    ! sleep(1) is an HP-UX system call that suspends a process
        !   for the number of seconds specified by the argument.
        !   The %VAL function tells Fortran that sleep expects
its
        !   argument to be passed by value.
        CALL sleep(%VAL(1))
        END DO
```

```
! Message from wake.f90:
        PRINT *, message

! The shared memory segment is destroyed when this program halts.

        END
```

NOTE *In the example above, you must use +U77 to access the correct sleep in the Fortran library. If you use +U77, the line above:*

CALL sleep (%VAL(1))

should instead read:

CALL sleep (1)

wake_up.f90

```
PROGRAM main
! This program, wake_up.f90, should be run just after its
!   companion, go_to_sleep.f90, starts to execute but in a
!   separate process.  The $HP$ SHARED_COMMON directive
!   enables both programs to share the same memory.

! Directive puts the common block in shared memory.
$SHARED_COMMON KEY='scb1' /shared_common/

        LOGICAL :: alarm
        CHARACTER(LEN=8) :: message


! Declare a named common block for shared memory.  It must
!   be laid out n exactly the same way in both programs.
        COMMON /shared_common/ alarm, message

! Write to message, sleep reads it.
        message = "I'm up!"

! Set alarm to wake up sleep.
        alarm = .TRUE.

! The shared memory segment will now be detached.
!   However, because go_to_sleep is still running,
!   the segment will still be present in memory until
!   it stops executing, too.

        END
```

Following are the command lines to compile each program:

```
$ f90 -o go_to_sleep go_to_sleep.f
$ f90 -o wake_up wake_up.f
```

Run the first program in any process by doing the following:

```
$ go_to_sleep
```

In another process, use the following command line to confirm that a shared memory segment has been created for the program (the last in the list is the newly created one):

```
$ ipcs -m
IPC status from /dev/kmem as of Fri Mar 21 15:55:29 1997
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m      0 0x4119c72b --rw-rw-rw-     root     root
m      1 0x4e180002 --rw-rw-rw-     root     root
m      2 0x41187bf4 --rw-rw-rw-     root     root
m      3 0x00000000 --rw-------     root      sys
m   7004 0x43186ea0 --rw-rw-rw-   daemon   daemon
m   6005 0x73636231 --rw-rw-rw-       ed     lang
```

Now run the second program in the second process:

```
$ wake_up
```

At this point, the program executing in the first process outputs the following and completes execution:

```
 I'm up!
```

The following command line confirms that the shared memory segment was released:

```
$ ipcs -m
IPC status from /dev/kmem as of Fri Mar 21 15:55:29 1997
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m      0 0x4119c72b --rw-rw-rw-     root     root
m      1 0x4e180002 --rw-rw-rw-     root     root
m      2 0x41187bf4 --rw-rw-rw-     root     root
m      3 0x00000000 --rw-------     root      sys
m   7004 0x43186ea0 --rw-rw-rw-   daemon   daemon
```

For information about sharing data between Fortran program units and C functions within the same program, see “Sharing data” on page 183. The *HP Fortran Programmer's Reference* provides detailed information about the `COMMON` statement and about the `$HP$ SHARED_COMMON` directive. Refer to the *shmop*(2) man page for information about HP-UX shared memory operations.

Modules vs. common blocks

The common block has been a mainstay of Fortran programs throughout the evolution of the language, and it continues to be a part of Fortran. The common block provides a convenient means to share data among program units, especially when the program units sharing data do not otherwise communicate with each other. The common block can also be used to share data between simultaneously executing Fortran programs (see "Sharing data among programs" on page 100) and between Fortran program units and C functions linked together in the same program (see "Sharing data" on page 183).

One of the problems with the common block, however, is that the programmer must replicate the `COMMON` declaration in each of the sharing program units. If any of the common variables are out of order or have a different type or size, the program units may not access the same data. The compiler gives no indication of this discrepancy because it assumes that the programmer is giving one program unit a different view of the shared storage—even when the discrepancy is owing to oversight.

To deal with this problem, many implementations of FORTRAN 77 have provided the `INCLUDE` extension. This extension enables the user to centralize common block definitions in one file. At compile-time, the compiler reads the file into program units that have the `INCLUDE` line. While this approach eliminates the problem of discrepant common blocks, it introduces another problem: the `INCLUDE` facility is nonstandard FORTRAN 77, and its use is nonportable.

To deal with the portability issue, Standard Fortran defines the `INCLUDE` line. Unfortunately, the definition in the Standard leaves many of the details up to the implementation, so that use of the `INCLUDE` line in Fortran programs still runs the risk of nonportability.

Another problem with the common block—especially when used with equivalencing—is that it can inhibit optimization. Common block variables are generally ineligible for register allocation, and **aliasing** variables in common can prevent the optimization of the program units that use the aliased variables.

The *module program unit* is the Fortran answer to the common block. The programmer declares shareable variables in a module. Any program unit that wants to access them references the name of the module in a `USE` statement. The concept of the module eliminates the need to re-declare the common variables, without requiring the `INCLUDE` line.

In addition, the module provides the following controls on access to module data:

- The `PUBLIC` and `PRIVATE` statements declare which module variables are accessible outside the module and which are not.
- The `USE` statement has an `ONLY` clause that specifies which module variables are accessible to a particular program unit.
- The `USE` statement also has a renaming feature to resolve name clashes between local variables and module variables.

Another feature of the module is that it can include procedures. This feature provides a way to package data with the procedures needed to operate on the data. A program unit accesses module procedures in the same way it does module data, with the `USE` statement. The interface of module procedures is available to the compiler, which can perform compile-time checks on the actual arguments that are passed to a module procedure.

Although the module does not completely replace the common block (see, for example, “Sharing data among programs” on page 100), it does provide a safer and more flexible alternative to the more common uses—and abuses—of the common block.

For an example of a program that uses the module to share data, see “Compiling programs with modules” on page 72. The *HP Fortran Programmer's Reference* provides detailed information about the module program unit and the `MODULE` and `USE` statements.

4 Debugging

This chapter describes different HP Fortran features for debugging programs. These features include compile-line options, compiler directives, and programming tools that are useful for locating errors in your program. More specifically, this chapter discusses the following topics:

- Using the HP WDB debugger
- Stripping debugging information
- Handling runtime exceptions
- Using debugging lines

Using the HP WDB debugger

The **HP WDB debugger** is the primary tool for debugging HP Fortran programs. The debugger provides such basic debugging functions as program control, process control, program and data monitoring, and expression evaluation. The debugger has both a graphical interface and a line-mode interface.

The debugger software includes different managers that enable it to handle different source languages, target machines, object file formats, and user formats. The Fortran language manager allows you to use Fortran syntax when entering expressions on the debugger command line.

Before beginning a debugging session, you must compile the program with the `-g` compile-line option. If you compile and link separately, you must use the `-g` option on both command lines. The option causes the compiler to generate additional information needed by the debugger and to insert it into the output code.

After compiling your program with the `-g` option, invoke the debugger with the `wdb` command, supplying the name of the executable as an argument. For example, the following command compiles `prog.f90` for debugging:

```
$ f90 -g prog.f90 -o db_prog
```

Here is the command to start debugging the executable program:

```
$ wdb db_prog
```

You can use the debugger to debug code that has been optimized at levels 0, 1, and 2. To debug optimized code, compile the program with both the `-g` and `+O`*opt-level* options, where *opt-level* is 0, 1, or 2. The following command line `prog.f90` at optimization level 2 and prepares for debugging:

```
$ f90 +O2 -g prog.f90 -o db_prog
```

Compiling with the `-g` option increases the size of both the object file and the executable file. After you have debugged your program and are ready to build the production version, you may want to recompile *without* the `-g` option.

For complete information about HP WDB debugger, refer to http://www.hp.com/go/wdb. Printed and online documentation are available at this site.

Stripping debugging information

Programs compiled with HP Fortran include minimal debugging information in the executable program. This information consists of a **symbol table**—a list of all the symbols in your program and their offset addresses. The symbol table provides the information needed to produce a procedure traceback. It is also used by the debugger and by the CXperf performance analysis tool.

However, the symbol table is not the same as the debugging information that is added to your program when you compile with the `-g` option. The symbol table is added to an executable even if the program is not compiled with the `-g` option.

If the size of executable is critical to your application, you can use the `+strip` option to remove symbol table information from the production version of your program. If you compile and link on separate command lines, you must use the `+strip` option on both command lines. Instead of recompiling with `+strip`, you can use the `strip` utility, which removes all debugging information, including the symbol table.

If the size of your executable is not important, you may want to retain the symbol table in the production version of your program. This table can be used by the debugger to provide minimal debugging. If a program has not been compiled with `-g` and does not include a symbol table, it is unusable by the debugger. Also, without the information provided by the symbol table, a procedure traceback displays virtual addresses only.

The amount of code that the symbol table information that adds to an executable is considerably less than the amount that compiling with `-g` adds. For descriptions of the `-g` and `+strip` options, refer to the *HP Fortran Programmer's Reference*. For information about the `strip` utility, refer to the *strip*(1) man page.

Handling runtime exceptions

Broadly defined, an **exception** is an error or fault condition that affects a program's results. Exceptions can range from the relatively benign inexact result condition that occurs in certain floating-point operations to the more severe **segmentation violation** that occurs when a runaway program attempts to access unallocated memory.

Exceptions that threaten the integrity of the operating system can cause HP-UX to raise an exception **signal** (for example, `SIGSEGV` for a segmentation violation) so that the process can take appropriate action to recover from the exception. Such exceptions may cause the program that took the exception to abort, but not necessarily. By trapping an exception—that is, by catching the signal—a program may handle the exception, if only by aborting when it occurs.

There are also a well-defined set of floating-point conditions that, although they pose no threat to the operating system, can also cause an exception—for example, dividing a floating-point number by zero. By default, **traps** for floating-point exceptions are disabled on HP 9000 computers, but they can be enabled by `+fp_exception` and `+FP` options. (You can also use the `ON` statement to enable traps for floating-point exceptions.)

Programs that have been compiled with the `+fp_exception` option can trap the exceptions listed in Table 21. Any of the exceptions listed in the second column will cause the operating system to generate the signal listed in the first column. Programs compiled with `+FP` can trap specific floating-point exceptions (`SIGFPE`).

Table 21 Signals recognized by +fp_exception

Signal	Exception
`SIGBUS`	Bus error instruction
`SIGFPE`	Floating-point exceptions
`SIGILL`	Illegal instruction
`SIGSEGV`	Segmentation violation or memory fault
`SIGSYS`	Bad argument to a kernel system call

When a program compiled with `+fp_exception` takes an exception, the following events occur:

- The program traps the exception.
- A procedure traceback is displayed on standard error. A procedure traceback lists routine names and their offsets that are helpful in locating the code that triggered the exception.
- The program aborts.

The following sections discuss each of exceptions listed in Table 21. For more information about signals, refer to the *signal*(2) and *signal*(5) man pages.

NOTE Standard Fortran 90 provides the `IOSTAT=` and `ERR=` specifiers for handling I/O runtime errors. For information about these specifiers, refer to the descriptions of the I/O statements (for example, `OPEN` and `READ`) in the *HP Fortran Programmer's Reference*. For a descriptive list of the error messages that can be returned by `IOSTAT=`, refer to the *HP Fortran Programmer's Reference*.

Bus error exception

A bus error exception occurs when a program references an inaccessible memory location, typically because the reference is to an unaligned or nonexistent address, or because of a hardware failure.

The most likely cause of a bus error is unaligned data reference. A program that passes an array of `(KIND=1)` elements to a routine that attempts to access them as `(KIND=4)` elements may take a bus error exception. Or if an array of `(KIND=1)` elements is declared in a common block and the third element is passed to a routine that attempts to access it as a `(KIND=4)` variable, the program will take a bus error exception. For information about the **alignment** of HP Fortran data types, refer to the *HP Fortran Programmer's Reference*.

Bus errors can occur (as can other exceptions) in any program that generates bad address references. Although less likely to happen with programs that use the standard Fortran 90 pointer, bad address references can happen when the Cray-style pointer extension is misused or when Fortran program unit passes a parameter by value to a C routine that attempts to use it as a pointer.

Floating-point exceptions

In accordance with the IEEE Posix Standard, floating-point exceptions are disabled on HP 9000 computers. Thus, if a program attempts the following operation:

```
x = 1.0/0.0
```

it will not trap it as an exception and will not abort. Instead, the value of a positive infinity (displayed as `+INF`) will be assigned to `x`.

HP Fortran provides two compile-line options, `+FP` and `+fp_exception`, which enable traps for floating-point exceptions. The differences between the two options are:

- The `+fp_exception` option enables traps for the following IEEE floating-point exceptions:
  - **Invalid operation**
  - **Division by zero**
  - **Overflow**
  - **Underflow**

  The `+FP` option also enables the trap for the inexact operation exception. For detailed descriptions of these exceptions, refer to the *HP-UX Floating-Point Guide*.
- Unlike the `+fp_exception` option, the `+FP` option includes a *flags* argument by which you can enable specific exceptions.
- The `+FP` option can also be used to enable **fast underflow** on systems that support it (chiefly PA2.0 systems).
- Both options cause your program to abort when it traps the exception. However, `+fp_exception` identifies the type of the exception that occurred and the virtual address of the statement that triggered it. Also, `+FP` causes a **core dump**; `+fp_exception` does not.

You can also trap floating-point exceptions with the `ON` statement. Although the `ON` statement requires you to modify source code, it enables you to write trap procedures so that your program can recover from exceptions. For more information about using the `ON` statement, see Chapter 5, “Using the ON statement,” on page 119.

Refer to the *HP Fortran Programmer's Reference*, for detailed information about the `+FP` and `+fp_exception` options. Also, the *HP-UX Floating-Point Guide* has a useful discussion of both options and includes detailed information on floating-point exceptions and how to handle them.

Illegal instruction exception

An illegal instruction exception occurs when a program attempts to execute a bit pattern that is not an op-code. A common cause of this exception is an overwritten stack. If a program overwrites the part of the stack that holds the return address, the new (and bad) address may cause execution control to jump to a memory location that contains data or some other nonexecutable bit pattern. The attempt to execute this location will result in an illegal instruction exception.

This exception can also occur if your program is linked to a bad library, especially if the library contains code that was written in assembler or if it was corrupted during a file transfer.

This exception may indicate a compiler error. If you cannot find the cause of this exception in your code, contact your HP support representative.

Segmentation violation exception

Before a program starts to execute, it is allocated a memory segment, which defines the area of memory that it can use. If the program attempts to access a memory location outside its segment, the operating system will raise the `SIGSEGV` signal, indicating a segmentation violation or memory fault.

Any program that can generate address references outside its segment—for example, by indexing beyond the declared boundary of an array—may cause a segmentation violation. In C programs, bad pointers often result in this exception. The standard Fortran 90 pointer is more self-protective than the C pointer, but it too can be misused and lead to the state of mind memorialized in the lyric (known only to Cooper Redwine[1]): "I've got those segmentation violation, core dumped blues." The Cray-style pointer extension is more like the C pointer and is therefore more susceptible to the abuse that results in segmentation violations.

1. See his *Upgrading to Fortran 90* (New York 1995), p. 278.

Programs that cause a **stack overflow** (for example, by attempting to allocate more local variables on the stack than the kernel can handle or by infinite recursion) can also cause a segmentation violation. If your program needs a bigger stack, run the System Administrator Manager (SAM) and increase the `maxssiz` parameter. Also, see the *HP-UX System Administration Tasks* manual for information about reconfiguring the kernel.

Segmentation violations are especially common when calling C functions from Fortran program units. If the number, type, or calling conventions of the arguments being passed do not match, the call is likely to result in an exception. For example, if you use the **built-in function** `%VAL` to declare an argument as passed by value, but the C function is expecting a pointer, a segmentation violation may occur. (`%VAL` and `%REF` are HP Fortran extensions; for information about using them when calling a C routine from Fortran, see "Argument-passing conventions" on page 168.)

In most cases, debugging requires locating the code that caused the segmentation violation and rewriting it. If your program aborts with this error, recompile it with the `+fp_exception` option. A program compiled with this option will display a procedure traceback when it aborts. The procedure traceback lists procedure names and offset addresses of the code that caused the exception.

If you suspect that an out-of-bounds array reference is causing the segmentation violation, you can use the `+check=all` option instead of the `+fp_exception` option. When compiled with the `+check=all` option, a program that attempts to reference an array element that is outside the declared array boundary will abort with an error message that gives the line number of where the reference was detected.

The `+check=all` also performs runtime checks for out-of-bounds substrings and for **integer overflow**; see "Calling a trap procedure" on page 125. The `+check` option is fully described in the *HP Fortran Programmer's Reference*.

Bad argument exception

This exception occurs when a bad argument (for example, an out-of-range argument) is passed to a kernel system routine. This exception can also occur in programs that make explicit calls to the kernel threads library, `/usr/lib/libpthread.sl`, and pass bad arguments.

Using debugging lines

An HP Fortran program that has been written in fixed source form can contain debugging lines. These are statements that begin with the letter `D` or `d` in column 1. When compiled with the `+dlines` option, the debugging lines are treated as statements and compiled; otherwise, they are treated as comments and ignored. A program that contains debugging lines must also be compiled for fixed source form; that is, the filename extension must be either `.f` or `.F`, or the program must be compiled with the `+source=fixed` option.

The `+dlines` option makes it possible to include `WRITE` statements as debugging lines in the source file and to remove them from the production version of the program without having to change source code. Instead of deleting the `WRITE` statements when you are ready to build the production version, you recompile without the `+dlines` option, or with the `+nodlines` option.

Although debugging lines are supported by many implementations of Fortran (especially FORTRAN 77), it is nonstandard and therefore nonportable. Use of this feature is even more restrictive by reason of its being incompatible with free source form. If you try to compile a Fortran 90 program as free source form and the program contains debugging lines, the compilation will almost certainly fail with syntax errors.

The C preprocessor (`cpp`) provides a set of directives that have the same functionality as debugging lines but are much more powerful and can be used in either fixed or free source form. Although the `cpp` directives are not part of standard Fortran 90, they are available on most UNIX systems, such as HP-UX.

The `cpp` directives are described in the *cpp*(1) man page. See the *HP Fortran Programmer's Reference* for information about the source form of HP Fortran programs and the `+dlines` option.

5 Using the ON statement

Whenever a runtime error occurs, the default action of your program depends on the type of the error. If the error results from a floating-point exception, the program will continue to execute. Other errors will cause it to abort.

As described in "Handling runtime exceptions" on page 111, the `+fp_exception` and `+FP` options provide control over how a program behaves when a runtime error occurs. The `ON` statement provides an additional level of control by enabling your program to handle floating-point and integer exceptions and +Ctrl-C interrupts. Before an exception can be handled, the flow of control must pass through an `ON` statement that specifies:

- The type of the exception
- One of the following actions:
  - Execute a trap procedure
  - Ignore the interrupt
  - Abort the program

The action specified by the `ON` statement can only be changed by another `ON` statement that specifies the same exception.

This chapter describes how to use the `ON` statement. The syntax of the `ON` statement is described in the *HP Fortran Programmer's Reference*. For detailed information about trapping math errors, see the *HP-UX Floating-Point Guide*.

NOTE If you include the `ON` statement in a program that you optimize at level 2 or higher and the program takes an exception, the results may vary from those you would get from an unoptimized program or from a program that didn't have the `ON` statement.

Exceptions handled by the ON statement

Like the `+fp_exception` option, the `ON` statement enables traps for floating-point exceptions (by default, traps for floating-point exceptions are disabled on HP 9000 computers). When traps are enabled, an executing program that takes any of the following exceptions will abort, unless an `ON` statement specifies a different action:

- Division by zero
- Overflow
- Underflow
- Inexact result
- Invalid (or illegal) operation

These exceptions are defined by the IEEE standard for floating-point operations. The `ON` statement enables traps for these exceptions, regardless of whether the exception is taken by user code or by a call to a library routine. In addition, the `ON` statement also enables traps for integer division by zero, integer overflow, and +Ctrl-C interrupts. The +Ctrl-C interrupt occurs when the user presses +Ctrl-C during program execution.

Table 22 on page 121 lists the exceptions handled by the `ON` statement and gives the keywords that must be specified in the `ON` statement to indicate the exception being handled. The first column indicates the type of exception. The second column gives the keywords that must appear in the `ON` statement, immediately following the word `ON`. The third column gives alternate keywords you can specify instead of those in the second column.

For example, the following `ON` statement will trap attempts to divide by zero with 8-byte floating-point operands:

```
ON REAL(8) DIV 0 CALL div_zero_trap
```

The next example `ON` statement does the same as the first but uses the alternate keywords from the third column of the table:

```
ON DOUBLE PRECISION DIV 0 CALL div_zero_trap
```

Table 22 Exceptions handled by the ON statement

Exceptions	Exception keywords	Alternate keywords
Division by zero	REAL(4) DIV 0	REAL DIV 0
	REAL(8) DIV 0	DOUBLE PRECISION DIV 0
	REAL(16) DIV 0	(none)
	INTEGER(2) DIV 0	INTEGER*2 DIV 0
	INTEGER(4) DIV 0	INTEGER DIV 0
Overflow	REAL(4) OVERFLOW	REAL OVERFLOW
	REAL(8) OVERFLOW	DOUBLE PRECISION OVERFLOW
	REAL(16) OVERFLOW	(none)
	INTEGER(2) OVERFLOW	INTEGER*2 OVERFLOW
	INTEGER(4) OVERFLOW	INTEGER OVERFLOW
Underflow	REAL(4) UNDERFLOW	REAL UNDERFLOW
	REAL(8) UNDERFLOW	DOUBLE PRECISION UNDERFLOW
	REAL(16) UNDERFLOW	(none)
Inexact result	REAL(4) INEXACT	REAL INEXACT
	REAL(8) INEXACT	DOUBLE PRECISION INEXACT
	REAL(16) INEXACT	(none)
Invalid (illegal) operation	REAL(4) ILLEGAL	REAL ILLEGAL
	REAL(8) ILLEGAL	DOUBLE PRECISION ILLEGAL
	REAL(16) ILLEGAL	(none)
+Ctrl-C interrupt	CONTROLC	(none)

Actions specified by ON

The action taken after an exception is trapped depends on the action specified by the most recently executed ON statement for that exception. To specify an action, the ON statement must include the keyword ABORT, IGNORE, or CALL. These keywords have the following meanings:

- If ABORT is specified, a standard error message is generated and the program is aborted.
- If IGNORE is specified, processing continues with the next instruction.

  If the exception is an integer division by zero, the result is set to zero. For other conditions, the previous content of the target register is supplied as the result.

  IGNORE is particularly useful for preventing +Ctrl-C interrupts at inconvenient times during program execution.
- If CALL is specified, the normal (ABORT) error message is suppressed, and control is transferred to the specified trap procedure.

  Zero or one parameter is passed to the trap procedure. If an argument is specified, it is the result of the operation that took the exception. The procedure can analyze this value to get more precise information, and it can assign another value to the parameter to recover from the error. The type of the argument must be the same as that specified in the keywords.

  The specified trap procedure is generally an external procedure. However, it is also possible to specify a dummy procedure argument.

The following sections describe how to use the ON statement to specify different actions to take in the event of an exception.

Terminating program execution

Use the ABORT form of the CALL statement to terminate the program when an exception occurs. In the following example, the log is taken of a negative number. The ABORT clause causes the program immediately after the exception is detected and to issue a procedure traceback:

abort.f90

```
PROGRAM main
  REAL :: x, y, z
  ! The next statement enables traps for floating-point
exceptions
  !   and specifies the action to take for divide by zero.
!  ON REAL DIV 0 ABORT
  x = 10.0
  y = 0.0
  z = x / y
  PRINT *, y
END PROGRAM main
```

Here is the command line and the output from a sample run:

```
$ f90 abort.f90
$ a.out
PROGRAM ABORTED :  IEEE divide by zero

PROCEDURE TRACEBACK:

( 0)  0x0000248c   _start + 0x6c  [./a.out]
```

The program would have the same result if you were to comment out the `ON` statement and compile with the `+fp_exception` option.

Ignoring errors

You can use the `ON` statement to ignore an exception by specifying the `IGNORE` keyword. The following paragraphs discuss an example program, `ignore.f90`, that uses the `ON` statement to ignore an invalid operation. The following program illustrates this.

ignore.f90

```
PROGRAM main
  REAL :: x, y, z

  ! The following ON statement enables traps for floating-point
  !   exceptions and causes the program to ignore an invalid
  !   operation exception.
  ON REAL ILLEGAL IGNORE

  ! The next two statements pass a negative argument to the LOG
  !   intrinsic, resulting in an invalid operation.  This
  !   exception is ignored, as specified by the ON statement.
  x = -10.0
  y = LOG(x)

  PRINT *, y

  ! The next three statements attempt to divide by zero.  The
  !   trap for this exception is enabled by the previous
```

```
  !   ON statement but no action is specified.  Therefore,
  !   the program will abort execution.
  x = 9.0
  y = 0
  z = x/y

  PRINT *, z

END PROGRAM main
```

As defined by the IEEE standard, a floating-point operation that results in a **NaN** is an exception known as an **invalid operation**. The example program performs an invalid operation when it passes a negative argument to the LOG intrinsic, causing the intrinsic to return a NaN. The following ON statement:

```
ON REAL INVALID IGNORE
```

causes the program to ignore this exception and continue execution.

The program also attempts to divide by zero. Although the ON statement enables the trap triggered by a divide-by-zero exception, the statement has no other effect. As a result, the exception will cause the program to abort. To ignore the divide-by-zero exception would require an additional ON statement:

```
ON REAL DIV 0 IGNORE
```

Here is command line to compile the program, followed by the output from a sample run:

```
$ f90 ignore.f90
$ a.out
 NaN
PROGRAM ABORTED :  IEEE divide by zero

PROCEDURE TRACEBACK:

( 0)  0x00002504   _start + 0xbc  [./a.out]
```

Calling a trap procedure

You can write trap procedures that are callable by the ON statement to handle arithmetic errors in user code and in library routines. Trap procedures can take zero or one argument. If an argument is specified, it is the result and must have the type specified by the exception keyword. For example, if the following ON statement occurs in a program:

```
ON DOUBLE PRECISION OVERFLOW CALL trap
```

then the procedure trap could declare one argument of type DOUBLE PRECISION. Note that the argument is optional. Also, depending on the exception, the contents of the argument may not always be meaningful.

The following sections discuss two example programs that use the ON statement to call a trap procedure for floating-point exception and for an integer exception.

Trapping floating-point exceptions

The following program, call_fptrap.f90, causes an invalid operation exception and includes an ON statement to handle the exception. The ON statement calls the trap procedure trap_illegal, which assigns a different value to the result argument. The program prints the result. Here is the program listing:

call_fptrap.f90

```
PROGRAM main
  REAL :: x, y
  ON REAL ILLEGAL CALL trap_illegal
  x = -10.0
  y = LOG(x)  ! causes an invalid operation
  PRINT *, y
END PROGRAM main

SUBROUTINE trap_illegal(res)
  ! res is the result value of the invalid operation
  !   trapped by the ON statement
  REAL :: res
  res = 99.87   ! assign another value to the result argument
END SUBROUTINE trap_illegal
```

Here is the command line, followed by the output from a sample run:

```
$ f90 call_fptrap.f90
$ a.out
 99.87
```

Upon exit from a trap procedure, control returns to the instruction following the one that activated the trap, regardless of whether the erring instruction appears in user code or in a library routine.

Without the ON statement, this program would never execute its trap procedure and output a NaN, as shown by the output from a similar program in "Ignoring errors" on page 123.

Trapping integer overflow exceptions

This section discusses an example program that illustrates how to use the ON statement to call a trap procedure for an integer overflow exception.

An integer overflow occurs when an operation on an integer variable results in the attempt to assign it an out-of-range value. HP Fortran does not trap this exception by default. However, you can use the ON statement in conjunction with the HP CHECK_OVERFLOW directive to trap an integer overflow. The following program, call_itrap.f90, illustrates how to do this:

call_itrap.f90

```
PROGRAM main
!$HP$ CHECK_OVERFLOW INTEGER ON

  INTEGER :: i

  ON INTEGER OVERFLOW CALL trap_oflow

  ! assign to i the biggest number it can hold
  i = 2147483647
  ! now add 1
  i = i + 1
  PRINT *, i
END PROGRAM main

SUBROUTINE trap_oflow(n)
  INTEGER :: n

  ! write error message to standard error
  WRITE (7, *) 'integer overflow occurred, assigning 0 to result'
  n = 0
END SUBROUTINE trap_oflow
```

Here is the command line, followed by the output from a sample run:

```
$ f90 call_itrap.f90
$ a.out
 integer overflow occurred, assigning 0 to result
 0
```

If you were to comment out the `ON` statement but keep the directive, the program would abort with a procedure traceback and a core dump. Compiling with the `+check=all` option would have the same effect.

Trapping +Ctrl-C trap interrupts

A +Ctrl-C interrupt can occur during the following circumstances:

- When the user enters the interrupt code from the terminal while the program is running or awaiting input
- During the execution of a `PAUSE` statement

The trap procedure for a +Ctrl-C interrupt must have no formal arguments. The interrupt code is the character defined by the HP-UX *stty*(1) command for the `intr` parameter. The system default for `intr` is +Ctrl-C.

You can use the +Ctrl-C form of the `ON` statement to handle the interrupt signal 2. In the following example, when an interrupt occurs, the program reports status information on standard output, assuring the user that the program is still at work in the `DO` loop. The program uses the `ON` statement to set the action for a +Ctrl-C interrupt to be the call to the trap handler `status`:

```
PROGRAM main
  COMMON i
  ON CONTROLC CALL status

  DO i = 1, 100000
  ...               ! Long computation
  END DO
END

SUBROUTINE status
  COMMON i
  PRINT *, 'Currently on iteration ', i
END SUBROUTINE status
```

When this program is run, a +Ctrl-C interrupt causes the `status` routine to be called, which prints the iteration count. The program then resumes executing the `DO` loop.

Allowing core dumps

If a program includes the ON statement and takes an exception other than the one specified by the exception keywords, the program will abort with a procedure traceback but without a core dump. If you want to allow a core dump for one or more signals for a program that includes the ON statement, you must revise the program for each such signal.

For example, you may wish to handle floating-point exceptions with the ON statement, but still allow a core dump for other signals (for example, a bus error). The following example program uses the SIGNAL routine in the libU77 library to reset the default behavior for a bus error signal. The program uses the ON statement to handle floating-point exceptions, but allows a core dump when a bus error occurs:

allow_core.f90

```
PROGRAM main
  ON REAL OVERFLOW IGNORE
  CALL take_err
END PROGRAM main

SUBROUTINE take_err
  DOUBLE PRECISION :: d
  POINTER (ip, d) ! Cray-style pointer
  REAL :: x, y
  INTEGER, PARAMETER :: sigbus=10, sigdfl=0
  INTEGER :: sigrtn, SIGNAL

  ! Set the action for bus error to be the default (DUMP CORE),
  !   overriding the action of issuing a procedure traceback
  !   that is established by using the ON statement.
  ! To suppress the core dump and enable a procedure traceback,
  !   comment out the next statement

  sigrtn = SIGNAL(sigbus, 0, sigdfl)

  x = 1.0E38
  x = y * 10.0       ! causes a real overflow

  ! Bus error is caused by the next statements
  ip = MALLOC(40)
  ip = ip + 4        ! ip is now 4-byte aligned
  d = 99.0           ! bus error
END SUBROUTINE take_err
```

This program must be compiled with the `+U77` option to link in the `libU77` library. Here is the command line and the output from a sample run:

```
$ f90 +U77 allow_core.f90
$ a.out
Bus error(coredump)
$ ls core
core
```

6 Performance and optimization

This chapter describes how to use different features of the HP Fortran to tune your program for optimum performance. The most important of these features is the optimizer. You invoke the optimizer when compiling your program by specifying either `+O`*n* (where *n* represents the level of optimization to be applied to your program) or the `-O` option for the default level of optimization (level 2). The `-O` option is provided for compatibility with the POSIX standard and has the same functionality as the `+O2` option.

The following command line compiles `prog.f90`, using the default level of optimization:

```
$ f90 -O prog.f90
```

For most applications, `-O` provides effective optimization. However, some applications can realize significant increases in performance at higher levels of optimization or when you use other features of the optimizer to boost performance. This chapter discusses these features as well as the following topics:

- Using profilers
- Using options to control optimization
- Conservative vs. aggressive optimization
- Parallelizing HP Fortran programs
- Vectorization
- Controlling code generation for performance

For information about getting the best performance from floating-point intensive applications running on HP-UX, see the *HP-UX Floating-Point Guide*.

Using profilers

A **profiler** is a tool for sampling a program during execution so that you can determine where your program spends most of its time. After examining the data provided by a profiler, you can decide whether to redesign parts of the program to improve their performance or to re-compile the program with optimization options. For example, if your program contains a loop with an embedded call and profiling reveals that the program spends much of its time in the loop, you may decide to inline the embedded call.

The following sections describe the **CXperf** performance analysis tool, which is bundled with HP Fortran as well as the two UNIX profilers, `gprof` and `prof`.

NOTE As described in "Stripping debugging information" on page 110, all programs compiled by HP Fortran include symbol table information in the executable file, unless you compile with the `+strip` option or have removed the symbol table with the `strip` utility. This information must be present in the executable in order to use the profiling tools.

CXperf

When working on HP V-Class systems, you can use the CXperf profiler to get loop-level and routine-level information on HP Fortran programs. For CXperf support, compile using the `+pa` option (for routine-level data) or the `+pal` option (for loop-level and routine-level data). For example:

```
% f90 +pal foo.f
```

The `+pa` and `+pal` options cause HP Fortran to run `cxoi` (the CXperf object instrumentor) as part of the compilation process to create an executable program that supports CXperf's methods of collecting statistics.

To collect profile statistics for a program that was compiled with `+pa` or `+pal`, run CXperf and specify the executable program you want to profile. For example:

```
% /opt/cxperf/bin/cxperf a.out
```

CXperf creates a profile of a program by collecting information on the wall clock time and CPU time spent per routine (and, if requested, per loop). It also can gather statistics on cache hits and misses and other aspects of the program's execution, such as the sequence in which routines are called (viewable as a graphical "call graph").

More information about CXperf is available from its Help menu.

gprof

The `gprof` profiler enables you to determine which subprograms are called the most and how much time is spent in each subprogram.To use `gprof`, do the following:

1 Compile the program with the `+gprof` option. For example:

```
$ f90 -o prog +gprof prog.f90
```

2 Run the program. This creates the file `gmon.out` in the current directory. For example:

```
$ prog
$ ls gmon.out
gmon.out
```

3 Run `gprof`, specifying the name of the program as an argument. It will display two tables to standard output: a flat profile and a call graph profile. Since these tables can be quite large, you may want to redirect the output from `gprof`, as follows:

```
$ gprof prog >gprof.out
```

The *flat profile* lists the number of times each subprogram was called and the percentage of the total execution time for each of the subprogram times. The *call graph profile* includes such information as the index of the function in the call graph listing, the percentage of total time of the program accounted for by a routine and its descendents, and the number of seconds spent in the routine itself.

4 Once `gprof` is finished, you can view the output tables using an ASCII editor.

For more information about `gprof`, see the *gprof*(1) man page.

prof

The `prof` profiler can also be used for profiling. Unlike the `gprof` profiler, `prof` does not generate the call graph profile. To use `prof`, do the following:

1 Compile the program with the `+prof` option. For example:

```
$ f90 -o prog +prof prog.f90
```

2 Run the program. This creates a file named `mon.out` in the current directory. For example:

```
$ prog
$ ls mon.out
mon.out
```

3 Run `prof`, giving the name of the program as an argument, as follows:

```
$ prof prog
```

`prof` produces a listing on standard output showing the time spent in each routine.

For more information about `prof`, see the *prof*(1) man page.

Using options to control optimization

HP Fortran includes a rich set of command-line options for controlling optimization. For most applications, we recommend optimizing with `-O`, which enables the default level of optimization. (For information about the default level of optimization, refer to Table 23 on page 136; look up `+O2` in the first column.) You can raise or lower the level of optimization with the `+O`*opt-level* option, and you can use the `+O`*optimization* option to control the kinds of optimizations that are available at each level.

The following sections describe how to use the `+O`*opt-level* and `+O`*optimization* options. For detailed descriptions of the optimization options, see the *HP Fortran Programmer's Reference*.

Using +O to set optimization levels

HP Fortran provides four levels of optimization. Each higher level is a superset of the lower levels; level 4 is the highest level and can result in a significant increase in program performance. Level 2 is the default level of optimization.

You invoke optimization by compiling with the `+O`*opt-level* option, where *opt-level* is an integer in the range 0 - 4. The following command line invokes the optimizer at the highest level:

```
$ f90 +O4 file.f90
```

You can invoke level 2 (the default level) by specifying the `-O` option.

Table 23 summarizes each level, giving the option that invokes that level, the advantages, disadvantages, and recommended usages. For technical information about the specific optimizations at each level, refer to the *HP PA-RISC Compiler Optimization Technology White Paper*. A PostScript version of this document is available online in `/opt/langtools/newconfig/white_papers/optimize.ps`.

NOTE You can debug programs optimized up to level 2. To prepare an optimized program for debugging, use the command line:

```
$ f90 -g +Oopt-level prog.f90
```

where *opt-level* is an integer in the range 0-2. If you use the `-g` option at a higher level of optimization, the compiler lowers the level to 2 and compiles for debugging.

Table 23 Optimization levels

Option	Optimizations performed	Advantages	Disadvantages	Recommended use
`+O0` default	Constant folding and partial evaluation of test conditions.	Compiles fastest; compatible with the debugger option `-g`.	Does very little optimization.	During program development.
`+O1`	Level 0 optimizations, plus branch optimization, dead code elimination, more efficient use of registers, instruction scheduling, and peephole optimization.	Produces faster programs than level 0; compiles faster than level 2; compatible with the debugger option `-g`.	Compiles slower than level 0.	During program development.
`+O2`, `-O`	Default level optimizations, including level 1, plus coloring register allocation, induction variable elimination and strength reduction, common subexpression elimination, loop invariant code motion, store/copy optimization, unused definition elimination, software pipelining, and register reassociation.	Can significantly increase performance over level 1; works with debugger option `-g`.	Compiles slower than level 0 and 1.	During program development and when building the production version; especially effective in optimizing loops that perform arithmetic operations on large float and double arrays.

Option	Optimizations performed	Advantages	Disadvantages	Recommended use
`+O3`	Level 2 optimizations, plus loop transforms, parallelization, vectorization, cloning, and inlining within a file. Some optimizations may require additional options; see "Using the optimization options" on page 137.	Can significantly increase performance over level 2.	Compiles slower than lower levels; increases object code size; not compatible with the debugger option `-g`.	When building the production version; especially effective when used on source files containing frequently executed loops and subprograms.
`+O4`	Level 3 optimizations applied across all program files compiled with `+O4`.	Provides the highest level of optimization; can significantly increase performance over level 3.	Can use large amounts of system resources; may increase link-time and object code size; not compatible with the debugger option `-g`.	When building the production version; especially effective when used on source files containing frequently executed loops and subprograms.

Using the optimization options

The `+O`*optimization* options enable you to control the kind of optimizations that are applied to your program at each level. Table 24 on page 138 and Table 25 on page 139 list the options. The first column of each table lists each option, the second column gives the optimization level at which the option can be used, and the third column identifies what the option does. When using any of these options except `+Oall`, you must also use the `+O`*n* option to specify the optimization level listed in the second column of the tables. The `+Oall` option automatically invokes the optimizer at the highest level.

Table 24 lists the "packaged" options. These options enable or disable a set of related optimizations, such as optimizations that do not increase code size. Table 25 lists options that enable or disable specific optimizations.

The options in both tables can be combined on the same command line, except as noted. For example, the following command line requests aggressive optimizations at level 2 that do not increase code size:

```
$ f90 +02 +Oaggressive +Osize prog.f90
```

Nearly all of the optimization options can be used to enable or disable an optimization or a package of optimizations. For example, the following command line requests aggressive level 4 optimizations that do not result in **roundoff errors**:

```
$ f90 +O4 +Oaggressive +Ofltacc prog.f90
```

The *Parallel Programming Guide for HP-UX Systems* fully describes all of the optimization options.

Table 24 Packaged optimization options

Option	Level	Function
`+O[no]aggressive`	`+O2` or higher	Enable [disable] optimizations that can significantly improve performance in standard-conforming programs. The default is `+Onoaggressive`. For more information about this option, see "Conservative vs. aggressive optimization" on page 142.
`+O[no]all`	Invokes highest level	Enable [disable] maximum optimization. The default is `+Onoall`.
`+O[no]conservative`	`+O2` or higher	Suppress [do not suppress] optimizations that assume strict conformity to the Fortran 90 standard. The default is `+Onoconservative`. For more information about this option, see "Conservative vs. aggressive optimization" on page 142.
`+O[no]limit`	`+O2` or higher	Enable [disable] optimizations that do not make large demands on system resources. The default is `+Onolimit`.

Option	Level	Function
`+O[no]size`	`+O2` or higher	Enable [disable] optimizations that do not significantly increase code size. The default is `+Onosize`.

Table 25 Fine-tuning optimization options

Option	Level	Function
`+O[no]cache_pad_common`	`+O3` or higher	Pad [do not pad] common blocks to avoid cache collisions. The default is `+Onocache_pad_common`.
`+O[no]dataprefetch`	`+O2` or higher	Insert [do not insert] instructions within innermost loops to explicitly prefetch data from memory into the data cache. The default is `+Onodataprefetch`.
`+O[no]entrysched`	All	Perform [do not perform] instruction scheduling on entry and exit code. The default is `+Onoentrysched`.
`+O[no]fastaccess`	All	Enable [disable] fast access to global data. The default is `+Onofastaccess` at levels 1, 2, and 3; `+Ofastaccess` at level 4.
`+O[no]fltacc`	`+O2` or higher	Disable [enable] floating-point optimizations that can result in numerical differences. By default, the optimizer does not perform such optimizations. For information about the effect this option can have on your program, refer to the *HP-UX Floating-Point Guide*.
`+O[no]info`	All	Display [do not display] information about the optimization process. This option is most useful at level 3 and above. The default is `+Onoinfo`.

Option	Level	Function
`+O[no]initcheck`	`+O2` or higher	Enable [disable] initialization of any local, scalar, automatic variable that is found to be uninitialized. The default is to initialize if the variable is uninitialized with respect to every path leading to its use. For more information about this option, see "Uninitialized variables" on page 226.
`+O[no]inline`	`+O3` or higher	Enable [disable] inlining. The default is `+Oinline`.
`+Oinline_budget=`*n*	`+O3` or higher	Perform more aggressive inlining, as specified by *n*. The default is `+Oinline_budget=100`.
`+O[no]libcalls`	All	Substitute [do not substitute] **millicode** versions of specific intrinsics. The default is `+Olibcalls`.
`+O[no]loop_unroll=`*n*	`+O2` or higher	Unroll [do not unroll] program loops by a factor of *n*. The default is `+Oloop_unroll=4`.
`+O[no]moveflops`	`+O2` or higher	Enable [disable] moving conditional floating-point instructions out of loops. The default is `+Omoveflops`.
`+O[no]parallel`	`+O3` or higher	Transform [do not transform] eligible loops for parallel execution. The default is `+Onoparallel`.
`+O[no]parmsoverlap`	`+O2` or higher	Suppress optimizations that assume [do not assume] that arguments may refer to the same memory locations. The default is `+Onoparmsoverlap`.
`+O[no]pipeline`	`+O2` or higher	Enable [disable] software pipelining. The default is `+Opipeline`.

Option	Level	Function
`+O[no]procelim`	All	Remove [do not remove] unreferenced procedures from the executable. The default is `+Onoprocelim` at levels 0 - 3, `+Oprocelim` at level 4.
`+O[no]regreassoc`	`+O2` or higher	Enable [disable] register association. The default is `+Oregreassoc`.
`+O[no]vectorize`	`+O32` or higher	Replace [do not replace] eligible loops with calls to the math library; for more information, see "Using the +Ovectorize option" on page 149. The default is `+Onovectorize`.

Conservative vs. aggressive optimization

At optimization level 2 or higher, the optimizer makes a number of assumptions about the program it is optimizing—for example, that re-ordering an expression for improved instruction scheduling will not change its results. In general, these assumptions relate to how closely the target program conforms to the Fortran 90 Standard. For programs that conform to the Standard, it is safe for the optimizer to apply certain optimizations that can significantly improve performance. For nonstandard-conforming programs, these same optimizations could change the results or behavior of the program in ways that may not be acceptable to the programmer.

The `+Oconservative` and `+Oaggressive` options enable you to set the optimizer's assumptions about which optimizations it can and cannot apply to a program. Each option invokes a subset of the fine-tuning options that balances safety and performance according to the coding style of the target program. You can use either option at optimization level 2 or higher.

NOTE `+Oaggressive` and `+Oconservative` are incompatible and must not appear on the same command line.

Table 26 on page 143 lists the assumptions that the optimizer makes about your program when you compile with `+Oconservative`, `+Oaggressive`, or neither option (the default). The table also lists the fine-tuning options that are invoked by `+Oconservative` and `+Oaggressive`. The options listed for the default case are the subset of the ones invoked by `+Oconservative` and `+Oaggressive`. For information about the fine-tuning options listed in the third column, see Table 25 on page 139.

Table 26 Conservative, aggressive, and default optimizations

Specified options	Assumptions	Invoked options
`+Onoconservative` `+Onoaggressive` (the default)	• Standard-conforming	`+Onoentrysched` `+Omoveflops` `+Onoparmsoverlap` `+Onovectorize`
`+Oconservative`	• Nonstandard • Sensitive to rounding differences • Contains floating-point expressions that must be evaluated in the specified order • Procedure arguments may overlap	`+Ofltacc` `+Onomoveflops` `+Oparmsoverlap`
`+Oaggressive`	• Standard-conforming • Contains floating-point expressions that permit re-ordering for optimization • Does not contain uninitialized variables	`+Oentrysched` `+Onofltacc` `+Onoinitcheck` `+Ovectorize`

Parallelizing HP Fortran programs

The following sections discuss how to use the `+Oparallel` option and the parallel directives when preparing and compiling HP Fortran programs for parallel execution. Later sections also discuss reasons why the compiler may not have performed parallelization. The last section describes runtime warning and error messages unique to parallel-executing programs.

For a description of the `+Oparallel` option, see "Fine-tuning optimization options" on page 54.

Compiling for parallel execution

The following command lines compile (without linking) three source files: `x.f90`, `y.f90`, and `z.f90`. The files `x.f90` and `y.f90` are compiled for parallel execution. The file `z.f90` is compiled for serial execution, even though its object file will be linked with `x.o` and `y.o`.

```
f90 +O3 +Oparallel -c x.f90 y.f90
f90 +O3 -c z.f90
```

The following command line links the three object files, producing the executable file `para_prog`:

```
f90 +O3 +Oparallel -o para_prog x.o y.o z.o
```

As this command line implies, if you link and compile separately, you must use `f90`, not `ld`. The command line to link must also include the `+Oparallel` and `+O3` options in order to link in the parallel runtime support.

Performance and parallelization

To ensure the best runtime performance from programs compiled for parallel execution on a multiprocessor machine, do not run more than one parallel program on a multiprocessor machine at the same time. Running two or more parallel programs simultaneously may result in their sharing the same processors, which will degrade performance. You should run a parallel-executing program at a higher priority than any other user program; see *rtprio*(1) for information about setting real-time priorities.

Running a parallel program on a heavily loaded system may also slow performance.

Profiling parallelized programs

You can profile a program that has been compiled for parallel execution in much the same way as for non-parallel programs:

1 Compile the program with the `+gprof` option.

2 Run the program to produce profiling data.

3 Run `gprof` against the program.

4 View the output from `gprof`.

The differences are:

- Step 2 produces a `gmon.out` file with the CPU times for all executing threads.
- In Step 4, the flat profile that you view uses the following notation to denote `DO` loops that were parallelized:

  routine_name`##pr_line_`*nnnn*

  where *routine_name* is the name of the routine containing the loop, `pr` (parallel region) indicates that the loop was parallelized, and *nnnn* is the line number of the start of the loop.

Conditions inhibiting loop parallelization

The following sections describe conditions that can cause the compiler not to parallelize. These include the following:

- Calling routines with side effects
- Indeterminate iteration counts
- Data dependences

Calling routines with side effects

The compiler will not parallelize any loop containing a call to a routine that has side effects. A routine has side effects if it does any of the following:

- Modifies its arguments
- Modifies a global, common-block variable, or save variable
- Redefines variables that are local to the calling routine
- Performs I/O
- Calls another subroutine or function that does any of the above

You can use the `DIR$ NO SIDE EFFECTS` directive to force the compiler to ignore side effects when determining whether to parallelize the loop. For information about this directive, see .

NOTE A subroutine (but not a function) is always expected to have side effects. If you apply this directive to a subroutine call, the optimizer assumes that the call has no effect on program results and can eliminate the call to improve performance.

Indeterminate iteration counts

If the compiler finds that a runtime determination of a loop's iteration count cannot be made before the loop starts to execute, the compiler will not parallelize the loop. The reason for this precaution is that the runtime code must know the iteration count in order to determine how many iterations to distribute to the executing processors.

The following conditions can prevent a runtime count:

- The loop is a `DO`-forever construct.
- An `EXIT` statement appears in the loop.
- The loop contains a conditional `GO TO` statement that exits from the loop.
- The loop modifies either the loop-control or loop-limit variable.
- The loop is a `DO WHILE` construct and the condition being tested is defined within the loop.

Data dependences

When a loop is parallelized, the iterations are executed independently on different processors, and the order of execution will differ from the serial order when executing on a single processor. This difference is not a problem if the iterations can occur in any order with no effect on the results. Consider the following loop:

```
DO I = 1, 5
    A(I) = A(I) * B(I)
END DO
```

In this example, the array `A` will always end up with the same data regardless of whether the order of execution is 1-2-3-4-5, 5-4-3-2-1, 3-1-4-5-2, or any other order. The independence of each iteration from the others makes the loop an eligible candidate for parallel execution.

Such is not the case in the following:

```
DO I = 2, 5
    A(I) = A(I-1) * B(I)
END DO
```

In this loop, the order of execution does matter. The data used in iteration `I` is dependent upon the data that was produced in the previous iteration (`I-1`). The array `A` would end up with very different data if the order of execution were any other than 2-3-4-5. The data dependence in this loop thus makes it ineligible for parallelization.

Not all data dependences inhibit parallelization. The following paragraphs discuss some of the exceptions.

Nested loops and matrices

Some nested loops that operate on matrices may have a data dependence in the inner loop only, allowing the outer loop to be parallelized. Consider the following:

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

The data dependence in this nested loop occurs in the inner (`J`) loop: each row access of `A(J,I)` depends upon the preceding row (`J-1`) having been assigned in the previous iteration. If the iterations of the `J` loop were to execute in any other order than the one in which they would execute on a single processor, the matrix would be assigned different values. The inner loop, therefore, must not be parallelized.

But no such data dependence appears in the outer loop: each column access is independent of every other column access. Consequently, the compiler can safely distribute entire columns of the matrix to execute on different processors; the data assignments will be the same regardless of the order in which the columns are executed, so long as the rows execute in serial order.

Assumed dependences

When analyzing a loop, the compiler may err on the safe side and assume that what looks like a data dependence really is one and so not parallelize the loop. Consider the following:

```
DO I = 101, 200
    A(I) = A(I-K)
END DO
```

The compiler will assume that a data dependence exists in this loop because it appears that data that has been defined in a previous iteration is being used in a later iteration. On this assumption, the compiler will not parallelize the loop.

However, if the value of `K` is 100, the dependence is assumed rather than real because `A(I-K)` is defined outside the loop. If in fact this is the case, the programmer can insert one of the following directives immediately before the loop, forcing the compiler to ignore any assumed dependences when analyzing the loop for parallelization:

- `DIR$ IVDEP`
- `FPP$ NODEPCHK`
- `VD$ NODEPCHK`

For more information about these directives, see “Compatibility directives” on page 196.

Vectorization

When **vectorization** is enabled, the optimizer replaces eligible loops with calls to specially tuned routines in the math library. When you compile with the `+Ovectorize` option, the optimizer vectorizes wherever it determines that it is safe and feasible to do so. However, you can use directives to limit vectorization. As an alternative to the optimizer's automatic vectorization, you can make explicit calls to the Basic Linear Algebra Subroutine (BLAS) library to perform common vector and matrix operations.

The following sections describe how to use the vectorizing capabilities of the optimizer.

Using the +Ovectorize option

To enable vectorization, you must compile the program at optimization level 3 or higher and specify the `+Ovectorize` option, as in the following example command line:

```
f90 +O3 +Ovectorize prog.f90
```

When vectorization is enabled, the optimizer uses a pattern-matching algorithm to identify program loops as eligible for vectorization. If the optimizer can also determine that:

- Vectorization will produce the same results as the original loop
- There are no other optimizations that will yield better performance

the optimizer replaces the loop by a call to one of the math library routines listed in Table 27.

Table 27 Vector routines called by `+Ovectorize`

Vector routine	Description
`daxpy`	Add a scalar multiple of a vector to a vector, using double-precision operands.
`ddot`	Compute the dot product of two double-precision vectors.
`memcpy`	See the *memory*(1) man page.

Vector routine	Description
`memmove`	See the *memory*(1) man page.
`memset`	See the *memory*(1) man page.
`saxpy`	Add a scalar multiple of a vector to a vector, using single-precision operands.
`sdot`	Compute the dot product of two single-precision vectors.
`vec_damax`	Find the maximum absolute value in a double-precision vector.
`vec_dmult_add`	Multiply a scalar by a vector and add the result to the result vector, using double-precision operands.
`vec_dsum`	Sum the elements of a double-precision vector.

If your PA2.0 application uses very large arrays, compiling with both `+Ovectorize` and `+Odataprefetch` may also increase performance. The math library contains special prefetching versions of the vector routines that are called if you specify both options.

If you compile with the `+Ovectorize` and `+Oinfo` options, the optimizer will identify which loops it vectorized. If you find that the extent of vectorization is not significant, you may want to consider some other optimization, such as parallelization.

Controlling vectorization locally

When you compile with the `+Ovectorize` option, the optimizer considers all loops in the source file as candidates for vectorization. The `*$* [NO]VECTORIZE` directive enables you to limit vectorization. You use the `*$* NOVECTORIZE` form of the directive to disable vectorization and the `*$* VECTORIZE` form to enable it. The directive applies to the beginning of the next loop and remains in effect for the rest of the program unit or until superseded by a later directive. The directive is ignored if you do not compile with the `+Ovectorize` option and specify an optimization of 3 or higher.

For example, if a file containing the following code segment were compiled with `+Ovectorize`, only one loop would be considered as a candidate for vectorization:

```
! This is line 1 of the source file.
!*$* NOVECTORIZE
.
.
.
!*$* VECTORIZE
DO i = 1, 100
.
.
.
END DO
!*$* NOVECTORIZE
.
.
.
```

Note that the `*$* VECTORIZE` directive does not force vectorization. The optimizer vectorizes only if:

- The loop performs a vector operation recognized by the optimizer as in its repertoire.
- The loop is safe to vectorize. The same conditions that can prevent parallelization—see, for example, “Data dependences” on page 147—can also prevent vectorization.
- The optimizer can discover no other transformations that can result in better performance.

The only way to ensure vectorization is for the programmer to edit the source file and substitute an appropriate call to the BLAS library for the loop, as described in “Controlling vectorization locally” on page 150.

For a detailed description of the `*$* [NO]VECTORIZE` directive, see the *HP Fortran Programmer's Reference*.

Calling BLAS library routines

The HP Fortran compiler is bundled with the Basic Linear Algebra Subroutine (BLAS) library. This library consists of specially tuned routines that perform low-level vector and matrix operations that conform to a de facto, industry-wide standard[1]. The BLAS routines are widely available, making them portable across many implementations of Fortran.

HP Fortran includes a library of the BLAS routines that have been especially tuned for performance on PA-RISC machines. You can call any of these routines in an HP Fortran program by compiling it with the `-lblas` option.

Consider the following program, which contains a loop that performs an operation on two arrays that is identical to the `saxpy` routine in the BLAS library, as noted in the comments:

saxpy.f90

```
PROGRAM main

INTEGER :: i, inc_x, inc_y, dim_num
REAL, DIMENSION(5) :: x, y
REAL :: b

b = 3.0
dim_num = 5
inc_x = 1
inc_y = 1

! initialize the two arrays x and y
DO i = 1, 5
    y(i) = i
    x(i) = i + 3.0
END DO
PRINT *, y

! add a scalar multiple of x to y
DO i = 1, 5
   y(i) = y(i) + b * x(i)
END DO
PRINT *, y

END PROGRAM main
```

1. See the *LAPACK User's Guide*, ed. J. Dongarra *et al* (Philadelphia, 1992). Each of the BLAS routines has its own man page; see *blas*(3X) for an introduction. Also, see the URL: `http://www.netlib.org`.

The following command lines compile and execute the program, and show the output from a sample run:

```
$ f90 saxpy.f90
$ a.out
 1.0 2.0 3.0 4.0 5.0
 13.0 17.0 21.0 25.0 29.0
```

As an alternative, you could replace the second loop with the following call to the `saxpy` routine in the BLAS library:

```
CALL saxpy(dim_num, b, x, inc_x, y, inc_y)
```

When you compile the revised program, you must add the `-lblas` option to the end of the command line to link in the BLAS library. The following show the command lines to compile and execute the revised program as well as the output from a sample run:

```
$ f90 saxpy_blas.f90 -lblas
$ a.out
 1.0 2.0 3.0 4.0 5.0
 13.0 17.0 21.0 25.0 29.0
```

If you call a BLAS routine that is a function, be sure to declare the return value of the routine in a data declaration statement and specify the `EXTERNAL` attribute, as in the following:

```
REAL, EXTERNAL :: sdot
```

Fortran uses implicit typing by default. Unless a function is explicitly declared as having a certain type, the type is determined by the first character of the BLAS routine. If that character implies a type other than that of the returned value, the result will be meaningless.

See the *HP Fortran Programmer's Reference* for information about the BLAS library.

Controlling code generation for performance

For optimum performance, the executable program should consist of code that can take advantage of the hardware features of the machine on which the program will run. If your program will run on the same machine as you use to compile it, code generation is not an issue. By default, the HP Fortran compiler generates code for the model of the machine on which you are running the compiler.

However, if you are compiling on a different machine from the one on which the program will run, you should use the `+DA`*model* option to ensure that the compiler generates code based on the target architecture. For information about using this option, see "Compiling for different PA-RISC machines" on page 77.

7 Writing HP-UX applications

This chapter discusses how HP Fortran applications running on the HP-UX operating system can use system resources to do the following:

- Accessing command-line arguments
- Calling HP-UX system and library routines
- Using HP-UX file I/O

Accessing command-line arguments

When invoking an HP Fortran executable program, you can include one or more arguments on the command line. The operating system will make these available to your program. For example, the following command line invokes the program `fprog`:

```
$ fprog arg1 "another arg" 222
```

and it also passes three character arguments to the program:

```
arg1
another arg
222
```

An HP Fortran program can access these arguments for internal use by calling the `IGETARG` and `IARGC` intrinsics; `IGETARG` is available either as a function or a subroutine. The `IGETARG` intrinsic gets the specified command-line argument; `IARGC` returns the number of arguments on the command line. You can also use the `GETARG` intrinsic to return command-line arguments, as illustrated in the following example program:

get_args.f90

```
PROGRAM get_args

INTEGER, PARAMETER :: arg_num = 1

! arg_str is the character array to be written to
!   by IGETARG
CHARACTER(LEN=30) :: arg_str

! IGETARG returns number of characters read within
! the specified parameter
!    arg_num is the position of the desired argument in the
!        the command line (the name by which the program
!        was invoked is 0)
!    arg_str is the character array in which the argument
!        will be written
!    30 is the number of characters to write to arg_str
PRINT *, IGETARG(arg_num, arg_str, 30)
PRINT *, arg_str

! IARGC returns the total number of arguments on the
!   command line
PRINT *, IARGC()

END PROGRAM get_args
```

When compiled and invoked with the following command lines:

```
$ f90 get_args.f90
$ a.out perambulation of a different sort
```

this program produces the following output:

```
13
perambulation
5
```

For more information about the `IGETARG` and `IARGC` intrinsics, see the *HP Fortran Programmer's Reference*. `GETARGC` is also available as a `libU77` routine; see the *HP Fortran Programmer's Reference*.

Calling HP-UX system and library routines

System calls provide low-level access to kernel-level resources, such as the `write` system routine. or example, see “File handling” on page 181 for an example of a program that calls the `write` routine. For information about system calls, refer to the *HP-UX Reference*.

HP-UX library routines provide many capabilities, such as getting system information and file stream processing. Library routines are also discussed in the *HP-UX Reference*.

You can access many HP-UX system calls and library routines from HP Fortran programs using the BSD 3F library, `libU77.a`. Another library provided with HP Fortran is the Basic Linear Algebra Subroutine (BLAS) library, `libblas.a`. These subroutines perform low-level vector and matrix operations, tuned for maximum performance. See “Additional HP Fortran libraries” on page 69 for information about linking to these libraries. For detailed information about the both libraries, see the *HP Fortran Programmer's Reference*.

Using HP-UX file I/O

HP-UX file-processing routines can be used as an alternative to Fortran file I/O routines. This section discusses HP-UX stream I/O routines and I/O system calls.

Stream I/O using FSTREAM

The HP-UX operating system uses the term *stream* to refer to a file as a contiguous set of bytes. There are a number of HP-UX subroutines for performing **stream I/O**; see *stdio*(3S) in the *HP-UX Reference*.

Unlike Fortran I/O, which requires a logical unit number to access a file, stream I/O routines require a stream pointer—an integer variable that contains the address of a C-language structure of type `FILE` (as defined in the C-language header file `/usr/include/stdio.h`.)

The following Fortran statement declares a variable for use as a stream pointer in HP Fortran:

```
INTEGER(4) :: stream_ptr
```

To obtain a stream pointer, use the Fortran intrinsic `FSTREAM`, which returns a stream pointer for an open file, given the file's Fortran logical unit number:

stream-ptr `= FSTREAM(`*logical-unit*`)`

The *logical-unit* parameter must be the logical unit number obtained from opening a Fortran file, and *stream-ptr* must be of type integer. If *stream-ptr* is not of type integer, type conversion takes place with unpredictable results. The *stream-ptr* should never be manipulated as an integer.

Once you obtain *stream-ptr*, use the `ALIAS` directive to pass it by value to stream I/O routines. (For an example of how to use the `ALIAS` directive, see “File handling” on page 181.) All HP Fortran directives are described in the *HP Fortran Programmer's Reference*.)

Performing I/O using HP-UX system calls

File I/O can also be performed with HP-UX system calls (for example, `open`, `read`, `write`, and `close`), which provide low-level access to the HP-UX kernel. These routines are discussed in the *HP-UX Reference*; see also the online man pages for these routines. For an example program that shows how to call the `write` routine, see “File handling” on page 181.

Establishing a connection to a file

HP-UX I/O system calls require an HP-UX **file descriptor**, which establishes a connection to the file being accessed. A file descriptor is an integer whose function is similar to a Fortran logical unit number. For example, the following `open` system call (called from a C-language program) opens a file named `DATA.DAT` for reading and writing, and returns the value of an HP-UX file descriptor:

```
#include <fcntl.h>  /* definition of O_RDWR contained here */
   ...
fildes = open("DATA.DAT", O_RDWR)
```

Obtaining an HP-UX file descriptor

The Fortran intrinsic `FNUM` returns the HP-UX file descriptor for a given logical unit. See the program in“File handling” on page 181 for an example of how to call the `FNUM` intrinsic. For information about `FNUM`, see the *HP Fortran Programmer's Reference*.

8 Calling C routines from HP Fortran

This section describes language differences between C and HP Fortran that affect calling C routines from an HP Fortran program. This includes the following topics:

- Data types
- Argument-passing conventions
- Case sensitivity
- Arrays
- C strings
- File handling
- Sharing data

Data types

Table 28 lists the corresponding data types for HP Fortran and C when compiled as 32-bit applications.

Table 28 Data type correspondence for HP Fortran and C

HP Fortran	C
`CHARACTER`	`char` (array of)
Hollerith (synonymous with `CHARACTER`)	`char` (array of)
`BYTE`, `LOGICAL(KIND=1)`, `INTEGER(KIND=1)`	`char`
`LOGICAL(KIND=2)`	`short`
`INTEGER(KIND=2)`	`short`
`LOGICAL`, `LOGICAL(KIND=4)`	`long` or `int`
`INTEGER`, `INTEGER(KIND=4)`	`long` or `int`
`INTEGER(KIND=8)`	`long long`
`REAL`, `REAL(KIND=4)`	`float`
`DOUBLE PRECISION`, `REAL(KIND=8)`	`double`
`REAL(KIND=16)`	`long double`
`COMPLEX`, `COMPLEX(KIND=4)`	`struct`
`DOUBLE COMPLEX`, `COMPLEX(KIND=8)`	`struct`
derived type	`struct`

Using the `+DA2.0W` option to compile HP Fortran programs in 64-bit mode has no effect on Fortran data types; see “Compiling in 64-bit mode” on page 85. However, it does change the sizes of some C data types. If your program calls functions written in C and is compiled in 64-bit mode, you should be aware of the size discrepancies and either promote individual data items or recompile with the `+autodbl` option to promote all default integer, real, and logical items to 64-bits.

Table 29 shows the differences between the corresponding data types in HP Fortran and C when compiling in 32-bit mode and in 64-bit mode. Table 30 shows the differences when the Fortran program is compiled with the `+autodbl` option. Notice that Fortran data items that are explicitly sized (for example, `INTEGER*4`) stay the same size regardless of whether they are compiled in 32-bit mode, in 64-bit mode, or with the `+autodbl` option.

Table 29 Size differences between HP Fortran and C data types

HP Fortran data types	C data types		Sizes (in bits)
	32-bit mode	64-bit mode	
`INTEGER`	`int` or `long`	`int`	32
`INTEGER*4`	`int` or `long`	`int`	32
`INTEGER*8`	`long long`	`long` or `long long`	64
`REAL`	`float`	`float`	32
`DOUBLE PRECISION`	`double`	`double`	64
`REAL*16`	`long double`	`long double`	128

Table 30 Size differences after compiling with `+autodbl`

HP Fortran data types	C data types		Sizes (in bits)
	32-bit mode	64-bit mode	
`INTEGER`	`long long`	`long`	64
`INTEGER*4`	`int` or `long`	`int`	32
`INTEGER*8`	`long long`	`long`	64
`REAL`	`float`	`float`	64
`DOUBLE PRECISION`	`long double`	`long double`	128
`REAL*16`	`long double`	`long double`	128

The following sections provide more detailed information about language differences for the following data types:

- Unsigned integers
- Logicals
- Complex numbers
- Derived types

Unsigned integers

Unlike Fortran, C allows integer data types (`char`, `int`, `short`, and `long`) to be declared as either signed or unsigned. If a Fortran program passes a signed integer to a C function that expects an unsigned integer argument, C will interpret the bit pattern as an unsigned value.

An unsigned integer in C can represent twice the number of positive values as the same-sized integer in HP Fortran. If an HP Fortran program calls a C function that returns an unsigned integer and the return value is greater than can be represented in a signed integer, HP Fortran will interpret the bit pattern as a negative number.

Logicals

C uses integers for logical types. In HP Fortran, a 2-byte `LOGICAL` is equivalent to a C `short`, and a 4-byte `LOGICAL` is equivalent to a `long` or `int`. In C and HP Fortran, zero is false and any nonzero value is true. HP Fortran sets the value `1` for true.

Complex numbers

C has no complex numbers, but they are easy to simulate. To illustrate this, create a `struct` type containing two floating-point members of the correct size — two `float`s for the complex type, and two `double`s for the double complex type. The following creates the `typedef` `COMPLEX`:

```
typedef struct
{
    float real;
    float imag;
} COMPLEX;
```

Consider a program that consists of two source files:

- The Fortran source file, which defines the main program unit
- The C source file, which defines a function `sqr_complex`, having the following prototype declaration:

```
COMPLEX sqr_complex(COMPLEX cmx_val);
```

The main subprogram calls `sqr_complex`, passing in a complex number. The C function squares the number and returns the result. There is no complex data type in C, but this example uses C's `typedef` feature to create one.

The Fortran source file for such a scenario is shown below in the example `pass_complex.f90`.

pass_complex.f90

```
PROGRAM main
! This program passes a complex number to a C function
!   that squares it and returns the result.  The C
!   function has the following declaration prototype:
!
!   complex sqr_complex(complex cmx_val);
!
!   "complex" is not an intrinsic type for C but it
!   creates a typedef for one, using a struct.

COMPLEX :: result, cmx_num = (2.5, 3.5)

! We have to declare the C function because we're calling it
!   as a function rather than a subroutine.  If we didn't
!   declare it, Fortran would use the implicit typing rules
!   by default and assume from the name, sqr_complex, that it
!   returns a real.
COMPLEX sqr_complex

PRINT *, 'C will square this complex number: ', cmx_num
```

```
! Use the %VAL built-in function to indicate that cmx_num
!   is being passed by value, as C expects it to be, and
!   and not by reference, as Fortran does by default
result = sqr_complex(%VAL(cmx_num))

PRINT *, 'The squared result is:             ', result

END PROGRAM main
```

The following is the C source file.

sqr_complex.c

```
#include <stdio.h>

/* simulate Fortran's complex number */
typedef struct
{

        float real;
        float imag;
}COMPLEX;

/* returns the square of the complex argument */
COMPLEX sqr_complex(COMPLEX cmx_val)
{
    COMPLEX result;
    float a, b;

    /* copy both parts of the complex number into locals */
    a = cmx_val.real;
    b = cmx_val.imag;

    /* square the complex number and store the results into
     * the return variable
     */
    result.imag = 2 * (a * b);
    a = a * a;
    b = b * b;
    result.real = a - b;

    return result;
}
```

Below are the command lines to compile, link, and execute the program, followed by the output from a sample run.

```
$ cc -Aa -c sqr_complex.c
$ f90 pass_complex.f90 sqr_complex.o
$ a.out
 C will square this complex number:  (2.5,3.5)
 The squared result is:              (-6.0,17.5)
```

Derived types

Although the syntax of Fortran's derived types differs from that of C's structures, both languages have similar default packing and alignment rules. HP Fortran uses the same packing rules and alignments when laying out derived-type objects in memory that HP C uses for structures.

Pointers

Although the Fortran pointer differs in some respects from the C pointer, a pointer passed by Fortran to a C function looks and acts the same as it does in C. The only precaution is that, when the pointer is to an array (which will almost always be the case), the two languages store and access arrays differently; see “Arrays” on page 173.

Allocatable arrays may be passed from Fortran to C like any other array, with the precaution about array differences between the two languages. Strings (an array of characters in C) are a different matter; see “C strings” on page 177 for information about passing strings from Fortran to C.

Argument-passing conventions

The important difference between the argument-passing conventions of HP C and HP Fortran is that Fortran passes arguments by reference — that is, it passes the address of the argument — whereas C passes non-array and non-pointer arguments by value — that is, it passes a copy of the argument. This difference affects calls not only to user-written routines in C but also to all HP-UX system calls and subroutines, which are accessed as C functions.

HP Fortran provides two built-in functions, `%VAL` and `%REF`, to override Fortran's default argument-passing conventions to conform to C. These functions are applied to the actual arguments you want to pass, either in the argument list of the routine you are calling or with the `$HP$ ALIAS` directive. The `%REF` function tells Fortran that its argument is to be passed by reference (as when passing an array or pointer in C), and the `%VAL` function tells Fortran that its argument is to be passed by value (the default case in C).

Consider a C function having the following prototype declaration:

```
void foo(int *ptr, int iarray[100], int i);
```

In Fortran, the actual arguments to be passed to `foo` would be declared as follows:

```
INTEGER :: ptr, i
INTEGER, DIMENSION(100) :: iarray
```

The call from Fortran to the C function (using the `%VAL` and `%REF` built-in functions) would be as follows:

```
CALL foo(%REF(ptr), %REF(iarray), %VAL(i))
```

If the Fortran program were to make numerous calls to `foo` at different call sites, you might find it more convenient to use the `$HP$ ALIAS` directive with the `%VAL` and `%REF` built-in functions. Using the `$HP$ ALIAS` directive allows you to establish the argument-passing modes for each parameter in a particular routine once and for all, without having to use `%VAL` and `%REF` at each call site. Here is the `$HP$ ALIAS` directive for the Fortran program that calls `foo`:

```
!$HP$ ALIAS foo(%REF, %REF, %VAL)
```

Note that the functions are used here without arguments; their positions in the argument list indicate the parameters to which each applies.

You can also use the `$HP$ ALIAS` directive to handle case-sensitivity difference between C and HP Fortran; "Case sensitivity" on page 170, which includes an example program that uses the `$HP$ ALIAS` directive and the `%VAL` and `%REF` built-in functions to call a C function. For other examples, see "Complex numbers" on page 165 and "File handling" on page 181. Note that the example Fortran program in "Arrays" on page 173 does not require the built-in functions because both Fortran and C pass arrays by reference.

For detailed information about the `$HP$ ALIAS` directive and the `%VAL` and `%REF` built-in functions, see the *HP Fortran Programmer's Reference*.

Case sensitivity

Unlike HP Fortran, C is a case-sensitive language. HP Fortran converts all external names to lowercase, and it disregards the case of internal names. Thus, for example, the names `foo` and `FOO` are the same in Fortran. C, however, is a case-sensitive language: `foo` and `FOO` are different in C. If an HP Fortran program is linked to a C object file and references a C function that uses uppercase characters in its name, the linker will not be able to resolve the reference.

If case sensitivity is an issue when calling a C function from an HP Fortran program, you have two choices:

- Compile the Fortran program with the `+uppercase` option, which forces Fortran to use uppercase for external names.
- Use the `$HP$ ALIAS` directive to specify the case that Fortran should use when calling an external name.

It is unusual that all names in the C source file would be uppercase, which would be the only case justifying the use of the `+uppercase` option. Therefore, we recommend using the `$HP$ ALIAS` directive. This directive enables you to associate an external name with an external name, even if the external name uses uppercase characters.

The `$HP$ ALIAS` directive also has the advantage that you can use it with the `%REF` and `%VAL` built-in functions to specify how the arguments are to be passed without having to repeat them at every call site.

Consider the following C source file, which contains a function to sort an array of integers:

sort_em.c

```
#include <stdio.h>

void BubbleSort(int a[], int size)
{
    int i, j, temp;

    for (i = 0; i < size - 1; i++)
        for (j = i + 1; j < size; j++)
            if (a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}
```

Before a Fortran program can call this function correctly, it must resolve two issues:

1 The name of the C function contains both uppercase and lowercase letters.

2 The function expects its second argument (the size of the array) to be passed by value.

The following `$HP$ ALIAS` directive handles both issues:

```
!$HP$ ALIAS bubblesort = 'BubbleSort'(%REF, %VAL)
```

The name `bubblesort` is the alias that Fortran will use to refer to the C function, and the `%REF` and `%VAL` built-in functions change Fortran's argument-passing conventions to conform to how the C function expects the arguments to be passed.

The following is an HP Fortran program that uses the `$HP$ ALIAS` directive to call the C function correctly.

test_sort.f90

```
PROGRAM main
! This program is linked with an object file that contains
!   a C function with the following prototype declaration:
!
!       void BubbleSort(int a[], int size);
!
!   The ALIAS directive takes care of the differences
!   between C and Fortran regarding case sensitivity
!   and argument-passing conventions.
```

```
!$HP$ ALIAS bubblesort = 'BubbleSort'(%REF, %VAL)
INTEGER, PARAMETER :: n = 10
INTEGER, DIMENSION(n) :: num=(/5,4,7,8,1,0,9,3,2,6/)

PRINT *, 'Before sorting: ', num
CALL bubblesort(num, n)
PRINT *, 'After sorting:  ', num

END PROGRAM main
```

Here are the command lines to compile, link, and execute the program, followed by the output from a sample run:

```
$ cc -Aa -c sort_em.c
$ f90 test_sort.f90 sort_em.o
$ a.out
 Before sorting:  5 4 7 8 1 0 9 3 2 6
 After sorting:   0 1 2 3 4 5 6 7 8 9
```

If you use the `$HP$ ALIAS` directive in many of the Fortran source files in your program, you may find it convenient to define all of the directives in one file and include that file in all of the Fortran source files with the `+pre_include=`*file* option. This option takes one argument, *file*, which is the name of the file you want to include. All text in *file* is prepended to each of the source files specified on the command line, before being passed to the compiler.

See “File handling” on page 181 for another example of a program that uses the `$HP$ ALIAS` directive. The *HP Fortran Programmer's Reference* fully describes the `%VAL` and `%REF` built-in functions, the `+uppercase` and `+pre_include` options. Th e `$HP$ ALIAS` directive is discussed in “HP ALIAS” on page 190.

Arrays

There are two differences between HP Fortran and C to consider when passing arrays from Fortran to C:

- In HP Fortran, array subscripts start by default at 1, whereas in C they always start at 0
- In HP Fortran, multi-dimensional arrays are laid out differently in memory than they are in C.

The difference in subscript-numbering does not result in any size discrepancies: an array of 10 elements in Fortran has 10 elements in C, too. But the subscripts in Fortran will be numbered `1 - 10`, whereas in C they will be numbered `0 - 9`. This difference should not require any change to the normal coding practice for C or for Fortran.

The difference in the way multi-dimensional arrays are laid out is well-known but more significant: Fortran lays out multi-dimensional arrays in **column-major order**, so that the *leftmost* dimension varies fastest; whereas C lays out multi-dimensional arrays in **row-major order**, so that the *rightmost* dimension varies fastest.

Figure 3 shows the Fortran and C declarations for a two-dimensional array of integers, each having the same number of rows and columns. The boxes under each array declaration represents the memory locations where each element of the array is stored. As shown, each language represents the six elements in a different order: the value stored at the first row and second column is not the same for Fortran as for C.

Figure 3 Memory layout of a two-dimensional array in Fortran and C

```
INTEGER, DIMENSION(2,3) :: a
```

a(1,1)	a(2,1)	a(1,2)	a(2,2)	a(1,3)	a(2,3)

```
int a[2][3];
```

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

To compensate for this difference, the dimensions of the array in either the C or Fortran code should be declared in the reverse order of the other. For example, if the array is declared in Fortran as follows:

```
INTEGER, DIMENSION(3,6) :: my_array
```

then the array should be declared in C as follows:

```
int my_array[6][3];
```

You can change the array declaration in either language, whichever is more convenient. The important point is that, to be conformable, the dimensions must be in reverse order.

Below is an example for a three-dimensional array, the first being for a Fortran declaration.

```
REAL, DIMENSION(2,3,4) :: x
```

Below is the same declaration as declared in C.

```
int x[4][3][2];
```

pass_array.f90

```
   PROGRAM main
   ! This program initializes a multi-dimensional array,
   !   displays its contents, then passes it to a C function,
   !   which displays its contents.  The C function has the
   !   following declaration prototype:
   !
   !   void get_array(int a[4][2]);
   !
   !   Note that the dimensions are declared in reverse order
   !   in C from the way they are declared in Fortran.
   INTEGER, DIMENSION(2,4) :: my_array = &
       RESHAPE(SOURCE = (/1,2,3,4,5,6,7,8/), SHAPE = (/2,4/))

   PRINT *, 'Here is how Fortran stores the array:'
   DO i = 1, 4
       DO j = 1, 2
           PRINT 10, j, i, my_array(j,i)
       END DO
   END DO

   ! There's no need to use the %VAL or %REF built-in functions
   !   because both C and Fortran pass arrays by reference.
   CALL get_array(my_array)
10 FORMAT('my_array(', I1, ',', I1, ') =', I2)
   END PROGRAM main
```

Below is the source file for a HP Fortran program that calls a C function, passing a two-dimensional array of integers.

The following is the source file for the C function.

get_array.c

```
#include <stdio.h>
/* get_array:  displays the contents of the array argument */
void get_array(int a[4][2])
{
    int i, j;

    printf("\nHere is the same array as accessed from C:\n\n");
    for (i = 0; i < 4; i++)
        for (j = 0; j < 2; j++)
            printf("a[%d][%d] = %d\n", i, j, a[i][j]);
}
```

Here are the command lines to compile, link, and execute the program, followed by the output from a sample run:

```
$ cc -Aa -c get_array.c
$ f90 pass_array.f90 get_array.o
$ a.out
 Here is how Fortran stores the array:
my_array(1,1) = 1
my_array(2,1) = 2
my_array(1,2) = 3
my_array(2,2) = 4
my_array(1,3) = 5
my_array(2,3) = 6
my_array(1,4) = 7
my_array(2,4) = 8

Here is the same array as accessed from C:

a[0][0] = 1
a[0][1] = 2
a[1][0] = 3
a[1][1] = 4
a[2][0] = 5
a[2][1] = 6
a[3][0] = 7
a[3][1] = 8
```

In this example, it is assumed that the C routine has the array size information already coded into it. If that is not the case, then the Fortran program must also pass the size as a separate argument, and the C routine must be changed to accept a second argument.

For an example of a Fortran program that passes an array and its size as arguments to a C function, see “Case sensitivity” on page 170. For an example of a Fortran program that passes character array arguments to C, see “Passing a string” on page 178.

C strings

C strings differ from Fortran character variables in two important respects:

- C expects strings to be **null**-terminated.
- For each character variable or character constant that Fortran passes to a C routine, it also passes a hidden length argument.

The following sections discuss these differences and explain how to code for them. The last section includes an example program.

C null-terminated string

Unlike HP Fortran programs written in C expect strings to be null-terminated; that is, the last character of a string must be the null character (`'\0'`). To pass a string from Fortran to C, you must do the following:

- Declare the character variable that is large enough to include the null character.
- Explicitly assign the null character to the final element of the character array or use the concatenation operator, as in the following example:

```
CALL csub ('a string'//CHAR(0))
```

If the Fortran program is going to use a string that has been passed back to it from C, then either the C function or the Fortran subprogram should strip off the null character before Fortran tries to use it. The example program in “Passing a string” on page 178 shows how to do this in C.

Fortran hidden length argument

For each `CHARACTER*`*n* argument passed to a Fortran subprogram, two items are actually passed as arguments:

- The address of the character argument in memory (that is, a pointer to the argument).
- The argument's length in bytes. This is the “hidden” length argument that is available to the subprogram from the stack.

To pass a string argument from Fortran to C, you must explicitly prepare the C function to receive the string address argument and the hidden argument. The order of the address arguments in the argument list will be the same in C as in Fortran. The hidden length arguments, however, will come at the end of the list. If more than one string argument is passed, the length arguments will follow the same order as the address arguments, but at the end of the C’s argument list.

Note that both C and Fortran both pass strings by reference. This means that, if Fortran passes only string arguments to C, you need not use the `%VAL` and `%REF` built-in functions to indicate how the arguments are to be passed. For information about these functions, see “Argument-passing conventions” on page 168.

Passing a string

The example program in this section illustrates how to pass a string—which, in Fortran, is a character variable or constant—to a C function. It also illustrates how to process a C string so that it can be manipulated in Fortran.

The program consists of two source files:

- The Fortran source file, which consists of a main program unit that declares two initialized character variables and passes them to a C function.
- The C source code, which consists of two functions:
  - `get_string`: receives the two character array arguments from Fortran and overwrites the strings in the arrays with new strings

– `fix_string_for_f90`: processes the string in its character array argument to replace the null-terminating character with a blank character and to blank-fill the remaining characters. This processing is necessary so that Fortran can manipulate the character variable.

The `get_string` function has two additional arguments in its argument list, which pick up the hidden string length arguments that Fortran implicitly passes with each string argument.

The following are example C and Fortran programs.

pass_chars.f90

```
   PROGRAM main
   ! This program passes to character variables to a C routine,
   !   which overwrites them.  This program displays the
   !   character variables before and after the call.

   ! Initialize the character variables and append null
   !   characters so that C can process them.
   CHARACTER(LEN=10) :: first_name = "Pete"//CHAR(0)
   CHARACTER(LEN=15) :: last_name = "Seeger"//CHAR(0)

   ! Note that character variables, like arrays, are passed by
   !   reference in both languages.  There's no need to use the
   !   %REF built-in function, so long as the C routine
   !   provides an extra argument for the "hidden" length
   !   parameter.  To suppress passing that parameter, use %REF.
   CALL get_string(first_name, last_name)

   PRINT 20, first_name, last_name

20 FORMAT(/, 'The names passed back to Fortran:  ', A, 1X, A)

   END PROGRAM main
```

get_string.c

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


void fix_string_for_f90(char s[], int len);

/* get_string:  overwrites the string arguments fname and lname;
 * fname_len and lname_len are the hidden length arguments, which
 * are implicitly passed by Fortran with each string argument.
 */
void get_string(char fname[], char lname[], int fname_len,
                int lname_len)
{
  printf("The names passed to C:  %s %s\n", fname, lname);
  printf("\nEnter the first and last names of a banjo player:
```

```
");
  scanf("%s%s", fname, lname);

  fix_string_for_f90(fname, fname_len);
  fix_string_for_f90(lname, lname_len);
}

/* fix_string_for_f90:  replaces the null at the end of the
string
 * in the character array and th a blank and blank fills the
 * remaining elements up to len;  this processing is necessary if
 * the character variable is to be manipulated by Fortran
 */
void fix_string_for_f90(char s[], int len)
{
  int i;

  for (i = strlen(s); i < len; i++)
    s[i] = ' ';
}
```

Below are the command lines to compile, link, and execute the program, followed by the output from a sample run.

```
$ cc -Aa -c get_string.c
$ f90 pass_chars.f90 get_string.o
$ a.out
The names passed to C:  Pete Seeger

Enter the first and last names of a banjo player:  Wade Ward

The names passed back to Fortran:  Wade  Ward
```

File handling

A Fortran unit number cannot be passed to a C routine to perform I/O on the associated file; nor can a C file pointer be used by a Fortran routine. However, a file created by a program written in either language can be used by a program in the other language if the file is declared and opened within the program that uses it.

C accesses files using HP-UX I/O subroutines and intrinsics. This method of file access can also be used from Fortran instead of Fortran I/O.

You can pass file units and file pointers from Fortran to C with the `FNUM` and `FSTREAM` intrinsics. `FNUM` returns the HP-UX file descriptor corresponding to a Fortran unit, which must be supplied as an argument; see “Establishing a connection to a file” on page 160 for information about file descriptors. `FSTREAM` returns C's file pointer for a Fortran unit number, which must also be supplied as an argument.

The following Fortran program calls the `write` system routine to perform I/O on a file, passing in a file descriptor returned by `FNUM`. (Because of the name conflict between the `write` system routine and the Fortran `WRITE` statement, the program uses the `ALIAS` directive to avoid the conflict by referring to `write` as `IWRITE`.)

fnum_test.f90

```
  PROGRAM fnum_test

! Use the ALIAS directive to rename the "write" system routine.
! The built-in functions %VAL and %REF indicate how the
! arguments are to be passed.

!$HP$ ALIAS IWRITE = 'write' (%VAL, %REF, %VAL)

  CHARACTER*1 :: a(10)
  INTEGER :: i, fd, status

! fill the array with x's
  a = 'x'

! open the file for writing
  OPEN(1, FILE='file1', STATUS='UNKNOWN')

! pass in the unit number and get back a file descriptor
  fd = FNUM(1)

! call IWRITE (the alias for the "write" system routine),
```

```
! passing in three arguments:
!     fd = the file descriptor returned by FNUM
!     a  = the character array to write
!     10 = the number of elements (bytes) to write
! the return value, status, is the number of bytes actually
! written; if the write was successful, it should be 10
  status=IWRITE(fd, a, 10)

  CLOSE (1, STATUS='KEEP')

! open the file for reading; we want to see if the write was
! successful
  OPEN (1, FILE='file1', STATUS='UNKNOWN')

  READ (1, 4) (a(i), i = 1, 10)
4 FORMAT (10A1)
  CLOSE (1, STATUS='DELETE')

  DO i = 1, 10
    ! if we find anything other than x's, the write failed
    IF (a(i) .NE. 'x') STOP 'FNUM_TEST failed'
  END DO

! check write's return value; it should be 10
  IF (status .EQ. 10) PRINT *, 'FNUM_TEST passed'

  END
```

Below are the command lines to compile, link, and execute the program, followed by the output from a sample run.

```
$ f90 fnum_test.f90
$ a.out
 FNUM_TEST passed
```

The *HP Fortran Programmer's Reference* describes the `FNUM` and `FNUM` intrinsics and the `ALIAS` directive. For information about the `write` system routine, see the *write*(2) man page.

Sharing data

Fortran programmers have traditionally relied on the common block to share large amounts of data among different program units. The convenience offered by the common block is that it can give storage access to program units that don't otherwise communicate with each other, even when they reside in separate files.[1]

Although C has no common blocks, it does provide *external variables*, which can also be used to share data among different parts of a C program. A variable becomes external when defined outside any function. To become accessible to a function, the external variable must be declared without being defined within the function that wants to access it. (In C, a variable is *defined* when storage is allocated for it, and *declared* when its name and type are stated without any storage allocation.) To declare a variable in C without defining it, you use the `extern` storage class specifier, which tells the linker to look elsewhere for the definition.

For example, the following statement (assuming that it is made outside any function) declares and defines the external variable `some_data`:

```
int some_data;
```

The next statement declares `some_data` without defining it, making it available to the function in which the declaration is made:

```
extern int some_data;
```

Fortran's common block and C's `extern` statement can work together to enable Fortran program units to share data with an HP C function. The storage is actually allocated (or in C terminology, *defined*) in the Fortran source file. The C source file declares but does not define the name of the common block, using the `extern` specifier. The linker resolves the reference at linktime.

1. However, overreliance on common blocks can make programs difficult to maintain. For a discussion of the advantages of the Fortran module over the common block, refer to Chapter 3, "Controlling data storage," on page 89.

Consider the following Fortran statements, which declare an array of integers and place the array in a common block named `globals`:

```
INTEGER, DIMENSION(100) :: global_array
COMMON /globals/global_array
```

The next statement is the `extern` statement that references (in C terminology, *declares*) the common block, making it available to a function in the C object file:

```
extern int globals[100];
```

Note that the `extern` specifier references the name of the common block, `globals`, not the name of the array. From C's point of view, the common block is treated as though it were the array.

The common block to be shared with a C function can contain more than one data item. To do so, the C source file must declare a structure whose members match the data items in common. Any C function needing access to an item in common uses the `extern` statement to declare a variable of the structure type. The name of the variable is that of the common block. To access an individual data item, the function uses the C notation for referencing members of a structure.

HP Fortran uses the same packing and alignment rules when laying out common blocks in memory that HP C uses for structures. However, the programmer must be sure to declare the number, types, and sizes of the structure members in the same order as they appear in the common block. Refer to Table 28 on page 162 for the data type correspondences for both languages.

The following example program consists of two source files that contain the Fortran main program unit and a C function called from Fortran. The main program unit specifies a common block having two double-precision variables. It writes to one of the variables and calls the C function. The C function reads the variable written by Fortran and writes to the other one. After the call returns, Fortran reads both variables.

The following are examples of Fortran and C source files.

shared_common.f90

```
   PROGRAM main
   ! This program uses the common block to share data with
   !   the C function get_nlog.  C uses a structure type to
   !   declare the same items in common.

   REAL(KIND=8) :: num, nlog_of_num
   COMMON /globals/num, nlog_of_num

   ! a header for the table that is printed by the following
   !   DO loop
   PRINT *, 'Number    Natural Log of Number'
   PRINT *, '-------+----------------------'

   ! At each iteration, write a value to the common block
   !   variable num, call the C function get_nlog, and
   !   print the contents of both common block variables
   !   to the screen.
   DO num = 2.0, 10.0
       CALL get_nlog()
       PRINT 10, num, '|', nlog_of_num
   END DO

10 FORMAT(3X, F3.0, 2X, A, 8X, F5.2)

   END PROGRAM main
```

shared_struct.c

```
#include <stdio.h>
#include <math.h>

/* declare a structure whose members match the data items
 * in the Fortran common block
 */
struct glob
{
    double num;
    double nlog_of_num;
} globals;

/* get_nlog:  reads the value in globals.num, passes it
 * to log() in the math library, and writes the write the
 * return value to globals.nlog_of_num
 */
void get_nlog(void)
{
    /* declare the name of the common block defined in the
     * Fortran file
     */
    extern struct glob globals;
    globals.nlog_of_num = log(globals.num);
}
```

Below are the command lines to compile, link, and execute the program, followed by the output from a sample run. The `-lm` option at the end of second command line tells the linker to look in the math library for the `log` function:

```
$ cc -Aa -c shared_struct.c
$ f90 shared_common.f90 shared_struct.o -lm
$ a.out
 Number    Natural Log of Number
 -------+----------------------
    2.  |          0.69
    3.  |          1.10
    4.  |          1.39
    5.  |          1.61
    6.  |          1.79
    7.  |          1.95
    8.  |          2.08
    9.  |          2.20
   10.  |          2.30
```

See the *HP Fortran Programmer's Reference* for a full description of the `COMMON` statement.

9 Using Fortran directives

Compiler directives are commands within the source program that affect how the program is compiled. They are similar in function to command-line options, but generally provide more local control. The directives provided by HP Fortran use a syntax that causes them to be treated as comments (and so ignored) when ported to another processor or when incorrectly formatted. The following sections describe the HP Fortran directives.

HP Fortran also recognizes C Preprocessor (cpp) directives. If you compile with the `+cpp=yes` option or if the name of the source ends in the `.F` extension, the source files are first passed to the C preprocessor for processing. For information about the C preprocessor, refer to *cpp*(1)..

Directive syntax

The syntax for specifying directives in HP Fortran source files varies according to the type of directive:

C preprocessor directives take the form:

`#[line]`*cpp-directive*

where *cpp-directive* is ANSI C-conforming except that the `line` keyword is optional, making it compatible with the HP C compiler.

HP Fortran compiler directives take the form:

comment-character `$HP$` *directive-name*

where *comment-character* is `!` in free-source format or `C`, `!`, or `*` in fixed-source format; and *directive-name* is one of the directives described in this chapter.

There must be no space between *comment-character* and `$HP$`. In fixed-source format, *comment-character* must be in column 1.

Using HP Fortran directives

HP Fortran provides a number of compiler directives that are useful for controlling certain functions (for example, optimization) within the source file. Table 31 lists and briefly describes these directives; they are listed in the order in which they appear in the sections below.

Table 31 HP Fortran directives

Directive	Function
`$HP$ ALIAS`	Associates the name of a subroutine, function, entry, or common block with an external name.
`$HP$ CHECK_OVERFLOW`	Generates code to trap integer overflows.
`$HP$ LIST`	Controls output of source lines in listing file.
`$HP$ OPTIMIZE`	Controls optimization within the source file.

In files that use free format, directives must start with the comment character `!`. In fixed format, they must start with the comment character `C`, `*`, or `!` in column 1. Keywords and any arguments must be delimited by at least one space character, as in the following:

```
!$HP$ OPTIMIZE ON
```

Using the comment character as the directive prefix ensures that, unless the compiler is specifically looking for the directive, it is otherwise treated as a comment and ignored.

The following sections describe each of the HP Fortran directives.

HP ALIAS

The ALIAS directive associates the name of a subroutine, function, entry, or common block with an external name and specifies the parameter-passing conventions of routines written in other languages.

Syntax

```
!$HP$ ALIAS name [= external-name ] [(arg-pass-mode-list)]
```

name

is the name used by the program to refer to a subroutine, function, or procedure entry point—but not to an internal subroutine. If *name* is enclosed by slashes, it is a common block name.

external-name

is a character constant that specifies a standard symbolic name.

arg-pass-mode-list

is used only when *name* is that of a procedure that takes arguments. The items in the list specify how the corresponding actual argument are to be passed. The items can be either of the following built-in functions:

- %VAL: pass the value of the actual argument
- %REF: pass the address of the actual argument

There must be as many items in the list as there are arguments in the procedure, they must be separated by commas, and they must correspond positionally to the arguments.

Description and restrictions

The HP ALIAS directive serves two purposes:

- It provides a way to associate the name used by your program when referring to a subroutine, function, entry, or common block with a distinct external name. This feature is especially useful when you want to access a variety of different graphics device drivers from the same source code so that different hardware configurations can be supported.
- When used in conjunction with the %VAL and %REF built-in functions, it provides a way to direct the compiler to use the appropriate parameter passing conventions to communicate with routines written in other high-level languages.

external-name should never conflict with the name of an HP-UX system routine (described in sections 2 and 3 of the *HP-UX Reference*) or with a Fortran library routine (for example, `OPEN`, `READ`, or `CLOSE`). The `$HP$ ALIAS` directive applies to subroutines, entries, and functions that are used externally. It does not apply to the main program unit.

`%VAL` is a built-in function that specifies that the value of the actual argument is to be passed to the called procedure. You can use this parameter with all types of arguments. However, when used with a procedure name, it has no effect; a pointer to the procedure is still passed.

`%REF` specifies that the address of the actual argument is to be passed to the called procedure. For non-character arguments, this is the default. For character arguments, `%REF` disables the passing of the hidden length parameter.

When `%VAL` and `%REF` are used with the `CALL` statement, they override the specification in the `$HP$ ALIAS` directive. For detailed information about these built-in functions and their use in the `CALL` statement, see the *HP Fortran Programmer's Reference*..

Note the following restrictions:

- Attempts to redefine `$HP$ ALIAS` names generate warning messages.
- The compiler always uses *external-name* exactly as it is entered. No case transformations occur, and no underscore is appended. The `+ppu` and `+uppercase` command-line options do not apply to external names specified by the `$HP$ ALIAS` directive.

Local and global usage

The `$HP$ ALIAS` directive can be used either locally or globally, as follows:

- The `$HP$ ALIAS` directive has local application only—that is, its effect is limited to a particular program unit—if it appears within the boundaries of that program unit. To have local application only, the directive must appear after any `PROGRAM`, `SUBROUTINE`, or `FUNCTION` statement and before the first occurrence of *name* in the target program unit.
- The `$HP$ ALIAS` directive has global application—that is, it applies to all subsequent program units—if it appears outside and before the boundaries of those program units to which it is to apply.

Examples

The `$HP$ ALIAS` directive is especially useful when calling a routine in a language that uses different conventions than Fortran. The following examples illustrate how to use the `$HP$ ALIAS` directive to resolve differences with:

- Case sensitivity
- Argument-passing conventions
- Strings

Case sensitivity

Names in HP Fortran are not case sensitive; that is, the compiler converts all names to lowercase. This means that if you reference a routine in a language that is case sensitive and the routine name contains uppercase letters, a call to that routine in HP Fortran will result in an unresolved reference—unless you use the `$HP$ ALIAS` directive to redefine the name in all lowercase letters, as in the following example:

```
!$HP$ ALIAS printnames = 'PrintNames'
```

Argument-passing conventions

By default, HP Fortran assumes that all parameters in a subroutine or function call are passed by reference; that is, the call passes the addresses of the parameters, not their values. On the other hand, C code assumes that parameters are passed by value; that is, the current value of the actual parameter is passed to the called routine. Without the `$HP$ ALIAS` directive, it would be difficult to call a C routine from a Fortran program.

For example, suppose you want to call the system routine `calloc` (see the *malloc*(3C) man page) to obtain dynamic memory. The man page describes the calling sequence as:

```
char *calloc(unsigned nelem, unsigned elsize);
```

It would be difficult, using standard Fortran constructs, to provide actual parameters corresponding to `nelem` and `elsize` because HP Fortran always passes addresses. The `$HP$ ALIAS` directive can solve this problem by directing the compiler to generate call-by-value actual parameters:

```
!$HP$ ALIAS calloc(%VAL, %VAL)
```

Strings

Programs written in C expect strings to be terminated with the null character (`'\0'`). But HP Fortran programs pass a hidden length parameter to indicate the end of a string argument. Thus, if you want to pass a string from HP Fortran to a C language function, you must explicitly append the null to the string and suppress the hidden length parameter. The `$HP$ ALIAS` directive enables you to pass the string from Fortran to C. For example, consider the following routine:

pr_str.c

```
void c_rout(char *s)
{
  printf("%s\n", s);
}
```

The `ALIAS` directive in the following program enables the string to be passed to `c_rout`:

pass_str.f90

```
PROGRAM main
!$HP$ ALIAS c_rout(%REF)
  CHARACTER(LEN=10) name
  name = 'Charlie'
! Append a null to the string so that C can handle it properly
  CALL c_rout(name//char(0))
END PROGRAM main
```

Here are the command lines to compile and link both files, and to execute the program, along with the output from a sample run:

```
$ cc -Aa -c pr_str.c
$ f90 pass_str.f90 pr_str.o
$ a.out
Charlie
```

For more information For detailed information about the `%REF` and `%VAL` built-in functions, see the *HP Fortran Programmer's Reference*.

HP CHECK_OVERFLOW

The `$HP$ CHECK_OVERFLOW` directive generates code to trap when an overflow occurs in integer arithmetic. By default, integer overflow is ignored.

Syntax

```
!$HP$ CHECK_OVERFLOW INTEGER [ON | OFF]
```

`ON`. causes the compiler to generate code to trap integer overflow exceptions.

`OFF`. causes the compiler not to generate code to trap integer overflow exceptions.

Description and restrictions

If you use `$HP$ CHECK_OVERFLOW` with the `ON` statement, you can cause your program to ignore the overflow, abort on the overflow, or branch to a trap subroutine. If this directive is not used, the `ON` statement has no effect on integer overflow errors.

This directive can appear anywhere in your program. It stays in effect until a subsequent `$HP$ CHECK_OVERFLOW` directive changes the status.

For more information

For more information about the `ON` statement see the *HP Fortran Programmer's Reference*.

HP LIST

The `$HP$ LIST` directive turns on or off the inclusion of subsequent source lines in the listing output.

Syntax

```
!$HP$ LIST [ON | OFF]
```

`ON`. enables the inclusion of source lines in the listing file.

`OFF`. disables the inclusion of source lines in the listing file.

Description and restrictions

The `$HP$ LIST` directive controls which source lines are output to the listing file. This directive is effective only when the source files are compiled with the `+list` option. It may appear anywhere in the source file.

If the `$HP$ LIST OFF` directive occurs in a file that is compiled with the `+list` option, the listing will contain everything in the source file up through the directive. The `$HP$ LIST OFF` directive applies to the rest of the file, or until a `$HP$ LIST ON` directive is encountered.

Example

The `$HP$ LIST` directive is especially useful for disabling the listing of include files, as in the following example:

```
!$HP$ LIST OFF
INCLUDE "/my_stuff/some_generic_declarations.h"
!$HP$ LIST ON
```

For more information

See "Incompatibilities with HP FORTRAN 77" on page 202 for information about the `+list` option.

HP OPTIMIZE

The `$HP$ OPTIMIZE` directive enables or disables the level of optimization that was specified on the command line.

Syntax

```
!$HP$ OPTIMIZE [ON | OFF]
```

`ON`. enables the level of optimization specified on the command line.

`OFF`. disables the level of optimization specified on the command line.

This directive is effective for all program units that follow it in your program. It should therefore be placed outside and before the program units it is to affect. If you insert this directive inside a program unit, it will have no effect on that program unit, only on those that follow.

Description and restrictions

The `$HP$ OPTIMIZE` directive allows you to determine which areas of your program that the optimizer will process. Specifying `$HP$ OPTIMIZE OFF` causes the following source lines not to be optimized. `$HP$ OPTIMIZE ON` re-enables optimization for the following source lines.

This directive is effective only if you have used either the `-O`*n* or `+O`*n* option when you compiled the program. If you have not specified either option, both `$HP$ OPTIMIZE ON` and `$HP$ OPTIMIZE OFF` will give you level 0 optimization.

For more information

For information about the `-O`*n* and `+O`*n* options `$HP$ OPTIMIZE` directive is also discussed in the *HP Fortran Programmer's Guide*.

Compatibility directives

HP Fortran supports the compiler directives listed in Table 32. These directives are provided for compatibility with programs developed on the platforms listed in the table.

Table 32 Compatibility directives recognized by HP Fortran

Vendor	Directive
Cray	`DIR$ NO SIDE EFFECTS`
	`DIR$ [NO]CONCUR`
	`DIR$ IVDEP`
	`FPP$ NODEPCHK`
KAP	`*$* [NO]CONCURRENTIZE`
	`*$* [NO]VECTORIZE`
VAST	`VD$ NODEPCHK`

In fixed format, each directive must be preceded by the comment character `C`, `!`, or `*` and must begin in column 1 of the source file. In free format, the directive must be preceded by the Fortran comment character (`!`).

If an option or argument is included with the directive name, the compiler ignores the directive.

The following sections describes these directives in detail.

Controlling vectorization

HP Fortran can vectorize eligible program loops that operate on vectors. This optimization causes the compiler to replace the loops with calls to selected routines in the Basic Linear Algebra Subroutine (BLAS) library. You can use the `*$* [NO]VECTORIZE` directive to enable or disable vectorization. The compiler considers the `*$* VECTORIZE` directive as a request to vectorize a loop. If the compiler determines that it cannot profitably or safely vectorize the loop, it ignores the directive.

To use the vectorization directive, you must compile and link with the `+Ovectorize` option. The directive applies to the beginning of the next loop and remains in effect for the rest of the program unit or until superseded by a later directive. For more information about this option, see the *Parallel Programming Guide for HP-UX Systems*.

Controlling parallelization

HP Fortran can parallelize eligible program loops by distributing different iterations of the loop to different processors for parallel execution on a multiprocessor machine. The following directives provide local control over parallelization:

- `*$* [NO]CONCURRENTIZE`
- `DIR$ [NO]CONCUR`

These directives have both enable and disable versions:
`*$* CONCURRENTIZE` and `DIR$ CONCUR` enable parallelization;
`*$* NOCONCURRENTIZE` and `DIR$ NOCONCUR` disable parallelization.

The parallelization directives are effective only if you have compiled and linked the program with the `+Oparallel` and the `+O3` option. Each directive applies to the beginning of the next loop and remains in effect for the rest of the program unit or until superseded by a later directive.

The compiler considers the `*$* CONCURRENTIZE` and `DIR$ CONCUR` directives as requests to parallelize a loop. If the compiler cannot profitably or safely parallelize the loop, it ignores the directive. For information about conditions that can inhibit parallelization, see the *Parallel Programming Guide for HP-UX Systems*.

Controlling dependence checks

The compiler will not parallelize a loop where it detects a possible data dependence, even if you use an option or directive that specifically requests parallelization. However, if you know that there is no actual data dependence in the loop in question, you can insert one of the following directives just before the loop:

- `DIR$ IVDEP`
- `FPP$ NODEPCHK`
- `VD$ NODEPCHK`

The effect of these directives is to cause the compiler to ignore data dependences within the next loop when determining whether to parallelize. The `DIR$ IVDEP` directive differs from the other two in that it causes the compiler to ignore only array-based dependences, but not scalar-based. All three directives apply to the next loop only.

Using these directives to incorrectly assert that a loop has no data dependences can result in the loop producing wrong answers.

Other conditions may limit the compiler's efforts to parallelize, such as the presence of the `VD$ NOCONCUR` directive. Such conditions may prevent parallelization even if you use a directive to disable dependence checking.

Controlling checks for side effects

The compiler will not parallelize a loop with an embedded call to a routine if the compiler finds that the routine has side effects. However, if you know that a routine that is called inside of a loop does not have side effects, you can insert the `DIR$ NO SIDE EFFECTS` directive in front of the loop to force the compiler to ignore any side effects in the referenced routine when it determines whether to parallelize the loop.

This directive affects only the immediately following loop.

NOTE Using this directive to incorrectly assert that a routine has no side effects can result in wrong answers when a call to the routine is embedded in a loop.

Cray's implementation of this directive requires that it precede any executable statement or statement function. HP Fortran does not enforce this requirement.

10 Migrating to HP Fortran

A major feature of HP Fortran is its compatibility with standard-conforming HP FORTRAN 77. Both source files and object files from existing HP FORTRAN 77 applications can be **migrated** to HP Fortran with comparatively little effort. However, some command-line options and nonstandard extensions in HP FORTRAN 77 programs may have to be changed to compile and execute correctly under HP Fortran.

To smooth the migration path, HP Fortran includes a number of extensions that are compatible with HP FORTRAN 77. HP Fortran also includes extensions that are designed to ease the job of **porting** applications from other vendors' Fortran dialects. For a summary list of all HP Fortran extensions, see the *HP Fortran Programmer's Reference*. For information about porting other vendors' Fortran programs to HP Fortran, see "Porting to HP Fortran" on page 219.

This chapter discusses the following topics:

- Incompatibilities with HP FORTRAN 77
- Migration issues
- Approaches to migration

Incompatibilities with HP FORTRAN 77

The following sections describe known incompatibilities between HP Fortran and HP FORTRAN 77. These incompatibilities include both source-level and object-code incompatibilities. A subset of these are detected by the HP `fid` tool, which is described in "Fortran incompatibilities detector" on page 216.

Command-line options not supported

The HP Fortran compiler does not accept the `f77` command-line options listed in Table 33, and the `f77` options listed in Table 34 have been renamed for `f90`. In addition, HP Fortran code may not link correctly with HP FORTRAN 77 object files that were compiled with these options; see "Object code issues" on page 213.

Table 33 f77 options not supported by f90

+800	+e	+N
+A	+I[2|4]	+R
+A3	+L8	+U
+A8	+LA	-w66
+apollo	-lisam	
+E	+mr	

Table 34 f77 options replaced by f90 options

f77 option	f90 replacement
-A	+langlvl[a]
-a	+langlvl[a]
+autodblpad	+autodbl[a]
+B	+escape
-D	+dlines

f77 option	f90 replacement
+es	+extend_source
-F	+cpp_keep
-L	+list
-onetrip	+onetrip
+Q	+pre_include
+s	+langlvl[a]
+T	+fp_exception
+ttyunbuf	+nottybuf
-U	+uppercase
-u	+implicit_none
-V	+list[a]

a. Does not fully replace.

Floating-point constants

The HP Fortran compiler differs from HP FORTRAN 77 in its handling of floating-point constants. The HP Fortran compiler conforms to the standard: a single-precision constant is treated as a single-precision data item in all situations, regardless of how many digits were supplied when specifying it. HP FORTRAN 77 actually scans and saves constants internally in double precision. This behavior can produce slightly different results.

In HP Fortran, the statement

```
DOUBLE PRECISION x = 3.1415926535
```

will initialize `x` to only 32 bits worth of the constant because it interprets the constant as single precision. Under HP Fortran, a constant must have a `D` exponent or a `KIND` suffix to be interpreted as double precision.

In programs that use double precision exclusively, you should consider using the `+real_constant=double` option, which causes real constants to default to double precision. For more information, refer to "Controlling data storage" on page 89.

Intrinsic functions

The Fortran 90 standard has introduced new intrinsics that may collide with function or subroutine names in FORTRAN 77 code. You can resolve such collisions by declaring all procedures that you have written—but especially those that have the same name as nonstandard HP Fortran intrinsics—with the `EXTERNAL` statement. For a list of nonstandard HP Fortran intrinsics, see Table 41 on page 224.

Also, HP FORTRAN 77 allows intrinsics to accept a wider variety of argument types than HP Fortran does. For example, in HP FORTRAN 77 the `MAX` and `MIN` intrinsics can take arguments of different types, while HP Fortran follows the standard and requires all arguments to be of the same type. The HP Fortran version of the `TIME` intrinsic takes a `CHARACTER*` argument; it will not accept an integer. Other intrinsics are similarly affected.

For a full description of all HP Fortran intrinsics, refer to the *HP Fortran Programmer's Reference*.

Procedure calls and definitions

When defining a procedure or making a procedure call, HP Fortran makes the following requirements, which HP FORTRAN 77 overlooks:

- Function references must include the parentheses for the argument list, even when no arguments are supplied. For example, if `foo` is a user-defined function returning `CHARACTER*10`, HP FORTRAN 77 permits `LEN(foo)` and returns 10. HP Fortran requires `LEN(foo())`.
- The argument list must not contain any extraneous commas, which HP FORTRAN 77 allows as "placeholders" for missing arguments. For example, the following is acceptable to `f77` but not `f90`:

  ```
  call foo (a,)
  ```

  To specify optional arguments in HP Fortran, use the `OPTIONAL` statement.

- The `SYSTEM INTRINSIC` directive, by which HP FORTRAN 77 determines interfaces, is not supported by HP Fortran.
- In HP Fortran, recursive procedures must be so declared with the `RECURSIVE` keyword; HP FORTRAN 77 allows recursive procedures by default.

Data types and constants

The following HP FORTRAN 77 extensions for data types and constants are *not supported* by HP Fortran:

- Double precision as the default storage for floating-point constants; see "Floating-point constants" on page 203.
- `I` and `J` integer suffixes. To express the HP FORTRAN 77 constant `10I` (or `I*2`) in HP Fortran, use `10_2`; for `10J` (or `J*4`), use `10_4`.
- Use of the `8#`*n* and `16#`*n* for octal and hex constants, respectively. In HP Fortran, use `O"`*n*`"` for octal constants and `Z"`*n*`"` for hexadecimal constants.
- **BOZ** constants (that is, constants in binary, octal, or hexadecimal format) in `COMPLEX` expressions.
- Non-integer array bounds and character length specifiers.
- Constant expressions that contain the `**` (exponentiation) operator, as in `PARAMETER (RV=1**1.2)`.
- Use of the `PARAMETER` statement without parentheses, as in

  ```
  PARAMETER i = 1
  ```

  In free format, `f90` treats this statement as an error. In fixed format, `f90` treats it as an assignment, identical to:

  ```
  PARAMETERi = 1
  ```

  In HP Fortran, use `PARAMETER (i=1)` instead.
- Use of the `DATA` statement to initialize integers with strings, as in:

  ```
  DATA i /"abcd"/
  ```

- Use of `COMPLEX(16)` temporaries. For example, given the declarations:

  ```
  COMPLEX(KIND=8) :: foo
  REAL(KIND=16) :: bar
  ```

  the expression `foo**bar` is legal in HP FORTRAN 77 but not in HP Fortran. (HP FORTRAN 77 coerces `COMPLEX(16)` entities to `COMPLEX(8)` in order to continue the computation.)

  Given the previous declarations, the following is acceptable in HP Fortran:

  ```
  foo**REAL(bar, 8)   ! foo**bar
  ```

  See the *HP Fortran Programmer's Reference* for information about the `REAL` intrinsic.

Input/output

The following I/O specifiers are recognized by the `OPEN` statement and by other I/O statements in HP FORTRAN 77 but are not supported in HP Fortran:

- `ACCESS=`*expr1*, where *expr1* is a constant expression other than `DIRECT` or `SEQUENTIAL`.
- `IOSTAT=`
- `KEY=`
- `NAME=`
- `READONLY`
- STATUS=*expr2*, where *expr2* is a constant expression other than `OLD`, `NEW`, `UNKNOWN`, `REPLACE`, or `SCRATCH`.
- `TYPE=`

In general, HP FORTRAN 77 allows more specifiers (and more options to specifiers) than does HP Fortran. There are additional differences between the HP FORTRAN 77 version of the `OPEN` statement and the HP Fortran version; compare the description of `OPEN` in the *HP Fortran Programmer's Reference* with that in the *HP FORTRAN/9000 Programmer's Reference*.

In HP FORTRAN 77, namelist-directed output character strings are always quote-delimited; how and whether such strings are delimited in HP Fortran depends on the `DELIM=` specifier. Also, HP FORTRAN 77 allows the `NAMELIST` statement to appear after executable statements; HP Fortran does not. For more information about the `NAMELIST` statement, see the *HP Fortran Programmer's Reference*.

Directives

Only a small number of the compiler directives from HP FORTRAN 77 are supported under HP Fortran. These are:

- `ALIAS`
- `CHECK_OVERFLOW`
- `LIST`
- `OPTIMIZE`
- `SHARED_COMMON`

The syntax and functionality of individual directives has also changed; for detailed information about the HP Fortran directives, see the *HP Fortran Programmer's Reference*.

All unsupported directives should be deleted or replaced by HP Fortran code that results in the same functionality (see Table 35 on page 210).

Miscellaneous

Following are miscellaneous incompatibilities between HP Fortran and HP FORTRAN 77:

- The syntax and functionality of the HP Fortran version of the `ON` statement is different from the HP FORTRAN 77 version. For example, `ON EXTERNAL` and `ON INTERNAL` are not supported in HP Fortran. For information about using the `ON` statement, see the "Using the ON statement" on page 119.
- HP FORTRAN 77 accepts statement functions that convert arguments; HP Fortran does not.
- HP FORTRAN 77 accepts the `{` character as comment syntax; HP Fortran does not.

- HP FORTRAN 77 accepts a `PROGRAM` statement with no name; HP Fortran requires the name.
- HP FORTRAN 77 extends the `PROGRAM` statement to enable access to command-line arguments; HP Fortran does not. For information about how to use intrinsics to access command-line arguments, see "Accessing command-line arguments" on page 156.
- HP FORTRAN 77 supports arrays up to rank 20; HP Fortran supports arrays up to rank 7.
- HP FORTRAN 77 accepts an expression like `+ -A`, but HP Fortran generates a syntax error. Use `+(-A)` instead.
- HP FORTRAN 77 does not print leading zeroes in floating-point numbers; HP Fortran does. This behavior is equivalent to compiling an HP FORTRAN 77 program with the `+E4` option (note that this option is not supported by `f90`).
- In HP FORTRAN 77, integers that overflow (through initialization or constant folding) are replaced with the maximum value for that type. If HP Fortran detects integer overflow, it treats it as an error; if it does not detect it, the overflow value is truncated at runtime.

Migration issues

Migration issues fall into four general categories:

- Source code issues
- Command-line option issues
- Object code issues
- Data file issues

Source code issues

For standard-conforming HP FORTRAN 77 code, migration to HP Fortran can be as simple as recompiling with the `f90` command. The `f90` command accepts source files with the extensions `.f` and `.F` (among others).

However, source code is likely to be the main obstacle on the migration path to HP Fortran. The reason is that HP FORTRAN 77 supports a number of compiler directives and intrinsic functions, some of which are supported by HP Fortran, but others of which are either unsupported or have changed. The following sections discuss how to change directives and intrinsics when migrating HP FORTRAN 77 source code to HP Fortran.

NOTE HP FORTRAN 77 accepts (or forgives) a number of common but nonstandard programming practices that HP Fortran does not. These nonstandard practices as well as all known incompatibilities between HP FORTRAN 77 and HP Fortran are listed in “Incompatibilities with HP FORTRAN 77” on page 202.

Directives

HP FORTRAN 77 supports more than seventy directives; of these, only a handful are supported by HP Fortran; see “Directives” on page 207, for the directives that are supported and for the new directive syntax. Note that, except for the `LIST` directive, the HP Fortran directives have more limited functionality than their HP FORTRAN 77 counterparts; see the *HP Fortran Programmer's Reference*.

Although most of the HP FORTRAN 77 directives are not supported by HP Fortran, some of their functionality is available through command-line options; see Table 35.

Table 35 HP FORTRAN 77 directives supported by f90 options

HP FORTRAN 77 directive	HP Fortran option	Remarks
ANSI	+langlvl=f90	Applies to Fortran 90 instead of FORTRAN 77.
ASSEMBLY	+asm	
AUTODBL DBL	+autodbl[4]	
AUTODBL OFF	+noautodbl	
CONTINUATIONS	not available	Obsolete; the functionality enabled by the directive is now the default.
DEBUG	-g	
IF/ELSE/ENDIF	not available	Use C preprocessor (cpp) directives.
GPROF (ON)	+gprof	
GPROF OFF	+nogprof	
HP_DESTINATION	+DA or +DS	
INCLUDE		Use the Fortran 90 INCLUDE line.
INIT	+Oinitcheck	Option also saves all symbols.
LIST_CODE	+asm	
LONG	+autodbl[4]	Option also affects reals.
LOWERCASE	+[no]uppercase	Lowercase is default.
NLS	+nls	
ONETRIP	+[no]onetrip	
POSTPEND	+[no]ppu	
RANGE (ON)	+check=all or -C	

HP FORTRAN 77 directive	HP Fortran option	Remarks
RANGE OFF	+check=none	
SAVE_LOCALS (ON)	+save	
SAVE_LOCALS OFF	+nosave	
SET	-D or -U	Use the C preprocessor #define directive.
STANDARD_LEVEL ANSI	+langlvl=f90	Applies to Fortran 90 instead of FORTRAN 77.
SYMDEBUG	-g	
UPPERCASE	+[no]uppercase	Lowercase is the default.
WARNINGS	-w	

Intrinsic functions

HP Fortran supports most of the intrinsics that HP FORTRAN 77 offers, and more. In addition, most of these intrinsics are available in HP Fortran without having to activate them with compiler directives or command-line options (as with HP FORTRAN 77).

With the larger number of available intrinsics in HP Fortran, there is the risk of name collisions with user-defined functions in existing HP FORTRAN 77 source code. Use of the EXTERNAL statement can prevent such collisions. Also, many HP FORTRAN 77 intrinsics accept additional (nonstandard) argument types; HP Fortran is more standard-conforming in this regard.

If the program you are migrating from HP FORTRAN 77 to HP Fortran calls libU77 routines in the BSD 3f library, the names of some of those routines may clash with names of HP Fortran intrinsics. Table 36 on page 212 lists the names of libU77 routines and intrinsic procedures that are the same. If your HP FORTRAN 77 program calls any of these libU77 routines, you should declare the routine with the EXTERNAL statement to get the libU77 routine; otherwise, the compiler will attempt to select the corresponding intrinsic procedure. (The f90 option that links in the library of libU77 routines is +U77.

Table 36 Conflicting intrinsics and libU77 routine names

FLUSH	IARGC	SYSTEM
FREE	IDATE	TIME
GETARG	LOC	
GETENV	MALLOC	

Refer to the *HP Fortran Programmer's Reference* for information about all of the HP Fortran intrinsics and the `libU77` routines.

Command-line option issues

Command-line options can become a migration issue in two ways:

- When you compile a program with the HP Fortran compiler, using an `f77` command line. If the command line contains an unsupported `f77` option, `f90` will flag the option with an error message.

  Table 37 lists the `f77` and `f90` that have the same functionality but different names. See Table 33 on page 202 for a list of `f77` options that are not supported by `f90` and Table 34 on page 202 for a list of `f77` options that have been replaced by `f90` options.

- When you execute a program that consists of a mix of object files that have been created by `f77` and `f90`. The problem here is that, although the object files may have been successfully linked, they may not be compatible. If they were incompatible, the resulting executable could behave unexpectedly or produce wrong results. Migration problems caused by incompatible object files are unusual but more difficult to detect and are discussed in the next section.

Table 37 f77 options supported by f90

f77 option	f90 option	function
-C	+check=all	Perform runtime subscript checking
-G	+gprof	Prepare for profiling with gprof
-K	+save	Use static storage for locals instead of stack
-N	+noshared	Mark linker output unshared
-n	+shared	Mark linker output shared
-p	+prof	Prepare for profiling with prof
-Q	+nodemand_load	Do not mark linker output demand load
-q	+demand_load	Mark linker output demand load
-R4	+real_constant=single	Make single precision the default for all single-precision constants
-R8	+real_constant=double	Make double precision the default for all single-precision constants
-S	+asm	Generate assembly listing
-s	+strip	Strip symbol table information from linker output
-Y	+nls	Enable Native Language Support
+Z	+pic=long	Generate position-independent code (large model)
+z	+pic=short	Generate position-independent code (small model)

Object code issues

Some migration problems do not manifest themselves until runtime, when the program behaves unexpectedly or produces incorrect results. Such problems can occur when incompatible HP FORTRAN 77 object files and HP Fortran object files are linked together.

Although the format of object files generated by f77 is compatible with the format of object files generated by f90, individual data items within the f77-generated file may not be. Problems with migration can occur if

the HP FORTRAN 77 object files represent data in a nonstandard form. For example, HP Fortran does not allow misaligned data or nonstandard logical representations, whereas HP FORTRAN 77 does.

Procedure interfaces, on the other hand, usually do not present problems, so long as the procedures are properly defined and called in the HP FORTRAN 77 source code. That is, as long as the definition and call match in argument types, return types, and alternate return capability, the HP Fortran compiler can do the appropriate conversions, copying, etc., to make the calls work.

To resolve object-code incompatibilities, you will need access both to the source file and to the `f77` command line that was used to generate the HP FORTRAN 77 object file. Examine the source file for directives that are not supported by HP Fortran, such as the `$LOGICAL` directive. See "Directives" on page 207 for a list of the directives that are supported. Also, look over the `f77` command line for any of the unsupported options that are listed in Table 33 on page 202.

If you find object-code incompatibilities, you should change the source code and recompile with the `f90` command.

Data file issues

In general, data files are the easiest files to migrate because the data files produced by the two Fortrans are compatible. However, problems can occur because of misaligned data and data types that are not supported under HP Fortran. For example, HP FORTRAN 77 permits misaligned data, especially when working with the structure extension. Also, HP FORTRAN 77 accepts nonstandard representations of logicals. Both examples can result in data files that are incompatible with HP Fortran.

To resolve problems with incompatible data files, examine the source file of the program that generated the data file as well as the command line that was used to compile the source file, following the suggestions discussed in "Object code issues" on page 213.

Approaches to migration

The most direct (and painstaking) approach to migrating an HP FORTRAN 77 program so that it will compile and execute correctly under HP Fortran is to make a clean sweep through the original source code, removing all extensions and rewriting all nonstandard programming practices to conform to the Fortran 90 standard. The result will be a highly portable program.

The disadvantage of the "clean-sweep" approach is that it may require a considerable expense of time and work that may not even be necessary. Many HP FORTRAN 77 extensions are also supported under HP Fortran. The only changes that you *must* make to the source are to remove or re-code the parts of the program that use unsupported or incompatible language extensions.

Although the task of migrating an HP FORTRAN 77 program to HP Fortran can be done manually, there are several utilities that can help to automate the search for incompatibilities. These utilities (including sources of information about migrating to Fortran 90) are described in the following sections.

HP-supplied migration tools

The HP migration tools include the HP FORTRAN 77 and HP Fortran compilers (`f77` and `f90`), `lintfor`, and `fid`.

HP FORTRAN 77 compiler

You can use the `f77` command to test source code for conformance to the FORTRAN 77 standard. The `-A` option causes the compiler to issue warnings when it encounters non-ANSI code.

If you use `f77` for this purpose, the source code must conform to the FORTRAN 77 grammar. In other words, `f77` will flag both HP-specific extensions as well as language features that are unique to Fortran 90. If the source code contains any Fortran 90 features (some of which are allowed in HP FORTRAN 77 but not in standard FORTRAN 77) or if you introduce any Fortran 90 features during the migration process, the `f77` command is no longer useful.

HP Fortran compiler

The `f90` command can be used similarly to the `f77` command to detect incompatibilities in HP FORTRAN 77 source files. The advantage of `f90` over `f77` is that you can use it on code that already contains Fortran 90 features or to which you are incrementally adding such features as part of the migration process.

The main drawback of `f90` as a migration tool is that a clean compilation under `f90` does not guarantee that all incompatibilities have been found; some do not manifest themselves until runtime. Also, linking under `f90` with `f77`-generated object files may yield unexpected behavior or incorrect results; see "Object code issues" on page 213 and "Data file issues" on page 214.

In addition, the `f90` command sometimes reports incompatibilities — especially in syntax—one at a time. Needless to say, fixing incompatibilities one at a time and recompiling after each fix may not be the most cost-effective approach to migrating a large FORTRAN 77 program to HP Fortran.

Lintfor

The `lintfor` tool can be used on HP FORTRAN 77 code to detect semantic assumptions that may not be valid for HP Fortran code. However, `lintfor` does not accept the Fortran 90 grammar and therefore has the same drawbacks as the `f77` command.

Fortran incompatibilities detector

The Fortran Incompatibilities Detector (`fid`) is an HP-supplied tool that was developed specifically to help in migrating HP FORTRAN 77 code to HP Fortran. It is located in:

```
/opt/fortran90/contrib/bin/fid
```

`fid` searches the target source-code file for various HP FORTRAN 77 extensions that are known to be incompatible with HP Fortran. It also detects incompatible command-line options when given an `f77` command line. `fid` reports both source-code and object-code incompatibilities between HP FORTRAN 77 and HP Fortran. Furthermore, if `fid` detects an incompatible extension whose functionality is enabled by some other means in HP Fortran, it will suggest a fix.

`fid` works by searching the entire program and reporting all its findings at once. Like the `f77` command, it expects the target program to conform to HP FORTRAN 77 syntax and will report syntax errors along with incompatibilities it detects. Unlike `f77`, however, if `fid` encounters a syntax error, it attempts to recover and continue parsing the rest of the program. This recovery mechanism allows `fid` to accept programs that contain HP Fortran language features.

Not all incompatibilities are on `fid`'s detection list. Some cannot be found by any automated means, and others require too much time to compute for even medium-sized programs.

To invoke `fid`, supply the `fid` command with one or more FORTRAN 77 source files and any desired `f77` options. If a file has been partially migrated to HP Fortran, change its extension to `.f` for use with `fid`. Following are example command lines:

```
$ fid +800 file.f
$ fid +es program.f
```

Following are examples of the warning messages `fid` issues when it detects an incompatibility:

```
fid Warning: The command-line option, +800,
             is both source incompatible
             and .o incompatible with F90

fid Warning on line 8 of file.f: ON EXTERNAL
             not supported by F90

fid Warning on line 9 of file.f: Detected IOSTAT
             specifier in OPEN statement:  Minor
             differences exist between F90 and F77
             IOSTAT error numbers
```

The incompatibilities currently detected by `fid` are:

- The I/O specifiers to the `OPEN` statement listed in “Input/output” on page 206.
- The HP FORTRAN 77 forms of `ON EXTERNAL` and `ON INTERNAL`.
- `LOGICAL` types used as operands to the `.EQ.` and `.NE.` operators.
- All HP FORTRAN 77 compiler directives except those listed in “Directives” on page 207.

- Command-line options that are not supported (see "f77 options not supported by f90" on page 202) or that have been replaced by `f90` options (see Table 34 on page 202).

NOTE `fid`'s list of incompatibilities will be periodically updated. For more information about the `fid` command, see the *fid*(1) man page.

11 Porting to HP Fortran

The goal of portability is to make it possible to compile and execute a program on different vendors' platforms, regardless of the platform on which it was written. A portable Fortran 90 program contains no language elements except those mandated by the Standard and adheres to generally accepted coding practices.

In practice, however, programming is rarely so simple. Many Fortran programs have a long history and were originally coded at a time when portability was not a concern because many programs were written to execute on one platform only. Older Fortran programs—so-called **dusty-deck programs**—are likely to have passed through different dialects of Fortran, picking up features from each, even after those features have become outmoded. **Porting** such a program may sometimes be as simple as identifying and removing the nonportable features. But more often than not, it involves finding ways to implement the functionality of the nonportable features.

To make the task of porting easier, HP Fortran includes the following features:

- **Language extensions**—statements, data types, directives, and intrinsic functions—that are compatible with other Fortran implementations.
- Compile-line options to help with the porting process.

The following sections describe these features.

- Compatibility extensions
- Using porting options

NOTE For information about migrating HP FORTRAN 77 programs to HP Fortran, refer to "Migrating to HP Fortran" on page 201.

Compatibility extensions

HP Fortran includes a variety of extensions to the Fortran 90 language. Most of these are compatibility extensions—statements, intrinsic routines, and compiler directives that are specific to nonstandard implementations of Fortran 90. For example, if you are porting a program that contains the `ACCEPT` statement, you do not have to edit the part of the program that contains this nonstandard statement because it is one of the compatibility extensions of HP Fortran.

The following sections describe the compatibility extensions. For a list of all HP Fortran language extensions, see the *HP Fortran Programmer's Reference*.

Statements

Except for the `ON` statement (see "Using the ON statement" on page 119), all of the nonstandard statements supported by HP Fortran are provided for compatibility. These are listed by vendor in Table 38. Check the description of each statement in the *HP Fortran Programmer's Reference* to confirm compatibility.

Table 38 Compatibility statements

Statement	Implementation	Description
`ACCEPT`	DEC	Reads from standard input.
`AUTOMATIC`	Sun	Allocates storage on the stack.
`BYTE`	DEC	Declares entities of type integer.
`DECODE`	Earlier versions of Fortran	Inputs formatted data from internal storage.
`DOUBLE COMPLEX`	Earlier versions of Fortran	Declares entities of type double complex.
`ENCODE`	Earlier versions of Fortran	Outputs formatted data to internal storage.

Statement	Implementation	Description
`END` (structure definition)	DEC	Terminates the definition of a structure or union.
`MAP`	DEC	Defines a union within a structure.
`POINTER` (Cray-style)	Cray	Declares Cray-style pointers and their objects.
`RECORD`	DEC	Declares a record of a previously defined structure.
`STATIC`	Sun	Allocates storage in static memory.
`STRUCTURE`	DEC	Defines a named structure.
`TYPE` (I/O)	DEC	Writes to standard output.
`UNION`	DEC	Defines a union within a structure.
`VIRTUAL`	DEC	Declares an array.
`VOLATILE`	DEC	Allows data sharing between asynchronous processes.

Compiler directives

Compiler directives are coded lines in the source file that control the compiler's state. Many vendors use a directive syntax that enables the compiler to treat the directive as a comment unless the compiler is specifically looking for that directive. For example, all directives recognized by HP Fortran begin with the character `!` in free format or `C`, `*`, or `!` in fixed format (in fixed format, the directive must also start in column 1).

A directive that uses the comment-like syntax will not cause the compilation to fail. However, if the compiler does not recognize the directive, then the functionality that the directive enables will be lost.

The directives listed in Table 39 are recognized by HP Fortran and are compatible with those available on other implementations. These directives are functionally compatible; that is, their effect on HP Fortran is compatible with that on the original implementation. Refer to the *HP Fortran Programmer's Reference* for detailed descriptions of the directives to check the level of compatibility. For usage information about these directives, see "Controlling vectorization locally" on page 150.

As noted in the table, some of the compatibility directives are effective only if the source file is compiled with either the `+Oparallel` or the `+Ovectorize` option; otherwise, the directive is treated as a comment and ignored. For information about using these options, see "Using the +Ovectorize option" on page 149.

Table 39 Compatibility directives

Vendor	Directive	Function	Option dependency
Cray	`DIR$ IVDEP`	Disables dependency checks.	`+Oparallel` or `+Ovectorize`
	`DIR$ NO SIDE EFFECTS`	Disables checks for side effects.	`+Oparallel` or `+Ovectorize`
	`DIR$ [NO]CONCUR`	Enables [disables] code generation for parallel execution.	`+Oparallel`
	`DIR$ [NO]VECTOR`	Enables [disables] vectorization.	`+Ovectorize`
	`FPP$ NODEPCHK`	Disables dependency checks	`+Oparallel` or `+Ovectorize`
KAI	`*$* [NO]CONCURRENTIZE`	Enables [disables] code generation for parallel execution.	`+Oparallel`

Vendor	Directive	Function	Option dependency
	`*$* [NO]VECTORIZE`	Enables [disables] vectorization.	`+Ovectorize`
VAST	`VD$ [NO]VECTOR`	Enables [disables] vectorization.	`+Ovectorize`
	`VD$ NODEPCHK`	Disables dependency checks.	`+Oparallel` or `+Ovectorize`

HP Fortran also recognizes several directive prefixes. A directive prefix is a vendor-specific sequence of characters that follows the comment character and precedes the directive name. The recognized prefixes are listed by vendor in Table 40. If HP Fortran reads a directive that begins with one of these prefixes but does not recognize the directive name, it issues a warning and ignores the directive. A directive takes effect only if the compiler recognizes both its prefix and name—that is, it must be either one of HP's own directives or one of those listed in Table 39.

Table 40 Directive prefixes recognized by HP Fortran

Prefix	Vendor
`$`	SGI
`$HP$`	HP
`$PAR`	X3H5
`*$*`	KAI
`DIR$`	Cray
`FPP`	Cray
`HPF$`	High Performance Fortran
`VD$`	VAST

Intrinsic procedures

In addition to the standard Fortran 90 intrinsics, HP Fortran provides a number of nonstandard intrinsics. Many of these are compatible with nonstandard intrinsics available on other implementations. Table 41 lists all HP Fortran nonstandard intrinsics by their generic names. Where a *specific* intrinsic exists, it can be accessed by referencing its generic name. See the *HP Fortran Programmer's Reference* for information about both specific and generic intrinsics.

Table 41 Nonstandard intrinsic procedures in HP Fortran

ABORT	DREAL	IDIM	IXOR	RSHFT
ACOSD	EXIT	IGETARG	JNUM	RSHIFT
ACOSH	FLUSH	IJINT	LOC	SECNDS
AND	FNUM	IMAG	LSHFT	SIND
ASIND	FREE	INT1	LSHIFT	SIZEOF
ASINH	FSET	INT2	MALLOC	SRAND
ATAN2D	FSTREAM	INT4	MCLOCK	SYSTEM
ATAND	GETARG	INT8	OR	TAND
ATANH	GETENV	INUM	QEXT	TIME
BADDRESS	GRAN	IOMSG	QFLOAT	XOR
COSD	HFIX	IQINT	QNUM	ZEXT
DATE	IACHAR	IRAND	QPROD	
DCMPLX	IADDR	IRANP	RAN	
DFLOAT	IARGC	ISIGN	RAND	
DNUM	IDATE	ISNAN	RNUM	

HP Fortran also provides nonstandard specific intrinsics that derive from standard generic intrinsics; these nonstandard specific intrinsics are not listed in Table 41. They provide generic intrinsics with the ability to operate on nonstandard data type sizes. For example, the generic intrinsic `ABS` is defined by the Fortran 90 Standard to return the absolute value of the standard data types. HP Fortran provides `BABS` and `ZABS` as extensions, enabling `ABS` to operate on `INTEGER(KIND=1)` and `DOUBLE COMPLEX` values—both of which are nonstandard. Many of the nonstandard specific intrinsics (including `BABS` and `ZABS`) are compatible with similarly named intrinsics available on other implementations.

Using porting options

HP Fortran provides a number of compile-line options for porting programs. The most important of these is the `+langlvl=90` option. Compiling your program with this option will cause the compiler to issue warning messages for all nonstandard features.

In addition, HP Fortran includes options that provide compatibility by changing the compiler's assumptions about the program or by causing the compiler to generate code that executes compatibly with the original implementation. The advantage of using options when porting is that they minimize having to edit and modify source code.

The following sections describe how options can help when porting programs that contain:

- Initialized variables
- Data types that are larger than the default sizes of HP Fortran data types
- Names that clash with HP-specific intrinsics
- Names that end in the underscore character (_)
- One-trip `DO` loops
- Different formats
- Escape sequences

Uninitialized variables

As noted in "Automatic and static variables" on page 91, the default behavior of HP Fortran is to allocate storage for program variables from the stack. However, older implementations of Fortran often allocate static storage for variables. One of the differences between stack storage and static storage is that static variables are initialized to `0`s by the compiler, whereas automatic variables (variables allocated from the stack) must be explicitly initialized by the programmer.

Programs written for implementations of Fortran that allocate static storage by default sometimes rely on the compiler to initialize variables. Compiling and executing such programs on implementations that

allocate stack storage can have disastrous results. To make HP Fortran compatible with implementations that allocate static storage, compile with the `+save` option. This option causes the compiler to act as though all local variables had the `SAVE` attribute.

As mentioned in "Automatic and static variables" on page 91, saving all variables in static storage can degrade performance. If performance is an issue, consider using the `+Oinitcheck` option. Unlike the `+save` option, `+Oinitcheck` does not "save" variables—it does not move variables into static storage. Instead, it causes the compiler to search for all local, nonarray, nonstatic variables that have not been defined before being used. Any that it finds are initialized to 0 on the stack each time the procedure in which they are declared is invoked.

For detailed information about the `+save` and `+Oinitcheck` options, see *HP Fortran Programmer's Reference*.

Large word size

The word size of default integers, reals, and logicals in HP Fortran is 4 bytes. However, some implementations of Fortran 90—notably, Cray—use an 8-byte word size. Programs written for these implementations may rely on the increased precision and range in their computations.

You can double the sizes of default integer, real, and logicals by compiling with the `+autodbl` option, making them compatible with implementations that use the larger word size. This option also doubles the sizes of items declared with the `COMPLEX` and `DOUBLE PRECISION` statements, but not the `BYTE` and `DOUBLE COMPLEX`) statements.

Increasing the size of double-precision items can degrade the performance of your program. If you do not need the extra precision for items declared with the `DOUBLE PRECISION` statement, use the `+autodbl4` option, which increases single-precision items only. Compiling with this option results in items declared as default real and double precision real having the same precision—a violation of the Standard.

For usage information about the `+autodbl` and `+autodbl4` options, see "Increasing default data sizes" on page 96). For detailed descriptions of these options, refer to the *HP Fortran Programmer's Reference*.

One-trip DO loops

If a DO loop is coded so that its initial loop count is greater than its final loop count, standard Fortran 90 requires that the loop never execute. However, under some implementations of FORTRAN 66, if a DO loop is reached, it executes for at least one iteration, even if the DO variable is initialized to a value greater than the final value. This is called a **one-trip DO loop**.

To duplicate the behavior of a one-trip DO loop in an HP Fortran program, compile with the +onetrip option. To see the effects of this option, consider the following program:

```
PROGRAM main

   DO 10 i = 2, 1
     PRINT *, 'Should never happen in standard Fortran 90.'
10 CONTINUE
END PROGRAM main
```

When compiled with the command line:

```
$ f90 test_loop.f90
```

the PRINT statement will never execute because the initial loop count is higher than the final loop count. To force the loop to execute at least once, compile it with the command line:

```
$ f90 +onetrip test_loop.f90
```

When you run the program now, it produces the output:

```
$ a.out
 Should never happen in standard Fortran 90.
```

Name conflicts

A common problem in porting Fortran programs is name conflicts: a user-written procedure may have the same name as an intrinsic procedure on the implementation to which you are porting, and the compiler selects the name of the intrinsic when you are expecting it to call the user-written procedure. For example, HP Fortran provides the nonstandard intrinsic FLUSH. If your program contains an external procedure with the same name and the procedure is *not* declared with the EXTERNAL statement, the HP Fortran compiler will assume that the reference is to the intrinsic.

One way to identify user routines that have the same names as HP-specific intrinsics is to compile the program with the `+langlvl=90` option. This option causes the compiler to issue warnings for all HP extensions in the source code, including nonstandard intrinsics. You can then edit the source file to declare the procedure that the compiler assumes is an intrinsic with the `EXTERNAL` statement.

The following are programs that illustrate the preceding concepts.

clash.f90

```
PROGRAM clash
  i = 4
  j = int1(i)
  PRINT *, 'j =', j
END PROGRAM clash

FUNCTION int1(i)
  int1 = i+1
END FUNCTION int1
```

If this is compiled as coded and without the `+langlvl=90` option, the compiler will assume that the reference is to the HP intrinsic named `INT1` and not to the external function. Executing the program will produce unexpected results, as appears in the following sample run:

```
$ f90 clash.f90
clash.f90
   program CLASH
   external function INT1

11 Lines Compiled
$ a.out
 j = 4
```

If the program is recompiled with the `+langlvl=90` option, the compiler flags the name of what it assumes to be a nonstandard intrinsic as well as the nonstandard source format:

```
$ f90 +langlvl=90 clash.f90
   program CLASH

 i = 4
^
Warning 4 at (3:clash.f90) : Tab characters are an extension to
standard Fortran-90
 j = int1(i)
     ^
Warning 39 at (5:clash.f90) : This intrinsic function is an
extension to standard Fortran-90
   external function INT1

 int1 = i+1
```

```
^
Warning 4 at (10:clash.f90) : Tab characters are an extension to
standard Fortran-90

11 Lines Compiled
```

Once you have identified the names of your routines that clash with intrinsic names, you can edit the source code to declare each procedure with the `EXTERNAL` statement, as follows:

```
EXTERNAL int1
```

Now when you compile and execute, you will get the expected behavior:

```
$ f90 clash.f90
clash.f90
   program CLASH
   external function INT1

11 Lines Compiled
$ a.out
 j = 5
```

NOTE The name-conflict problem can occur in Fortran programs that call routines in the `libU77.a` library. Some implementations link `libU77.a` by default. HP Fortran does not; to link in this library, you must compile your program with the `+U77` option. If you do not compile with this option and your program references a `libU77` routine with the same name as an HP Fortran intrinsic, the compiler will wrongly (and sometimes disastrously) assume that the reference is to an intrinsic.

If you are not sure if your program references `libU77` routines, compile it with the `+langlvl=90` option, which will cause the compiler to issue warnings for references to nonstandard routines. For problems that can occur when migrating HP FORTRAN 77 programs that reference `libU77` routines, see “Intrinsic functions” on page 204.

Names with appended underscores

In some implementations of Fortran (but not HP Fortran), the compiler automatically appends underscores to external names. If you are porting a mixed-language program from such an implementation (for example, a program consisting of C and Fortran source files), the linker may not be able to find the names in the C code because the names in the Fortran code do not have the appended underscore. The reason is that the C code has explicitly added underscores to match the names of the Fortran procedures in the object code.

Using the `+ppu` option causes the HP Fortran compiler to append an underscore to external names (including procedures and common blocks), making them consistent with the name as it appears in the non-Fortran source file. For example, if a Fortran source file contains the procedure `proc_array`, and a C source file reference this procedure as `proc_array_`, compiling the Fortran source file with the `+ppu` option causes the compiler to use `proc_array_` as the name of the procedure in the Fortran object file.

For information about how to resolve other name conflicts in mixed-language programs, see "Case sensitivity" on page 170.

Source formats

Standard Fortran 90 permits source code in either fixed or free form, though not both in the same file. Furthermore, if the source is in fixed form, the Standard requires statements not to extend beyond column 72. Also, Standard Fortran 90 does not allow tab formatting.

HP Fortran's scheme for handling the different formatting possibilities is this:

- If the name of the source file ends with the `.f90` extension, the file is compiled as free form. The compiler accepts tab characters in the source.
- If the name of the source file ends with the `.f` or `.F` extension, the file is compiled as fixed form.
- If the file is compiled with the `+langlvl=90` option, the interprets the format as either fixed or free form, depending on the filename extension (as described above). However, the compiler issues warnings if it encounters tab characters.

- If the file is compiled with the `+source=fixed` option, the compiler assumes fixed form, regardless of the extension. Tab characters are allowed.
- If the file is compiled with the `+source=free` option, the compiler assumes free form, regardless of the extension.
- If the file is compiled with the `+extend_source` option, the compiler allows lines as long as 254 characters in either fixed or free form. The default line length is 72 characters for fixed form and 132 characters for free form.

See the *HP Fortran Programmer's Reference* for detailed information about the different source and the `+langlvl=90`, `+source`, and `+extend_source` options.

Escape sequences

Some implementation of Fortran process certain characters preceded by the backslash (`\`) as a C-like escape sequence. For example, if a program containing the statement:

```
PRINT *, 'a\nb\nc'
```

were compiled under an implementation that recognized escape sequences, the statement would output:

```
 a
b
c
```

When compiled in strict compliance with the Standard, the same statement would output:

```
 a\nb\nc
```

Although HP Fortran does not recognize escape sequences by default, you can use the `+escape` option to make the compiler to recognize them. Refer to the *HP Fortran Programmer's Reference* for more information about escape sequences.

Glossary

A-B

archive library A library of routines that can be linked to an executable program at link-time. The names of archive libraries have the `.a` extension.

See also shared library.

aliasing Referencing a variable by more than one name. Examples of aliasing include:

- Passing the same variable as two or more actual arguments.
- Using the `EQUIVALENCE` statement.
- Referencing an element of an array declared in common with an out-of-bounds subscript.
- Passing a common variable as an actual argument.

In general, aliasing inhibits optimization.

alignment The positioning of data within memory. Except for objects larger than 8 bytes, HP Fortran 90 aligns data on a byte boundary that is a multiple of its size. Objects larger than 8 bytes are aligned on 8-byte boundaries.

automatic variable A variable that is allocated on the stack. By default, program variables in HP Fortran 90 are automatic. Two characteristics of automatic variables are of note:

- They are allocated at each invocation of the procedure in which they are declared and deallocated upon return from the procedure. This means that automatic variables do not retain their value between invocations.
- They must be explicitly initialized.

See also static variable.

back-end The component of the compiler that optimizes and generates object code.

See also front-end.

Basic Linear Algebra Subroutine library A library of de facto standard routines for performing low-level vector and matrix operations. To access routines in this library, you must compile with the `-lblas` option.

BLAS *See Basic Linear Algebra Subroutine library*.

BOZ constant An integer constant that is used as an initializer in a `DATA` statement and is formatted in binary (`B`), octal (`O`), or hexadecimal (`Z`) notation.

buffering, tty
See tty buffering.

built-in functions The two HP Fortran 90 extensions, `%VAL` and `%REF`. `%VAL` forces an argument to be passed by value, and `%REF` forces it to be passed by reference.

C-D

cpp
See C preprocessor.

C preprocessor A C language utility that removes or adds statements in a program source text, in accordance with directives that have been inserted in the source file. HP Fortran 90 can pass source files to the C preprocessor (`cpp`) for preprocessing and then send the output to the compiler.

column-major order The method of storing Fortran 90 arrays in memory. Column-major order requires the columns of a two-dimensional array to be in contiguous memory locations. For example, given the array `a(3,4)`, element `a(1,1)` would be stored in the first location, `a(2,1)` in the second, `a(3,1)` in the third, and so on.

See also row-major order.

core dump A core image of an executing program that is deposited in a file after the program aborted execution. The core dump (also called a *core file*) may contain information that is useful in debugging the aborted program.

data dependence The relationship that can obtain between the definition of data and its use. The occurrence of a data dependence in a loop can prevent the optimizer from **parallelizing** it.

dde The command for invoking the **HP Distributed Debugging Environment**, the source-level debugger that is included with HP Fortran 90.

debugger
See HP Distributed Debugging Environment.

division by zero The floating-point **exception** that occurs whenever the system attempts to divide a nonzero value by zero.

driver The component of the compiler that retains control throughout the entire compilation process.

dusty-deck programs Older, pre-FORTRAN 77 programs. Dusty-deck programs are so called because they were presumably encoded and stored on punched cards. Such programs are difficult to **port** and **optimize**.

E-K

exception A condition occurring during the execution of a program that may require special handling to make further execution

meaningful. Some exceptions can be **trapped** by the system and handled within the program.

extension
See filename extension and language extension.

fast underflow A hardware feature for handling **underflow** by substituting zero for the operation that causes the underflow.

file descriptor An integer that is returned by certain HP-UX system I/O routines and then passed to others to provide access to a file. A file descriptor is similar to Fortran's logical unit number. When the Fortran 90 intrinsic `FNUM` is given a logical unit number, it returns a file descriptor.

filename extension A sequence of characters that begins with a period (.) and is added to a filename to indicate the function or contents of the file.

See also language extension.

floating-point exception
See exception.

front-end The component of the compiler that parses source code and issues warning and error messages.

See also back-end.

High-Level Optimizer One of the optimizing components of HP Fortran 90 that performs **optimizations** across procedures and files.

HLO *See High-Level Optimizer.*

HP DDE *See HP Distributed Debugging Environment.*

HP Distributed Debugging Environment. The source-level debugger for HP Fortran 90 programs.

See also dde.

integer overflow An **exception** condition that occurs when attempting to use an integer to represent a value that falls outside its range. The `ON` statement can be used to trap integer overflow.

invalid operation The floating-point **exception** that occurs whenever the system attempts to perform an operation that has no numerically meaningful interpretation, such as a **NaN**.

L-N

language extension A feature of a programming language that has been added by a vendor and is not defined in (or is in violation of) the language standard. The `ON` statement is an HP language extension to the Fortran 90 Standard.

See also filename extension.

libU77 routines Routines in the BSD 3f library (`libU77.a`) that provide a Fortran 90 interface to selected system calls in `libc.a`. The `libU77.a` library is part of HP Fortran 90 and is accessed with the `+U77` option.

migrating In this document, *migrating* refers to the processing of moving a program written for HP FORTRAN 77 to HP Fortran 90.

See also porting.

memory fault
See segmentation violation.

millicode routines Millicode versions of frequently called intrinsics, having very low call overhead and little error-handling. One of the optimizations performed by HP Fortran 90 is to replace calls to eligible intrinsics with millicode versions.

.mod file A file that is created and read by the compiler when processing Fortran 90 source files that define or use **modules**.

module A type of Fortran 90 program unit that is used for sharing data. Modules can also be used to contain subprograms.

NaN Not-a-Number, the condition that results from a floating-point operation that has no mathematical meaning, such as infinity divided by infinity. The `ON` statement can be used to trap operations that result in NaN.

null The null character (`'\0'`) that is used in C programs to terminate strings.

O-Q

one-trip DO loop A `DO` loop that, if reached, executes for at least one iteration. Programs written for some implementations of FORTRAN 66 rely on one-trip `DO` loops.

optimization Code transformations made by the compiler to improve program performance.

overflow An **exception** condition that occurs when the result of a floating-point operation is greater than the largest normalized number.

See also integer overflow.

parallel execution Program execution on multiple processors at the same time. One of the optimizations performed by the compiler is to transform eligible program loops for parallel execution.

parallelization An optimization that transforms eligible program loops for **parallel execution** on a multiprocessor machine.

PIC
See position-independent code.

porting In this document, *porting* refers to the process of moving a program that was coded for another vendor's Fortran to HP Fortran 90.

See also migrating.

position-independent code Object code that contains no absolute addresses. Position-independent code (PIC) has linkage tables that contain pointers to code and data. This table is filled in by the loader at

runtime. Object code that consists of PIC can be used to create **shared libraries**.

precision The number of digits to which floating-point numbers are represented. Double-precision numbers can have greater precision than single-precision numbers.

profilers Programming tools that determine where a program spends its execution time. Profilers that come with HP Fortran 90 include `prof`, `gprof`, and **CXperf**.

R-S

roundoff error The loss of precision that can occur as a result of floating-point arithmetic. Different orders of evaluating a floating-point expression can produce different accumulations of roundoff errors, which in turn can sometimes cause the expression to yield significantly different results.

row-major order The method of storing C-language arrays in memory. (Fortran arrays are stored in **column-major order**.) Row-major order requires the rows of a two-dimensional array to be in contiguous memory locations. For example, given the array `a[3][4]`, element `a[0][0]` would be stored in the first location, `a[0][1]` in the second, `a[0][2]` in the third, and so on.

segmentation violation A type of **exception** that occurs when an executing program attempts to access memory outside of its allocated memory segment; also called a *memory fault*.

serial execution Program execution on only one processor at a time.

See also parallel execution.

shared executable An executable program whose text segment (that is, its code) can be shared by multiple processes.

shared library A library of routines that can be linked to an executable program at runtime and shared by several programs simultaneously. The names of shared libraries have the `.sl` extension.

See also archive library.

side effects A condition that prevents the optimizer from **parallelizing** a loop. A procedure that is called within a loop has side effects if it communicates with the outside world other than through a return value.

signal
See trap.

stack overflow An error condition that occurs when the runtime system attempts to allocate more memory from the stack than is available. This condition can occur when attempting to allocate very large arrays or when a recursive program is out of control.

static variable Variables that are allocated from static storage (sometimes referred to as the *heap*). Static variables have two characteristics of note:

- They preserve their value for the lifetime of the program.
- They are initialized when they are allocated.

By default, program variables in HP Fortran 90 are **automatic**.

stream I/O A type of I/O that is based on the concept of a stream—a flow of data to or from a file or I/O device. Streams are managed by the HP-UX operating system. Access to a stream is provided by a stream pointer, which is the address of a C-like structure that contains information about a stream. When the Fortran 90 intrinsic `FSTREAM` is given a logical unit number, it returns a stream pointer, providing Fortran programs with access to stream-based system routines.

symbol table A table of names of procedures and data, including their offset addresses. The compiler inserts a symbol table in the object file for use by the debugger and profiler.

T-Z

thread An independent flow of control within a single process, having its own register set and program counter. The HP-UX operating system supports multiple-executing threads within the same process.

Thread Trace Visualizer *See ttv*.

trap A change in system state that is caused by an **exception** and that may be detected by the executing program that took the exception. Traps are hardware features that may be enabled or disabled. If traps are enabled, they can change the flow of control in the program that took the exception. In response to a trap, the system may generate a signal (for example, `SIGFPE`), which the program can detect. Such a program can be designed to handle traps. HP Fortran 90 provides the `ON` statement to handle traps.

ttv A tool for analyzing parallel-executing programs.

tty buffering A method for efficiently processing data that is directed to standard output by capturing it in a buffer before sending it to the screen.

underflow An **exception** condition that occurs when the result of a floating-point operation is smaller than the smallest normalized number. On systems that support it, **fast underflow** is an efficient method of handling this exception.

vectorization An optimization technique that replaces eligible program loops that operate on arrays with calls to specially tuned routines that perform the same operation.

wall-clock time Time spent by an executing program that includes system time as well as

process time. In contrast, *virtual time* takes into account process time only. Profilers (such as **CXperf**) that track both virtual time and wall-clock time provide information about when a program is blocked as well as when it is running.

Index

Numerics

A

D

F

G

S

T