HP Fortran 90 1.1 Release Notes

HP 9000 Computers

HP Part No. 5965-4445
Printed in USA 05/97

First Edition
E0597

FINAL TRIM SIZE : 7.5 in x 9.0 in

Legal Notices

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

Hewlett-Packard makes no warranty of any kind with regard to this manual, 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 direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

This software and documentation is based in part on materials licensed from The Regents of the University of California. We acknowledge the role of the Computer Systems Research Group and the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley and the other named Contributors in their development.

Trademarks. The trademarks used in this document are:

UNIX — UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Preface

This document contains the following chapters:

- New and Changed Features
- Using HP Fortran 90
- Installation
- Relevant Documentation
- Problem Descriptions and Fixes

Contents

Tables

1

New and Changed Features

The 1.1 release of the HP Fortran 90 compiler supports HP 9000 workstations and servers running on 10.30 HP-UX systems.

This release document describes new and changed features of the 1.1 release of the HP Fortran 90 compiler. The 1.0 release of HP Fortran 90 is fully documented in the *HP Fortran 90 Programmer's Reference* (B3908-90001).

New Features

The new features for release 1.1 include:

- New statements to provide compatibility with other vendors' Fortran compilers
- `+Oparallel` optimization option
- Compatibility directives for parallelizing and vectorizing programs
- `MP_NUMBER_OF_THREADS` environment variable
- Support for multi-threaded programming
- Support for large-file systems

The following sections fully describe these features.

HP Fortran 90 Statements

The following statements are supported by this release of HP Fortran 90:

- `BUFFER IN` statement
- `BUFFER OUT` statement
- `OPTIONS` statement

All of these statements are extensions to the Fortran 90 standard. They are fully described in the following sections. For a full discussion of all other HP Fortran 90 statements, refer to the *HP Fortran 90 Programmer's Reference*, Chapter 10.

BUFFER IN statement (Extension)

Provides compatibility with the Cray `BUFFER IN` statement

Syntax

`BUFFER IN (`*unit*`,` *mode*`) (`*begin-loc*`,` *end-loc*`)`

unit	is either a unit identifier (integer expression) or a file name (character expression)
mode	is a mode identifier (integer expression) that controls the record position. If *mode* is ≥ 0, full-record processing occurs, as in standard Fortran I/O. If *mode* is < 0, partial-record processing occurs; that is, if n is the last word that was transferred, then the record is positioned to transfer the n`+1` word.
begin-loc, *end-loc*	are symbolic names of the variables, arrays, or array elements that mark the beginning and end locations of the `BUFFER IN` operation. *begin-loc* and *end-loc* must be either elements of a single array (or equivalenced to an array) or members of the same common block.

Description

The `BUFFER IN` statement is an HP Fortran 90 extension that provides compatibility with the Cray `BUFFER IN` feature. The statement causes data to be transferred while allowing any subsequent statements to execute concurrently. The `BUFFER IN` statement is provided as a porting aid for existing Cray code; it is not likely to produce superior performance compared to conventional Fortran 90 I/O methods.

The following restrictions apply to the `BUFFER IN` statement:

- Any data format conversions specified in the `OPEN` statement do not affect data read or written with the `BUFFER IN` statement.
- Using the `BUFFER IN` statement with data types less than eight bytes produces a fatal compiler error. Use the `+autodbl` option to increase the size of numeric and logical default data types, as described in the *HP Fortran 90 Programmer's Reference*, Chapter 13.

- Other Fortran I/O statements (for example, `READ`, `WRITE`, `PRINT`, `ACCEPT`, and `TYPE`) cannot be used on the same unit as the `BUFFER IN` statement.
- The `BACKSPACE` statement cannot be used with files that are capable of being transferred by the `BUFFER IN` statement. Such files are referred to as *pure-data* (unblocked) files.

HP Fortran 90 also provides the following library routines for use with the `BUFFER IN` statement:

stat `= UNIT(`*unit*`)`	returns the status of a `BUFFER IN` operation. *stat* is of type real, and *unit* is of type integer.
len `= LENGTH(`*unit*`)`	returns the number of words successfully transferred. *len* and *unit* are of type integer.
pos `= GETPOS(`*unit*`)`	returns the current record number. *pos* and *unit* are of type integer.
`SETPOS(`*unit*`,` *pos*`)`	sets the record number, *pos*, for a file. *pos* and *unit* are of type integer. This routine cannot be used with magnetic tape; doing so produces an error message.

These routines reside in `libcl.[a|sl]` and are implicitly loaded by the linker. If you want to use them, make sure that your own routines don't have the same names.

Example

The following program shows how to use the `BUFFER IN` and `BUFFER OUT` statements. The program must be compiled with the `+autodbl` option; see *HP Fortran 90 Programmer's Reference*, Chapter 13.

```
PROGRAM bufferedIoTest

! buffered i/o example: compile with +autodbl

  INTEGER a(10)

  OPEN ( UNIT = 7, NAME = 'test.dat', FORM = 'UNFORMATTED' )

  a = (/ (i,i=1,10) /)                    ! initialize the array a
```

```
  BUFFER OUT ( 7, 0 ) ( a, a(10) )      ! write out array a twice
  CALL unit  ( 7 )
  BUFFER OUT ( 7, 0 ) ( a, a(10) )
  CALL unit  ( 7 )

  ! now position the file 40 bytes (5 integer values) into the file
  CALL setpos ( 7, 5 )

  ! read the remainder of the 1st record, and half of the second
  BUFFER IN ( 7, 0 ) ( a, a(10) )

  WRITE(6,*) a
  CLOSE (7)
END PROGRAM bufferedIoTest
```

BUFFER OUT statement (Extension)

Provides compatibility with the Cray `BUFFER OUT` statement

Syntax

`BUFFER OUT (`*unit*`,` *mode*`) (`*begin-loc*`,` *end-loc*`)`

unit	is either a unit identifier (integer expression) or a file name (character expression).
mode	is a mode identifier (integer expression) that controls the record position. If *mode* is ≥ 0, full-record processing occurs, as in standard Fortran I/O. If partial-record processing is in progress, *mode* ≥ 0 ends a series of partial-record transfers. If *mode* is < 0, the record is left positioned to receive additional words.
begin-loc, *end-loc*	are symbolic names of the variables, arrays, or array elements that mark the beginning and end locations of the `BUFFER OUT` operation. *begin-loc* and *end-loc* must be either elements of a single array (or equivalenced to an array) or members of the same common block.

Description

The `BUFFER OUT` statement is an HP Fortran 90 extension that provides compatibility with the Cray `BUFFER OUT` feature. The statement causes data to be transferred while allowing any subsequent statements to execute concurrently. The `BUFFER OUT` statement is provided as a porting aid for existing Cray code; it is not likely to produce noticeably superior performance compared to conventional Fortran 90 I/O methods. In fact, the `BUFFER OUT` statement will always be slightly slower than unformatted fixed record length I/O.

The following restrictions apply to the `BUFFER OUT` statement:

- Any data format conversions specified in the `OPEN` statement do not affect data read or written with the `BUFFER OUT` statement.
- Using the `BUFFER OUT` statement with data types less than eight bytes in size produces a fatal compiler error. Use the `+autodbl` option to increase the size

of numeric and logical default data types, as described in the *HP Fortran 90 Programmer's Reference*, Chapter 13.

- Other Fortran I/O statements (for example, `READ`, `WRITE`, `PRINT`, `ACCEPT`, and `TYPE`) cannot be used on the same unit as the `BUFFER OUT` statement.
- The `BACKSPACE` statement cannot be used with files that are capable of being transferred by the `BUFFER OUT` statement. Such files are referred to as *pure-data* (unblocked) files.

HP Fortran 90 also provides the following library routines for use with the `BUFFER OUT` statement:

stat `= UNIT(`*unit*`)`	returns the status of a `BUFFER IN` operation. *stat* is of type real, and *unit* is of type integer.
len `= LENGTH(`*unit*`)`	returns the number of words successfully transferred. *len* and *unit* are of type integer.
pos `= GETPOS(`*unit*`)`	returns the current record number. *pos* and *unit* are of type integer.
`SETPOS(`*unit*`,` *pos*`)`	sets the record number, *pos*, for a file. *pos* and *unit* are of type integer. This routine cannot be used with magnetic tape; doing so produces an error message.

These routines reside in `libcl.[a|sl]` and are implicitly loaded by the linker. If you want to use them, make sure that your own routines don't have the same names.

Example

For an example of the `BUFFER OUT` statement, see the description of the `BUFFER IN` statement.

OPTIONS statement (Extension)

Changes the optimization level.

Syntax

```
OPTIONS +On
```

where `+On` (or `-On`) specifies a level of optimization that is equal to or less than the level specified on the command line. The `+On` and `-On` options are described in the *HP Fortran 90 Programmer's Reference*, Chapter 13.

Description

The `OPTIONS` statement is an extension of HP Fortran 90 and is used to specify a level of optimization that is equal to or less than the level specified on the command line; if the level specified by the `OPTIONS` statement is higher than that specified on the command line, the statement is ignored. The `OPTIONS` statement must be placed outside all program units. The changed level of optimization applies to the beginning of the next program unit and remains in effect for all succeeding program units or until superseded by another `OPTIONS` statement or by the `$HP$ OPTIMIZE` directive.

The `$HP$ OPTIMIZE` directive takes precedence over the `OPTIONS` statement. That is, if the directive is used to disable optimization, any subsequent `OPTIONS` statement has no effect until a later directive enables optimization.

Note The `OPTIONS` statement differs from the `$HP$ OPTIMIZE` directive, which enables or disables optimization but does not change the optimization level. For more information about the `$HP$ OPTIMIZE` directive, refer to the *HP Fortran 90 Programmer's Reference*, Chapter 14.

Example

In the following example, the first `OPTIONS` statement optimizes the subroutine `go_fast` at optimization level 3. The second `OPTIONS` statement lowers the optimization level to 2. It is assumed that the file that contains this code was compiled with the `+O3` or `-O3` option.

```
OPTIONS +O3
SUBROUTINE go_fast
```

```
:
END SUBROUTINE go_fast

OPTIONS +O2
SUBROUTINE not_so_fast
:
END SUBROUTINE not_so_fast
```

+Oparallel Option

The HP Fortran 90 compiler supports the `+O[no]parallel` optimization option. This option optimizes a program for parallel execution.

Syntax

```
+O[no]parallel
```

Description

The `+Oparallel` option causes the compiler to transform eligible loops for parallel execution on multiprocessor machines. This option requires the `+O3` option.

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

If you link separately from the compile 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.

Note The `+Oparallel` option should not be used for programs that make explicit calls to the kernel threads library (see Table 2-7).

For information about using the `+Oparallel` option to parallelize your Fortran programs, see "Parallelizing HP Fortran 90 Programs" in Chapter 2. To set the number of processors that will execute your program, use the `MP_NUMBER_OF_THREADS` environment variable; see "`MP_NUMBER_OF_THREADS` Environment Variable".

Compiler Directives

HP Fortran 90 supports the compiler directives listed in Table 1-1. These directives are provided for compatibility with programs developed on the platforms also listed in the table.

Table 1-1. Compatibility Directives Recognized by HP Fortran 90

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 immediately 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 90 comment character (`!`). If an option or argument is included with the directive name, the compiler will ignore the directive.

The following sections describe the directives in detail.

Controlling Vectorization

HP Fortran 90 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 file or until superseded by a later directive. For more information about this option, see *HP Fortran 90 Programmer's Reference*, Chapter 13; for information about the BLAS library, see Chapter 12.

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 file 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 "Conditions Inhibiting Loop Parallelization" in Chapter 2. For more information about parallelizing your Fortran programs, see "Parallelizing HP Fortran 90 Programs" in Chapter 2.

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. (For a discussion of why loops with a data dependence cannot be parallelized, see "Data Dependencies" in Chapter 2.) 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.

Note 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. (For a discussion of side effects and why they can prevent parallelization, see "Calling Routines with Side Effects" in Chapter 2.) 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 90 does not enforce this requirement.

`MP_NUMBER_OF_THREADS` Environment Variable

The `MP_NUMBER_OF_THREADS` environment variable enables you to set the number of processors that are to execute your program in parallel. If you do not set this variable, it defaults to the number of processors on the executing machine.

On the C shell, the following command sets `MP_NUMBER_OF_THREADS` to indicate that programs compiled for parallel execution can execute on two processors:

```
setenv MP_NUMBER_OF_THREADS 2
```

If you use the Korn shell, the command is:

```
export MP_NUMBER_OF_THREADS=2
```

To optimize your program for parallel execution, you must use the `+O[no]parallel` option; see "+Oparallel Option". For information about other environment variables that are available with HP Fortran 90, see "Environment Variables" in Chapter 2.

Multi-Threaded Programming

HP Fortran 90 programs can execute in a multi-threaded environment; see "+Oparallel Option". The HP Fortran 90 I/O library has been thread-safed for correct execution within this environment.

Support for Large File Systems

Programs compiled with this release of HP Fortran 90 can create files greater than 2 gigabytes. The program must be running on HP-UX 10.30, and you must have an HP-UX Hierarchical File System (HFS) that is configured for large files. By default, the HP-UX HFS file system does not allow files greater than 2 gigabytes. To allow large files, use the `fsadm` command; for more information, see the *fsadm*(1m) and *fsadm_hfs*(1m) man pages.

The maximum size of a record and the maximum number of records in a direct-access file continue to have the same limit, 2 gigabytes—that is, `MAXINT`, or 2147483647.

Changes to HP Fortran 90

The following sections document changes to HP Fortran 90 for this release.

PA-RISC 1.0 Architecture Not Supported

Starting with this release, the HP Fortran 90 compiler no longer supports the PA-RISC 1.0 architecture. This means that the `+DA` and `+DS` compile-line options will not accept the `1.0` argument. Refer to the *f90*(1) man page or to the *HP Fortran 90 Programmer's Reference*, Chapter 13, for acceptable arguments to these options.

Instruction Scheduling

Instruction scheduling is determined (as currently documented) by the argument you specify with the `+DS` option. However, if you do not use this option, the compiler will use the argument you specify with the `+DA` option. If you specify neither `+DA` nor `+DS`, the default instruction scheduling is based on that of the system on which you are compiling.

The following command lines summarize the change:

```
# code generation and instruction scheduling based on the
#    model of the machine on which f90 is executing:
f90 prog.f90

# code generation based on 1.1, instruction scheduling on 2.0
f90 +DS1.1 +DA2.0

# code generation and instruction scheduling based on 2.0
f90 +DA2.0

# code generation and instruction scheduling based on 1.1
f90 +DAportable
```

+Olibcalls Option

Compiling at optimization level 2 or higher enables the optimization performed by the `+Olibcalls` option. If a program uses an intrinsic routine for which a millicode version exists and the program is compiled with `-O`, `+O2`, or `+O3`, the optimizer will substitute the millicode version.

The default at optimization levels 0 and 1 is still `+Onolibcalls`.

+Oregionsched Option

Starting with this release, `f90` ignores the `+O[no]regionsched` option. Changes to the optimizer have reduced the performance benefit of the optimization enabled by this option. The compiler will recognize this option if specified on the command line, but the option has no effect and will be removed from the compiler at a future release.

STAT= Specifier for ALLOCATE Statement

The values returned by the `STAT=` specifer for the `ALLOCATE` statement have been changed to provide more precise error control. A return vaue of zero has the same meaning: the operation was successful. But the meaning of a nonzero return value (an error status) has been changed according to the kind of error, as follows:

`1`	Error occurred after array was allocated; for example, array was previously allocated.
`2`	Error occurred before array was allocated; for example, dynamic memory allocation failure.
`3`	Errors occurred both before and after allocation. This kind of an error can only occur if the same `ALLOCATE` statement is used to allocate more than one array, and both kinds of errors occur.

2

Using HP Fortran 90

This chapter provides usage information on the following topics:

- Compiling and linking HP Fortran 90 programs
- Parallelizing HP Fortran 90 Programs
- Migrating applications from HP FORTRAN 77 to HP Fortran 90
- Incompatibilities between HP FORTRAN 77 and HP Fortran 90
- Calling C routines from HP Fortran 90 programs
- Writing HP Fortran 90 applications for HP-UX

Compiling and Linking HP Fortran 90 Programs

This section discusses the following topics:

- The `f90` command line
- Filename extensions
- Compiling HP Fortran 90 programs with modules
- Compile-line options
- Compiler directives
- Environment variables
- Libraries used by the compiler
- Diagnostic messages issued by the compiler
- Optimization

f90 Command Line

The command-line syntax for invoking the HP Fortran 90 command (`f90`) for compiling and linking is:

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

where *options* is a space-delimited list of compile-line options and *files* is a space-delimited list of files containing source or object code. *options* and *files* can be interspersed on the command line.

Filenames Accepted by the f90 Command

Files that end in the extensions listed in Table 2-1 are compiled as Fortran 90 source files. For each source file that compiles successfully, the corresponding object code is placed in the current directory in a file whose name is that of the source, with the `.o` extension.

Table 2-1. Extensions of Filenames Compiled by f90

Extension	Meaning
`.f90`	Free-format source file, processed by the compiler
`.F`	Fixed-format source file, processed first by the C preprocessor `cpp`, then by the compiler
`.f`	Fixed-format source file, processed by the compiler
`.i90`	Free-format output from C preprocessor, if source ends in `.f90`
`.i`	Fixed format output from C preprocessor, if source ends in `.F` or `.f`

Only files ending in `.F` are preprocessed by the C preprocessor by default. Use the `+cpp=yes` option to preprocess files that end in `.f90` and `.f`.

The `f90` command accepts but does not compile files with other extensions, passing them to another process. For example, filenames with the `.o` extension are assumed to be object files and are passed to the linker (`ld`); and filenames with the `.s` extension are assumed to be assembly-language source files and are passed to the assembler (`as`). Except for filenames ending in `.s` or any of the extensions listed in Table 2-1, all others are passed to the linker.

Compiling HP Fortran 90 Modules

Files that end in `.mod` are HP Fortran 90 modules that are created and read by the compiler. However, the compiler does not process any `.mod` files that may be specified on the command line.

Note 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.

When compiling a program that defines and uses modules in different source files, `f90` creates a `.mod` file for each module in the source files, in addition to the `.o` files. The compiler must have created any `.mod` files before compiling the files that use the modules. Consider, for example, a program that consists of two files: the first (`file1.f90`) defines the module `module1` and the second (`file2.f90`) uses it. The following command lines compile and link the program correctly:

```
f90 -c file1.f90
f90 -c file2.f90
f90 -o my_program file1.o file2.o
```

The crucial lines are the first two, which must be specified in order. The first one creates two files: `file1.o` and `MODULE1.mod`. The second needs `MODULE1.mod` to compile `file2.f90`.

The same effect can be produced with one command line, so long as the defining file is specified before the using file, as in the following:

```
f90 -o my_program file1.f90 file2.f90
```

All `.mod` files are written to and read from the current directory by default. Use the `+moddir=`*directory* and `-I`*directory* options to specify different directories:

- The `+moddir=`*directory* option causes the compiler to write `.mod` files to *directory*.
- The `-I`*directory* option causes the compiler to search *directory* for `.mod` files to read.

Compile-Line Options

Almost all of the `f90` compile-line options have their counterparts among the `f77` options. A few of the UNIX-type options (for example, `-g` and `-o`) have the same name they had under `f77`. However, most of the `f90` options have been renamed for readability. For the sake of compatibility with HP FORTRAN 77, many of the renamed options also have their `f77` names; for example, to prepare a program for profiling by `gprof`, you can specify either `-G` or `+gprof`. A later section, "f77 Options Supported by f90", discusses the relationship between `f77` options and `f90` options.

For an online list of the `f90` compile-line options, use the `+usage` option, as follows:

```
f90 +usage
```

If you misspell an option name on the `f90` command line, the compiler 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 options 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=
```

`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
```

For detailed information about the options, see the *HP Fortran 90 Programmer's Reference*, Chapter 13. The options are summarized in the *f90*(1) man page.

Commonly Used Options

Table 2-2 lists commonly used HP Fortran 90 options. All but the `-L` and `+save` options have the same name and function as in `f77`. The `-L` option differs in that it uses `fort77` semantics, and the `+save` option is a different name for the `-K` option in `f77`.

Table 2-2. Commonly Used Compile-Line Options

Option	Effect
`-c`	Compile without linking; produce an object (`.o`) file from each source file.
`-g`	Prepare program for debugging (with HP DDE) or line-level performance analysis (with Puma).
`-L`*directory*	Use *directory* as the search path for libraries specified in succeeding `-l` options. This option is also supported by `fort77`, but not `f77`.
`-l`*x*	Link with `lib`*x*`.a` or `lib`*x*`.sl`.
`-O`	Optimize at level 2.
`-o` *outfile*	Name the output file (executable or object file) *outfile*.
`+save`	Save all local variables upon exit from a program unit. This option is useful for porting older programs that may contain uninitialized variables or that require static storage for all variables. `f90` also accepts the `f77` form of this option, `-K`.
`-v`	Compile in verbose mode, reporting progress to `stderr`.

f77 Options Supported by f90

The `f90` command recognizes many of the `f77` options by their old names as well as by their `f90` names; these are listed in Table 2-3. When specified on an `f90` command line, these `f77` options have the same effect as their `f90` replacements. For example, `f90` recognizes either `-G` or `+gprof`; both prepare the program for profiling by `gprof`.

Table 2-3. 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)

Table 2-4 lists `f77` options that have been fully or partially replaced by a renamed `f90` option. Table 2-5 lists `f77` options that are not recognized by the `f90` command and that have no `f90` replacement.

Table 2-4. f77 Options Replaced by f90 Options

f77 Option	f90 Replacement
`-A`	`+langlvl`[1]
`-a`	`+langlvl`[1]
`+autodblpad`	`+autodbl`[1]
`+B`	`+escape`
`-D`	`+dlines`
`+es`	`+extend_source`
`-F`	`+cpp_keep`
`+I[2|4]`	`+autodbl`[1]
`-L`	`+list`
`-onetrip`	`+onetrip`
`+Q`	`+pre_include`
`+s`	`+langlvl`[1]
`+T`	`+fp_exception`[1]
`+ttyunbuf`	`+nottybuf`
`-U`	`+uppercase`
`-u`	`+implicit_none`
`-V`	`+list`[1]

1 Does not fully replace; see *HP Fortran 90 Programmer's Reference*, Chapter 13.

Table 2-5. f77 Options Not Supported by f90

+800	+N
+A	+O4[1]
+A3	-O4[1]
+A8	+Ofailsafe[1]
+apollo	+Oloop_transform[1]
+df[1]	+Osideeffects[1]
+E	+Owhole_program_mode[1]
+e	+P[1]
+I[1]	+pgm[1]
+L8	+R
+LA	+U
-lisam	-y[1]
+mr	-w66

1 Will be supported at a later release.

Compiler Directives

HP Fortran 90 supports the following compiler directives:

- ALIAS
- CHECK_OVERFLOW
- LIST
- OPTIMIZE

The new syntax for specifying directives in free source form is:

!HP *directive*

In fixed source form, the syntax is the same except that the comment character can be *, !, or C, and the comment character must be in the first position.

The use of the comment character in the directive syntax promotes program portability by allowing the directive to be treated as a comment except when the compiler is specifically looking for it.

Compiler directives are fully described in the *HP Fortran 90 Programmer's Reference*, Chapter 14. HP Fortran 90 also supports a number of compatibility directives for controlling optimization; see "Compiler Directives" in Chapter 1.

Environment Variables

This section describes the following HP Fortran 90 environment variables:

- HP_F90OPTS
- TMPDIR
- TTYUNBUF
- NLSPATH

In addition, the MP_NUMBER_OF_THREADS envionment variable is now available for use with parallel executing programs; see "MP_NUMBER_OF_THREADS Environment Variable" in Chapter 1.

HP_F90OPTS

The HP_F90OPTS environment variable contains options and arguments for the compiler. The compiler picks up the value of HP_F90OPTS and places its contents before any arguments on the command line. For example, if HP_F90OPTS has the value -v, the following command line:

```
f90 +list prog.f90
```

is equivalent to

```
f90 -v +list prog.f90
```

The bar (|) character can 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, to set HP_F90OPTS so that -O and -lmylib would always be invoked whenever you compiled and that -O would be recognized first and -lmylib last, you would use the following sh commands:

```
HP_F90OPTS="-O | -lmylib"
export HP_F90OPTS
```

or the csh command:

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

In either case, compiling `prog.f90` with the command line:

```
f90 -v prog.f90
```

is equivalent to:

```
f90 -O -v prog.f90 -lmylib
```

TMPDIR

The `TMPDIR` environment variable specifies a directory for temporary files to be used instead of the default directory `/var/tmp`.

TTYUNBUF

The `TTYUNBUF` environment variable controls tty buffering. To enable tty buffering, set `TTYUNBUF` to zero. To disable tty buffering, set `TTYUNBUF` to a nonzero value.

NLSPATH

The `NLSPATH` environment variable specifies the message catalog to be used for the internationalization of compiler messages. For information about diagnostic messages issued by the compiler, see below, "Diagnostic Messages".

Libraries Searched by f90

The compiler searches the libraries listed in Table 2-6 and Table 2-7 during the link phase to create executable programs; some are searched by default (so indicated in the tables), others by specifying the `+U77`, `-lblas`, and `-lm` options. You can specify other libraries by using the `-L` or `-l` options.

Table 2-6. Libraries Shipped with HP Fortran 90

Library	Contents
`/opt/fortran90/lib/libU77.a`	`libU77` routines
`/opt/fortran90/lib/libblas.a`	BLAS library
`/opt/fortran90/lib/libF90.a`[1]	Fortran 90 intrinsics
`/opt/fortran90/lib/libisamstub.a`[1]	Stub library to satisfy ISAM references
`/opt/fortran90/lib/libisamstubs.a`	Stub library to satisfy ISAM references

1 Searched by default.

Table 2-7. Libraries Shipped with HP-UX Operating System

Library	Contents
`/usr/lib/libc.a`[1]	C runtime library
`/usr/lib/libm.a`	Math routines
`/usr/lib/libcl.a`[1]	Fortran runtime library
`/usr/lib/lib*.sl`[2]	Shareable versions of libraries
`/usr/lib/libcps.sl`[3]	Runtime support for parallel execution
`/usr/lib/libpthread.sl`[3]	Kernel threads library for parallel execution

1 Searched by default.

2 By default, the linker searches for shared versions before archived versions; see *HP Fortran 90 Programmer's Reference*, Chapter 12.

3 Searched by default when `+Oparallel` is specified.

Diagnostic Messages

Errors and warnings are written to standard error. If you use the `+list` option to request a listing, errors and warnings are also written to standard output.

The compiler also lists on `stderr` the names of each source file, procedure, and module as they are encountered.

Optimization

As recommended in the *HP Fortran 90 Programmer's Reference*, Chapter 13, the `+Oall` option generally gives maximum performance. However, some HP Fortran 90 programs may execute faster if they are compiled with the `+O2` and `+Oaggressive` options.

See "Parallelizing HP Fortran 90 Programs" for information about parallel optimization.

Parallelizing HP Fortran 90 Programs

The following sections discuss how to use the `+Oparallel` option and the parallel directives when preparing and compiling HP Fortran 90 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 "+Oparallel Option" in Chapter 1. For information about directives that you can use to control parallelization, see "Controlling Parallelization" in Chapter 1.

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 parallel runtime support.

Performance from 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.

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, 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 in a called routine when determining whether to parallelize the loop. For information about this directive, see "Controlling Checks for Side Effects" in Chapter 1.

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 can assume 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 Dependencies

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 "Controlling Dependence Checks" in Chapter 1.

Migrating to HP Fortran 90

A major feature of HP Fortran 90 is its compatibility with standard-conforming HP FORTRAN 77. Both source files and object files from existing HP FORTRAN 77 applications can be compiled by HP Fortran 90 with comparatively little effort. However, some compile-line options and nonstandard extensions in HP FORTRAN 77 programs may require modification.

To smooth the migration path, HP Fortran 90 includes a number of extensions that are compatible with HP FORTRAN 77. HP Fortran 90 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 90 extensions, see the *HP Fortran 90 Programmer's Reference*, Appendix A.

This section discusses issues and approaches to migrating applications from HP FORTRAN 77 to HP Fortran 90.

Migration Issues

Migration issues fall into four general categories:

- Source code
- Compile-line options
- Object code
- Data files

Source Code Issues

For standard-conforming HP FORTRAN 77 code, migration to HP Fortran 90 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 90. The reason is that HP FORTRAN 77 supports a number of compiler directives and intrinsic functions, some of which are supported by HP Fortran 90, but others of which are either unsupported or have changed. The following paragraphs discuss how to change directives and intrinsics when migrating HP FORTRAN 77 source code to HP Fortran 90.

Note HP FORTRAN 77 accepts (or forgives) a number of common but nonstandard programming practices that HP Fortran 90 does not. These nonstandard practices as well as all known incompatibilities between HP FORTRAN 77 and HP Fortran 90 are listed below, in "Incompatibilities with HP FORTRAN 77".

Directives. HP FORTRAN 77 supports more than seventy directives; of these, only a handful are supported by HP Fortran 90; see above, "Compiler Directives", for the directives that are supported and for the new directive syntax. Note that, except for the `LIST` directive, the HP Fortran 90 directives have more limited functionality than their HP FORTRAN 77 counterparts; see the *HP Fortran 90 Programmer's Reference*, Chapter 14.

Although most of the HP FORTRAN 77 directives are not supported by HP Fortran 90, some of their functionality is available through compile-line options; see Table 2-8.

Table 2-8. HP FORTRAN 77 Directives Supported by f90 Options

HP FORTRAN 77 Directive	HP Fortran 90 Option	Remarks
ANSI	+langlvl=f90	Applies to Fortran 90 instead of FORTRAN 77.
ASSEMBLY	+asm	
AUTODBL DBL	+autodbl[4]	
AUTODBL OFF	+noautodbl	
CONTINUATIONS		Obsolete; the functionality enabled by the directive is now the default.
DEBUG	-g	
IF/ELSE/ENDIF		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	

Table 2-8.
HP FORTRAN 77 Directives Supported by f90 Options (continued)

HP FORTRAN 77 Directive	HP Fortran 90 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 default.
`WARNINGS`	`-w`	

Intrinsic Functions. HP Fortran 90 supports most of the intrinsics that are available in HP FORTRAN 77, and more. In addition, most of these intrinsics are available in HP Fortran 90 without having to activate them with compiler directives or compile-line options (as with HP FORTRAN 77).

With the larger number of available intrinsics in HP Fortran 90, 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, you should be aware that many HP FORTRAN 77 intrinsics accept additional (nonstandard) argument types; HP Fortran 90 is more standard-conforming in this regard.

For information about all of the HP Fortran 90 intrinsics, see the *HP Fortran 90 Programmer's Reference*, Chapter 11.

Compile-Line Option Issues

Compile-line options can become a migration issue in two ways:

- If you compile a program with the HP Fortran 90 compiler and the command line contains an unsupported `f77` option, `f90` will flag the option with an error message.

  Refer to Table 2-3 for a list of the options that are supported under their `f77` names as well as their `f90` names. Table 2-4 lists the `f77` options that have been replaced by `f90` options, and Table 2-5 lists the `f77` options that are not supported by `f90`.

- 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 below, in "Object-Code Issues".

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 are linked to HP Fortran 90 object files.

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. Migration problems can occur if the HP FORTRAN 77 object files represent data in a nonstandard form. For example, HP Fortran 90 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 90 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 90, such as the `$LOGICAL` directive. (See "Compiler Directives" 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 2-5.

If you find object-code incompatibilities, you should clean up 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 90. 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 90.

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 made above, in "Object-Code Issues".

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 90 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 90; see, for example, Table 2-3 and Table 2-8. The only changes that you *must* make to the source are to remove or recode incompatible extensions.

Although the task of migrating an HP FORTRAN 77 program to HP Fortran 90 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 90 compilers, `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 90 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 above, "Object-Code Issues" and "Data-File Issues".

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 90.

Lintfor

The `lintfor` tool can be used on HP FORTRAN 77 code to detect semantic assumptions that may not be valid for HP Fortran 90 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 90. 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 90. It also detects incompatible compile-line options when given an `f77` command line. `fid` reports both source-code and object-code incompatibilities between HP FORTRAN 77 and HP Fortran 90. Furthermore, if `fid` detects an incompatible extension whose functionality is enabled by some other means in HP Fortran 90, 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 90 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 90, 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 Table 2-9.

Table 2-9. I/O Specifiers Not Supported by f90

`ACCESS=`*expr1*[1]	`READONLY=`
`IOSTAT=`	`STATUS=`*expr2*[2]
`KEY=`	`TYPE=`
`NAME=`	

1 where *expr1* is a constant expression other than `DIRECT` or `SEQUENTIAL`.

2 where *expr2* is a constant expression other than `OLD`, `NEW`, `UNKNOWN`, `REPLACE`, or `SCRATCH`.

- 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 above, in "Compiler Directives".
- Compile-line options that are not supported (see Table 2-5) or that have been replaced by `f90` options (see Table 2-4).

`fid`'s list of incompatibilities will be periodically updated. For more information about the `fid` command, see the *fid*(1) man page.

Third-Party Migration Tools and Information

Following is a list of third-party tools and sources of information relevant to migrating programs to Fortran 90. All items except the first are available online.

- Cooper Redwine's *Upgrading to Fortran 90* (Springer, 1996).
- The Fortran FAQ contains some migration-related information, including pointers to tools, books, and Internet resources. Some of the third-party tools in this list are also mentioned or reviewed in the FAQ. FAQ is available on either of the following:

  ```
  URL: ftp://rtfm.mit.edu/pub/usenet/Fortran_FAQ
  news:comp.lang.fortran
  ```

- The USENET group, `news:comp.lang.fortran`, includes discussions relevant to both FORTRAN 77 and Fortran 90.
- World Wide Web Pages: The following sites are of particular interest to programmer's migrating applications to Fortran 90. Following each listing is its URL.
  - The HP Fortran Web Page contains up-to-date information on HP's Fortran products; see:

    ```
    http://www.hp.com/go/hpfortran
    ```

  - Metcalf's Fortran Information contains a long list of implementations, books, course material, and other resources; see:

    ```
    http://www.fortran.com/fortran/metcalf.html
    ```

  - The Fortran 90 resource list contains a list of Fortran Web resources, including FAQs, code repositories, and USENET groups; see:

```
http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/resource.html
```

  - Michael Metcalf's `convert.f90` program converts standard FORTRAN 77 code into Fortran 90. According to the Fortran FAQ, this program also performs updates such as indenting `DO` loops and `IF` blocks, inserting

interface blocks drawn from procedure source code, and changing nonstandard length-specification syntax. See:

```
ftp://jkr.cc.rl.ac.uk/pub/MandR/convert.f90
```

- Robert Moniot's `ftnchek` performs a variety of semantic checks on FORTRAN 77 programs. It is not designed as a FORTRAN 77 syntax checker. It can accept some nonstandard language extensions and provides an extensive set of options for customizing the checks performed. The output is detailed and informative. See:

  ```
  ftp://netlib.org/fortran/ftnchek.tar.gz
  ```

Incompatibilities with HP FORTRAN 77

This list of known incompatibilities includes both source-level and object-code incompatibilities. A subset of these are detected by the HP `fid` tool, as described above, in "Fortran Incompatibilities Detector".

Command-Line Options Not Supported

The HP Fortran 90 compiler does not accept the `f77` compile-line options listed in Table 2-5. In addition, HP Fortran 90 code may not link correctly with HP FORTRAN 77 object files that were compiled with these options; see above, "Object-Code Issues".

Floating-Point Constants

The HP Fortran 90 compiler differs from HP FORTRAN 77 in its handling of floating-point constants. The HP Fortran 90 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 90, 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 90, 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.

Intrinsic Functions

The Fortran 90 standard has introduced new intrinsics that may collide with function names in FORTRAN 77 code. You can resolve such collisions by using the `EXTERNAL` statement.

Also, HP FORTRAN 77 allows intrinsics to accept a wider variety of argument types than HP Fortran 90 does. For example, in HP FORTRAN 77 the `MAX` and `MIN` intrinsics can take arguments of different types, while HP Fortran 90 follows the standard and requires all arguments to be of the same type. The HP Fortran 90 version of the `TIME` intrinsic takes a `CHARACTER*` argument; it will not accept an integer. Other intrinsics are similarly affected.

Procedure Calls and Definitions

When defining a procedure or making a procedure call, HP Fortran 90 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 90 requires `LEN(foo())`.
- Extraneous commas, such as may be used in HP FORTRAN 77 as "placeholder" arguments, are not accepted. The following is acceptable to `f77` but not `f90`:

  ```
  call a (a,)
  ```

  To specify optional arguments in HP Fortran 90, use the `OPTIONAL` statement.
- The `SYSTEM INTRINSIC` directive, by which HP FORTRAN 77 determines interfaces, is not supported by HP Fortran 90.

- 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 90:

- Double precision as the default storage for floating-point constants; see above, "Floating-Point Constants".
- `I` and `J` integer suffixes. To express the HP FORTRAN 77 constant `10I` (or `I*2`) in HP Fortran 90, 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 90, 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
  ```

  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 90. (HP FORTRAN 77 coerces `COMPLEX(16)` entities to `COMPLEX(8)` in order to continue a computation.)

Given the previous declarations, the following is acceptable in HP Fortran 90:

```
foo**REAL(bar, 8)   ! foo**bar
```

See the *HP Fortran 90 Programmer's Reference*, Chapter 11, for information about the `REAL` intrinsic.

Input/Output

Some of the I/O specifiers that you can give to `OPEN` and other I/O statements in HP FORTRAN 77 are not supported in HP Fortran 90; these are listed in Table 2-9. (Also, compare the description of `OPEN` in the *HP Fortran 90 Programmer's Reference*, Chapter 10, with the description of HP FORTRAN 77's `OPEN` statement in the *HP FORTRAN/9000 Programmer's Reference*, Chapter 10.) In general, HP FORTRAN 77 allows more specifiers (and more options to specifiers) than does HP Fortran 90.

In HP FORTRAN 77, namelist-directed output character strings are always quote-delimited; how and whether such strings are delimited in HP Fortran 90 depends on the `DELIM=` specifier. Also, HP FORTRAN 77 allows the `NAMELIST` statement to appear after executable statements; HP Fortran 90 does not.

Directives

Only a small number of the compiler directives from HP FORTRAN 77 are supported under HP Fortran 90; see above, "Compiler Directives", which also gives the new directive syntax. The syntax and functionality of individual directives has also changed; for detailed information, see the *HP Fortran 90 Programmer's Reference*, Chapter 14. All unsupported directives should be deleted or replaced by HP Fortran 90 code that results in the same functionality (see Table 2-8).

Miscellaneous

Following are miscellaneous incompatibilities between HP Fortran 90 and HP FORTRAN 77:

- The syntax and functionality of the HP Fortran 90 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 90. For a full description of the `ON` statement with example programs showing how to use it, refer to the *HP Fortran 90 Programmer's Reference*, Appendix D.
- HP FORTRAN 77 accepts the `{` character as comment syntax; HP Fortran 90 does not.
- HP FORTRAN 77 accepts a `PROGRAM` statement with no name; HP Fortran 90 requires the name.
- HP FORTRAN 77 extends the `PROGRAM` statement to enable access to command-line arguments; HP Fortran 90 does not. For information about how to use intrinsics to access command-line arguments, see below, "Accessing Command-Line Arguments".
- HP FORTRAN 77 supports arrays up to rank 20; HP Fortran 90 supports arrays up to rank 7.
- HP FORTRAN 77 accepts an expression like `+ -A`, but HP Fortran 90 generates a syntax error. Use `+(-A)` instead.
- HP FORTRAN 77 does not print leading zeroes in floating-point numbers; HP Fortran 90 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`).
- HP FORTRAN 77 accepts statement functions that convert arguments; HP Fortran 90 does not.
- In HP FORTRAN 77, integers that overflow (through initialization or constant folding) are replaced with the maximum value for that type. If HP Fortran 90 detects integer overflow, it treats it as an error; if it does not detect it, the overflow value is truncated at runtime.

Calling C Routines from HP Fortran 90

This section describes the following language differences between C and HP Fortran 90 that are relevant to calling C routines from an HP Fortran 90 program unit:

- Data types
- Arrays
- Argument-passing Conventions
- Strings
- Case sensitivity
- File handling

Data Types

Table 2-10 lists the corresponding data types for HP Fortran 90 and C.

Table 2-10. Data-Type Correspondence for Fortran 90 and C

HP Fortran 90	C
`CHARACTER`	`char` (array of)
Hollerith (synonymous with `CHARACTER`)	`char` (array of)
`BYTE`, `LOGICAL(1)`, `INTEGER(1)`	`char`
`LOGICAL(2)`	`short`
`INTEGER(2)`	`short`
`LOGICAL`, `LOGICAL(4)`	`long` or `int`
`INTEGER`, `INTEGER(4)`	`long` or `int`
`INTEGER(8)`	`long long`
`REAL`, `REAL(4)`	`float`
`DOUBLE PRECISION`, `REAL(8)`	`double`
`REAL(16)`	`long double`
`COMPLEX(4)`	`struct`
`DOUBLE COMPLEX`, `COMPLEX(8)`	`struct`
derived type	`struct`

The following sections provide more detailed information about language differences for the following data types:

- Logicals
- Complex numbers
- Derived types

Logicals

C uses integers for logical types. In HP Fortran 90, 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 Fortran, zero is false and any nonzero value is true. HP Fortran 90 sets the value `1` for true.

Complex Numbers

C has no complex numbers, but they are easy to simulate. 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;
```

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.

Arrays

The important difference between arrays in HP Fortran 90 and arrays in C is that Fortran uses a column-major storage representation for its multi-dimensional arrays, whereas C uses row-major ordering. For proper accessing, the order of the subscripts must be reversed.

For example, an array that is declared in C as

```
int my_array[2][3];
```

must be declared in HP Fortran 90 as

```
INTEGER, DIMENSION (3,2) :: my_array
```

Argument-Passing Conventions

The important difference between the argument-passing conventions of C and HP Fortran 90 is that Fortran 90 passes arguments by reference—that is, it passes the address of the argument—whereas C usually passes 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 that are accessed as C functions.

HP Fortran 90 provides two built-in functions, `%VAL` and `%REF`, for use when passing arguments from Fortran to C. These functions override Fortran's argument-passing conventions so that Fortran passes each argument as C expects to receive them, by value (`%VAL`) or by reference (`%REF`).

The `%VAL` and `%REF` built-in functions can also be used with the `$HP$ ALIAS` directive. For detailed information, see the *HP Fortran 90 Programmer's Reference*, Chapter 14. See also the example program in "File Handling".

Strings

Unlike HP Fortran 90, 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 therefore explicitly assign the null character to the final element of the character array, as in the following:

```
CALL csub ('a string'//CHAR(0))
```

For each `CHARACTER*`*n* argument passed to a Fortran subprogram, two items are actually passed as arguments:

1. The address of the argument in memory (that is, a pointer to the argument).
2. The argument's length in bytes. This is a "hidden" 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—at the end of the list.

Here is the HP Fortran 90 code to pass two strings and an integer to a C function:

```
INTEGER :: int1
CHARACTER(LEN=7) :: str1
CHARACTER(LEN=15) :: str2
LOGICAL :: result
   ⋮
result = func(str1, int1, str2)
```

To receive these arguments, the C function must have the following prototype declaration:

```
int func (char *s1, int *i, char *s2, int len1, int len2);
```

Case Sensitivity

Unlike HP Fortran 90, C is case-sensitive. HP Fortran 90 converts all external names to lowercase, and it disregards the case of internal names. For example, the names `foo` and `FOO` are the same in Fortran, but different in C.

If case sensitivity is an issue when calling a C function from an HP Fortran 90 program, you can either compile the Fortran program with the `+uppercase` option, which forces Fortran to use uppercase for external names; or you can use the `$HP$ ALIAS` directive specify the case that Fortran should use when calling an external name.

See *HP Fortran 90 Programmer's Reference* for information about the `+uppercase` option (Chapter 13) and the `$HP$ ALIAS` directive (Chapter 14). See also the next section, "File Handling", for an example of the `$HP$ ALIAS` directive.

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 by Fortran programs 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 below, "Establishing a Connection to the File", for more information about file descriptors. `FSTREAM` returns a C file pointer for a Fortran unit number. The unit number must 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`. The program also uses the `%VAL` and `%REF` built-in functions to force Fortran to pass the arguments as the `write` routine expects to receive them: the first by value, the second by reference, and the third by value.

```
  PROGRAM fnum_test

! Use the ALIAS directive to rename the "write" 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
  DO i = 1, 10
    a(i) =  'x'
  END DO

! 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 "write"), 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
```

See the *HP Fortran 90 Programmer's Reference* for detailed information about the `FNUM` and `FSTREAM` intrinsics (Chapter 11) and the `ALIAS` directive and `%VAL` and `%REF` built-in functions (Chapter 14). For information about the `write` system routine, see the *write*(2) man page.

Writing HP Fortran 90 Applications for HP-UX

This section discusses how to use system resources in an HP Fortran 90 application designed to execute on the HP-UX operating system. These resources include:

- Access to command-line arguments from HP Fortran 90 programs
- HP-UX system calls and standard library routines
- HP-UX file I/O

Accessing Command-Line Arguments

HP FORTRAN 77 extends the `PROGRAM` statement to enable access to command-line arguments. This extension is not available in HP Fortran 90. However, an HP Fortran 90 program can nevertheless access command-line arguments by calling the `IGETARG` and `IARGC` intrinsics.

For example, the following command line invokes the program `fprog` with arguments:

```
fprog arg1 "another arg" 222
```

HP-UX captures the entire command line and makes the following strings available to your program:

```
arg1
another arg
222
```

To access these arguments, your program must call the `IGETARG` and `IARGC` intrinsics. `IGETARG` (available either as a function or as a subroutine) gets a specific command-line argument. `IARGC` returns the number of arguments on the command line.

The following program illustrates how to use both intrinsics:

```
PROGRAM test_igetarg

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
```

If this program is compiled and invoked by the name `a.out` in the following command line:

```
a.out perambulation of a different sort
```

it produces the output:

```
13
perambulation
5
```

For more information about the `IGETARG` and `IARGC` intrinsics, see the *HP Fortran 90 Programmer's Reference*, Chapter 11. You can also use the `GETARG` intrinsic to return command-line arguments. `GETARG` is also available as a `libU77` routine; see the *HP Fortran 90 Programmer's Reference*, Chapter 12.

HP-UX System Calls and Library Routines

System calls provide low-level access to kernel-level resources, such as the `write` system routine. For an example of a program that calls the `write` routine, see above, "File Handling". For information about system calls, refer to *HP-UX Reference*, Section 2.

HP-UX library routines provide many capabilities, such as getting system information and file stream processing. Library routines are discussed in the *HP-UX Reference*, Section 3.

You can access many HP-UX system calls and library routines from Fortran programs using the BSD 3F library, `libU77.a`. For details on accessing routines in this library, see the *HP Fortran 90 Programmer's Reference*, Chapter 12.

Another library provided with Fortran 90 is the Basic Linear Algebra Subroutine (BLAS) library, `libblas.a`. These subroutines perform low-level vector and matrix operations, tuned for maximum performance. For information, see the *HP Fortran 90 Programmer's Reference*, Chapter 12.

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 the *stdio*(3S) man page.

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 90 statement declares a variable for use as a stream pointer in HP Fortran 90:

```
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. (See above, "File Handling", for an example program that uses the `ALIAS` directive. All HP Fortran 90 directives are described in the *HP Fortran 90 Programmer's Reference*, Chapter 14.)

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*, Section 2; see also the online man pages for these routines. For an example that shows how to call the `write` routine, see above, "File Handling".

Establishing a Connection to the 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 and 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 in Fortran

The Fortran intrinsic `FNUM` returns the HP-UX file descriptor for a given logical unit. The example program listed in the section "File Handling" calls the `FNUM` intrinsic. For information about `FNUM`, see the *HP Fortran 90 Programmer's Reference*, Chapter 11.

3

Installation Information

You can install HP Fortran 90 after loading the HP-UX operating system 10.3 or later. HP Fortran 90 requires approximately 46 MB of disk space: 18 MB for the compiler and 28 MB for HP DDE, Blink Link and HP PAK.

To install your software, run the SD-UX `swinstall` command. It will invoke a user interface that leads you through the installation process and gives you information about product size, version numbers, and dependencies.

For more information about installation procedures and related issues, refer to *Managing HP-UX Software with SD-UX* and other README, installation, and upgrade documentation included or described in your HP-UX operating system package.

Note

During the installation, the following `WARNING` and `ERROR` messages may appear in the files `/var/adm/sw/swmodify.log` and `/var/adm/sw/swagent.log`:

```
WARNING: Cannot delete the definition for
         "//opt/langtools/lbin/ucomp.tmp" from
         the fileset "Auxiliary-Opt.LANG-AUX".
         The file does not exist in this fileset.

ERROR:   The selected software was not modified.
         All of the specified file modifications
         are invalid.  See the ERROR and/or WARNING
         messages above.
```

These messages are not valid and should be ignored.

4

Related Documentation

Refer to the following documents for information about the HP Fortran 90 compiler:

- *HP Fortran 90 Programmer's Reference* (B3908-90001).
- *f90*(1) man page, which provides a summary reference to the `f90` compile-line options.
- *fid*(1) man page, which describes the Fortran Incompatibilities Detector (`fid`).
- `http://www.hp.com/go/hpfortran`, which provides current information about the HP Fortran 90 compiler.

For corrections to the documentation, see Chapter 5.

For information about HP's optimizing compilers, see the *HP PA-RISC Compiler Optimization Technology White Paper* (5964-9846E). A PostScript file of this document is available online in:

```
/opt/langtools/newconfig/white_papers/optimize.ps
```

5

Restrictions, Problems, and Fixes

This chapter tells you where to look for information about compiler problems and fixes, and describes important restrictions, known problems (along with their workarounds), and corrections to the documentation.

Locating Information on Problems and Fixes

HP customers on support can find a list of HP Fortran 90 compiler problems and their fixes in the current "Software Status Bulletin" (SSB), referencing the release number (1.1) and one of the following product numbers:

B3906BB—HP Fortran 90 Series 700
B3908BB—HP Fortran 90 Series 800

To display the product number and the release version of your HP Fortran 90 compiler, execute this HP-UX command:

```
what /opt/fortran/bin/f90
```

Any user can access the HP SupportLine database on the World Wide Web, which permits searching for bug descripions and available patches. The URL is:

```
http://us.external.hp.com:80/
```

Restrictions in Version 1.1

- Softbench support is not available.
- PBO and `+O4`

  Profile Based Optimization (PBO) and level 4 optimizations (`+O4`) are not available in this release of HP Fortran 90; see Table 2-5 for other optimization options that are not currently available.

- Debugging Cray-style pointers

  Cray-style pointers establish an implicit connection between the pointer and the pointee, which DDE does not recognize. Therefore, you cannot perform debugger operations on the pointee. For example, given the statement

  ```
  POINTER(P, iarr(nelem))
  ```

  you can successfully print the pointer `P` and dereference it, but not the pointee `iarr`.

  ```
  dde> print iarr
  Variable does not exist in this scope.
  ```

- The `ON` statement

  When compiling at optimization level 2 or 3, the user should be aware that the optimizer makes assumptions about the program that do not take into account the behavior of user-defined procedures called by the `ON ... CALL` statement. Such procedures must therefore be well-behaved in optimized programs. The following restrictions apply when using the `ON` statement in an optimized program:

  - The `ON` procedure must not assume that any variable in the interrupted procedure or in its caller has its current value. (The optimizer may have placed the variable in a register until after the call to the interrupted procedure is complete.)
  - The `ON` procedure must not change the value of any variable in the interrupted procedure or in its caller if the effect of the `ON` procedure is to return program control to the point of interrupt.

  These restrictions do not apply if you compile at optimization level 0 or 1.

- Cray-Style pointers and double-precision values

  When the `+autodbl` option is used with a Cray-style pointer, HP Fortran 90 can have different semantics from those in effect on Cray machines. Cray-style pointers are not expanded to eight-byte entities. The HP Fortran 90 compiler generates a warning when a program using Cray-style pointers is compiled with `+autodbl`.

Known Problems

- The `BTEST` and `RNUM` intrinsics fail when compiled with the `+autodbl[4]` option on the PA8000 architecture at optimization levels 2 and 3.

  Workaround: Recompile at a lower optimization level.

- The compiler incorrectly allows the `+Oaggressive` option to be specified at optimization levels 0 and 1. As documented, `+Oaggressive` is only to be used at optimization levels 2 or higher. You must include `+O`*optlevel* on the same command line with the `+Oaggressive` option, and *optlevel* must be set to 2 or higher.

  Also, if you specify `+Oaggressive` at level 2 (that is, if you also specify `+O2`), you must include the `+Onovectorize` option on the same command line.

  The following command lines summarize the correct and incorrect ways to use `+Oaggressive`:

```
f90 +Oaggressive           # Wrong, do not use +Oaggressive at level 0.
f90 +O0 +Oaggressive       # Wrong, same reason.
f90 +O1 +Oaggressive       # Wrong, do not use +Oaggressive at level 1.
f90 +O2 +Oaggressive       # Wrong, add +Onovectorize
f90 +O2 +Oaggressive +Onovectorize  # OK
f90 +O3 +Oaggressive       # OK
```

- The `ON` statement may cause your program to go into an infinite loop when it traps an interrupt. There is no workaround for this problem.

Corrections to the Documentation

The following sections describe documentation errors.

"OUT OF FREE SPACE" Error

The *HP Fortran 90 Programmer's Reference*, Appendix C, states that the `IOSTAT=` and `ERR=` specifiers return error 913 (`OUT OF FREE SPACE`) when the I/O library attempts to use more memory than is available. However, these specifiers do not catch all instances of error 913, especially those caused by memory allocation failures in the I/O library.

+fp_exception Option

The name of the `+fp_exception` option is misspelled as "+fp_exceptions" in the current version of the *HP Fortran 90 Programmer's Reference*, Chapter 13, and in the *f90*(1) man page. The man page has been corrected for this release. The *HP Fortran 90 Programmer's Reference* will be corrected at the next revision.

Index

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W