Compaq Fortran

User Manual for Tru64 UNIX and Linux Alpha Systems

Order Number: AA–Q66TE–TE

January 2002

This manual provides information about the Compaq Fortran program development and run-time environment on Compaq Tru64 UNIX and Linux Alpha systems.

Revision/Update Information: This manual supersedes the previous version of this manual, order number AA–Q66TD–TE.

Software Version: Compaq Fortran for Tru64 UNIX Systems: Version 5.5 or higher
Compaq Fortran for Linux Alpha Systems: Version 1.2 or higher

Compaq Computer Corporation
Houston, Texas

First Printing, June 1994
Revision, January 2002

ZK6325

This document is available on CD–ROM.

This document was prepared using DECdocument, Version 3.3-1n.

Contents

3 f90 and fort Command-Line Options

4 Using the Ladebug Debugger

5 Performance: Making Programs Run Faster

6 Parallel Compiler Directives and Their Programming Environment

7 Compaq Fortran Input/Output (I/O)

8 Run-Time Errors and Signals

9 Data Types and Representation

12 Compaq Fortran Library Routines

13 Using the Compaq Extended Math Library (CXML)

14 Controlling Floating-Point Exceptions

A Compatibility: Compaq Fortran 77 and Compaq Fortran on Multiple Platforms

B Compaq Fortran Environment Variables

C Compiler Output Listings

D Parallel Library Routines

Index

Examples

Figures

Tables

Preface

This manual describes the Compaq Fortran compiler command, compiler, and run-time environment. This includes how to compile, link, execute, and debug Compaq Fortran programs using the Compaq Tru64™ UNIX operating system and the Linux operating system on Alpha hardware.

This manual does *not* cover running, debugging, and profiling programs that execute in parallel using High Performance Fortran (HPF) features.

Intended Audience

This manual makes the assumptions that:

- You already have a basic understanding of the Fortran 95/90 language. Tutorial Fortran 95/90 language information is widely available in commercially published books (see the online release notes or the Preface of the *Compaq Fortran Language Reference Manual*).
- You are familiar with the operating system shell commands used during program development and a text editor, such as `emacs` or `vi`. Such information is available in your operating system documentation set or commercially published books.
- You have access to the *Compaq Fortran Language Reference Manual*, which describes the Compaq Fortran language.

Structure of This Document

This manual consists of the following chapters and appendixes:

- Chapter 1 introduces the programmer to the Compaq Fortran compiler, its components, and related commands.
- Chapter 2 explains how to compile and link Compaq Fortran source programs, previously compiled module files, object files, and routines written in other supported programming languages, such as C.

- Chapter 3 describes the Compaq Fortran compiler command-line options in detail.
- Chapter 4 describes using the Compaq Ladebug debugger to debug Compaq Fortran nonparallel programs.
- Chapter 5 describes ways to improve Compaq Fortran run-time performance for nonparallel programs, including general software environment recommendations, appropriate compiler command-line options, data alignment, efficiently performing I/O and array operations, other efficient coding techniques, profiling, and optimization.
- Chapter 6 describes how to use parallel compiler directives in Compaq Fortran programs to generate code that executes in parallel.
- Chapter 7 provides information on Compaq Fortran I/O, including statement forms, file organizations, I/O record formats, access modes, logical unit numbers, and efficient use of I/O.
- Chapter 8 lists run-time messages and describes how to control certain types of I/O errors and signal-handling considerations.
- Chapter 9 describes native Compaq Fortran Alpha data types, including their numeric ranges, representation, and floating-point exceptional values. It also discusses the intrinsic data types used with numeric data.
- Chapter 10 describes how to access unformatted files containing numeric little endian and big endian data different than the format used in memory.
- Chapter 11 describes the Compaq Fortran language interface, passing arguments, calling between Compaq Fortran and Compaq Fortran 77, and calling between Compaq Fortran and C.
- Chapter 12 lists and briefly describes the Section 3f interface routines, which include some routines specifically designed for the Compaq Fortran environment on Compaq Tru64 UNIX systems.
- Chapter 13 describes the Compaq Extended Math Library (CXML), including how to call and link CXML routines.
- Chapter 14 explains additional ways to control run-time floating-point exceptions.
- Appendix A provides compatibility information for those porting Compaq Fortran 77 (formerly DEC Fortran) applications and Compaq Fortran OpenVMS™ applications, including an overview of Compaq Fortran 77 and Compaq Fortran extensions supported on the various platforms.

- Appendix B lists the Compaq Fortran environment variables recognized at compile time and run time.
- Appendix C describes the source listing formats supported by Compaq Fortran.
- Appendix D summarizes the library routines available for use with directed parallel decomposition.

The appendix formerly titled "Parallel Compiler Directives Reference Material" is no longer in this manual. The contents of this appendix are in the *Compaq Fortran Language Reference Manual*.

Associated Documents

The following documents may also be useful to Compaq Fortran programmers:

- *Compaq Fortran Language Reference Manual*

  Describes the Compaq Fortran 95/90 source language for reference purposes, including the format and use of statements, intrinsic procedures, and other language elements. Compaq extensions to the Fortran 95 standard are identified by blue-green color in the printed document and HTML versions. It also provides an overview of new Fortran 90 features (not available in FORTRAN-77). Language differences between Compaq Fortran platforms are identified.

- *Compaq Fortran Installation Guide for Tru64 UNIX Systems*

  Explains how to install Compaq Fortran on the Compaq Tru64 UNIX operating system, including prerequisites and requirements.

  To install Compaq Fortran on Linux Alpha systems, refer to the online README file supplied with your kit.

- Compaq Fortran online release notes

  Provide more information on this version of Compaq Fortran, including known problems and a summary of the Compaq Fortran run-time error messages.

  The online release notes are located at:
  *TU*X only* `/usr/lib/cmplrs/fort90/relnotes`
  *L*X only* `/usr/doc/cfal-1.2.`*n*`/README`

  where the value of *n* in this location comes from the list 0, 1, 2, . . .

- Compaq Fortran online reference pages

Describe the Compaq Fortran software components, including f90(1) (Tru64 UNIX systems) or fort(1) (Linux systems), fpr(1), fsplit(1), intro(3f), numerous Fortran library routines listed in intro(3f), and numerous parallel Fortran library routines listed in intro(3hpf).

- Official specification for OpenMP Fortran 1.1 Application Program Interface at:

  http://www.openmp.org/specs/

- Compaq Tru64 UNIX operating system documentation

  The operating system documentation set includes reference pages for operating system components and a programmer's subkit, in which certain documents describe the commands, tools, libraries, and other aspects of the programming environment:

  – For programming information, see the *Compaq Tru64 UNIX Programmer's Guide* and the *Compaq Tru64 UNIX Using Programming Support Tools*.

  – For performance information, see the *Compaq Tru64 UNIX System Tuning and Performance*.

  – For an overview of Compaq Tru64 UNIX documentation, see the *Compaq Tru64 UNIX Reader's Guide*.

- Other layered product documentation

  If you are using a programming-related layered product package from Compaq, consult the appropriate documentation for the layered product package for use of that product.

Platform Labels

A **platform** is a combination of operating system and central processing unit (CPU) that provides a distinct environment in which to use a product (in this case, a language). All the information in this manual applies to both Compaq Tru64 UNIX on Alpha systems and Linux on Alpha systems unless otherwise labeled as shown below:

TU*X	Applies to Tru64 UNIX on Alpha systems
L*X	Applies to Linux on Alpha systems

For example, the shcom_connect library routine in Table 12–2 is labeled *(TU*X only)*, so this routine is only valid for Tru64 UNIX operating systems on Alpha processors.

Sending Compaq Your Comments on This Manual

Compaq welcomes your comments on this or any other Compaq Fortran manual. You can send comments by e-mail to:

`fortran@compaq.com`

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Compaq also welcomes general comments.

Communicating with Compaq

If you have a customer support contract and have comments or questions about Compaq Fortran software, you can contact our Customer Support Center (CSC), preferably using electronic means (such as DSNlink). In the United States, customers can call the CSC at 1-800-354-9000.

You can also send comments, questions and suggestions about the Compaq Fortran product to the following e-mail address: `fortran@compaq.com`. Note that this address is for informational inquiries only and is not a formal support channel.

Compaq Fortran Web Site

The Compaq Fortran home page is located at:

`http://www.compaq.com/fortran`

This site contains information about software patch kits, example programs, and additional product information.

Conventions Used in This Document

This manual uses the conventions listed in Table 1.

Table 1 Conventions Used in This Document

Convention	Meaning
%	The default user prompt is your system name followed by a right angle bracket. This manual uses a percent sign (%) to represent this prompt. The actual user prompt varies with the shell in use.
RETURN	This symbol indicates that you must press the named key on the keyboard.
Ctrl/*x*	This symbol indicates that you must press the Ctrl key while you simultaneously press the key labeled *x*.
% **pwd** /usr/usrc/jones	This manual displays system prompts and responses using a monospaced font. User input is displayed in a bold monospaced font.
`monospaced`	This typeface indicates the name of a command, option, pathname, file name, directory path, or environment variable. This typeface is also used in examples of program code, interactive examples, and other screen displays.
`cat`(1)	A shell command name followed by the number 1 in parentheses refers to a command reference page. Similarly, a routine name followed by the number 2 or 3 in parentheses refers to a system call or library routine reference page. (The number in parentheses indicates the section containing the reference page.) To read online reference pages, use the `man` command. Your operating system documentation also contains reference page descriptions.
new term	Bold type indicates the introduction of a new term in text.
variable	Italic type indicates important information, a complete title of a manual, or variable information, such as user-supplied information in command or option syntax.
UPPERCASE lowercase	The operating system shell differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
{ }	Large braces enclose lists from which you must choose one item. For example: { STATUS DISPOSE DISP }

(continued on next page)

Table 1 (Cont.) Conventions Used in This Document

Convention	Meaning
[ ]	Square brackets enclose items that are optional. For example: BLOCK DATA [name]
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: s[,s] . . .
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
real	This term refers to all floating-point intrinsic data types as a group.
complex	This term refers to all complex floating-point intrinsic data types as a group.
logical	This term refers to logical data types as a group.
integer	This term refers to integer data types as a group.
Compaq Fortran	The term Compaq Fortran (formerly DIGITAL Fortran 90) refers to language information that is common to the Fortran 95/90 standards and any Compaq Fortran extensions.
Fortran	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 95/90, and Compaq Fortran.
Fortran 95/90	This term refers to language information that is common to ANSI/ISO Fortran 95 and ANSI/ISO Fortran 90.
`f90`	This command invokes the Compaq Fortran compiler on Tru64 UNIX Alpha systems while the `fort` command invokes the Compaq Fortran compiler on Linux Alpha systems. This manual frequently uses the `f90` command to indicate invoking the Compaq Fortran on both systems, so replace this command with `fort` if you are working on a Linux Alpha system.
`fort`	This command invokes the Compaq Fortran compiler on Linux Alpha systems. See the previous convention for the `f90` command.

(continued on next page)

Table 1 (Cont.) Conventions Used in This Document

Convention	Meaning
OpenMP	This term refers to OpenMP Fortran as specified in the OpenMP Fortran 1.1 Application Program Interface.
HPF	This term refers to the High Performance Fortran extensions to the Fortran language.

1
Getting Started

This chapter contains the following topics:

- Section 1.1, Compaq Fortran Programming Environment
- Section 1.2, Commands to Create and Run an Executable Program
- Section 1.3, Creating and Running a Program Using a Module and Separate Function
- Section 1.4, f90 or fort Command and Related Software Components
- Section 1.5, Program Development Stages and Tools
- Section 1.6, Compaq Fortran and Standards It Conforms To

1.1 Compaq Fortran Programming Environment

The following aspects of Compaq Fortran are relevant to the compilation environment and should be considered before extensive coding begins:

- To install Compaq Fortran on your Compaq Tru64 UNIX Alpha system, first obtain the UNIX Software Program Library Compact Disc (CD) media. Then perform the installation as described in *Compaq Fortran Installation Guide for Tru64 UNIX Systems*.
- To install Compaq Fortran on your Compaq Linux Alpha system, use the Compaq Fortran CD–ROM supplied with your kit. Perform the installation using the installation letter in your kit.
- *(TU*X only)* Once Compaq Fortran is installed on a Compaq Tru64 UNIX Alpha system, you can:
  - Use the `f90` command to compile and link programs.
  - Use the online `f90`(1) reference page and this manual to provide information about the `f90` command.

- *(L*X only)* Once Compaq Fortran is installed on a Compaq Linux Alpha system, you can:
  - Use the `fort` command to compile and link programs.
  - Use the online `fort`(1) reference page and this manual to provide information about the `fort` command.
- Make sure you have adequate stack size, especially if your programs use large arrays as data. Users may be able to overcome this problem by increasing the per-process data limit, using the `limit` command (C shell) or `ulimit` command (Korn and Bourne and bash *(L*X only)* shells) (see `csh`(1), `ksh`(1), `sh`(1), or `bash`(1)).

  Determine whether the maximum per-process data size is already allocated by checking the value of the maxdsiz parameter in the system configuration file. If necessary, increase its value. Changes to the configuration file do not take effect until the operating system kernel has been rebuilt and the system has been rebooted.

  For example, the following C shell commands check the current stacksize limit and then increase the size to a larger value (if this limit can be increased from your process):

```
% limit stacksize
stacksize       4096 kbytes
% limit stacksize 32676
% limit stacksize
stacksize       32676 kbytes
```

  You can also remove the limitation on stacksize:

```
% limit stacksize unlimited
```

  Similarly, with the Korn and Bourne and bash *(L*X only)* shells, use the `ulimit` command with the `-s` option (see `ksh`(1) and `sh`(1) and `bash`(1)). For example:

```
$ ulimit -s
4096
$ ulimit -s 32676
$ ulimit -s
32676
```

  If you are unable to increase your limits to a value needed by your program, contact your system administrator.
- Make sure you have an adequate process file descriptor limit, especially if your programs use a large number of module files.

During compilation, your application may attempt to use more module files than your descriptor limit allows. In this case, the Compaq Fortran compiler will close a previously opened module file before it opens another to stay within your descriptor limit. This results in slower compilation time. Increasing the descriptor limit may improve compilation time in such cases.

Users can view and usually increase the per-process limit on the number of open files by using the `limit` command (C shell) or `ulimit` command (Korn and Bourne and bash *(L*X only)* shell). (See `csh`(1), `ksh`(1), `sh`(1), or `bash`(1).)

Determine whether the maximum per-process limit is already allocated by checking the value of the appropriate descriptor parameter in the system configuration file. If necessary, increase its value. Changes to the configuration file do not take effect until the operating system kernel has been rebuilt and the system has been rebooted.

For example, the following C shell commands check the current limits and then increase the size to a larger value for cases where this limit can be increased from your process:

```
% limit descriptor
descriptors 100 files
% limit descriptor 4096
% limit descriptor
descriptors 4096 files
```

With the Korn, Bourne, and bash *(L*X only)* shells, you can use the `ulimit` command with the `-n` option (see `ksh`(1), `sh`(1), and `bash`(1). For example:

```
$ ulimit -n
2048
$ ulimit -n 4096
$ ulimit -n
4096
```

- Compaq Fortran supports the use of the environment variable TMPDIR to specify a working directory (instead of `/tmp`) to contain temporary files created during compilation. Several other environment variables can similarly be used during program execution (see Appendix B).

  If you need to set environment variables frequently, consider setting these in your `.login` file or appropriate shell initialization file (`.cshrc` or `.profile`).

- Your source files can be in free or fixed form. The `f90` and `fort` commands recognize certain file suffixes as files containing fixed form or files containing free form. You can also specify an option on the command

line to specify the source form. Recognized file suffixes are described in Section 2.1.1.

A special type of fixed source form is tab form (a Compaq extension). For details about the source forms, see the *Compaq Fortran Language Reference Manual*.

- Each source file to be compiled must contain at least one program unit (main program, external subroutine, external function, module, block data). Consider the following aspects of program development:

  – Modularity and efficiency

    For a large application, using a set of relatively small source files promotes incremental application development.

    When compiling multiple source files into a single object file with a single `f90` (or `fort`) command, certain interprocedure optimizations will occur to minimize run-time execution time (unless you specify a lower level of optimization).

  – Code re-use

    Modules, external subprograms, and included files allow re-use of common code. Code used in multiple places in a program should be placed in a module, external subprogram (function or subroutine), or included file.

    When using modules and external subprograms, there is one copy of the code for a program. When using INCLUDE statements, the code in the specified source file is repeated once for each INCLUDE statement.

    In most cases, using modules or external subprograms makes programs easier to maintain and minimizes program size.

For More Information:

- On modules, see Section 2.1.3.
- On the types of subprograms and using an explicit interface to a subprogram, see Chapter 11.
- On performance considerations, including compiling source programs for optimal run-time performance, see Chapter 5.

1.2 Commands to Create and Run an Executable Program

Example 1–1 shows a short Fortran 95/90 main program using free form source.

Example 1–1 Sample Main Program

```
! File hello.f90

     PROGRAM HELLO_TEST

       print *, 'hello world'
       print *, ' '

     END PROGRAM HELLO_TEST
```

To create and revise your source files, use a text editor, such as `vi` or `emacs`. On Linux systems, the `peco` editor is available. For instance, to use `vi` to edit the file `hello.f90`, type:

```
% vi hello.f90
```

The following command compiles the program named `hello.f90` and automatically uses `ld` to link the main program into an executable program file named `a.out`:

```
% f90 hello.f90
```

The `f90` command (or on Linux systems, the `fort` command) automatically passes a standard default list of Compaq Fortran Run-Time Libraries to the `ld` linker. In this example, because all external routines used by this program reside in these standard libraries, additional libraries or object files are not specified on the `f90` (or `fort`) command line.

If your path definition includes the directory containing `a.out`, you can run the program by simply typing its name:

```
% a.out
```

If the executable image is in your current directory, specify:

```
% ./a.out
```

If the executable image is not in a directory in your path definition and it is not in your current directory, then specify its full path. For example:

```
% /usr/disk5/mcdonald/a.out
```

1.3 Creating and Running a Program Using a Module and Separate Function

Example 1–2 shows a sample Fortran 95/90 main program using free source form that uses a module and an external subprogram.

The function CALC_AVERAGE is contained in a separately created file and depends on the module ARRAY_CALCULATOR for its interface block.

Example 1–2 Sample Main Program that Uses a Module and Separate Function

```
! File: main.f90
! This program calculates the average of five numbers

  PROGRAM MAIN

    USE ARRAY_CALCULATOR                    ❶
    REAL, DIMENSION(5) :: A = 0
    REAL :: AVERAGE

    PRINT *, 'Type five numbers: '
    READ  (*,'(F10.3)') A
    AVERAGE = CALC_AVERAGE(A)               ❷
    PRINT *, 'Average of the five numbers is: ', AVERAGE

  END PROGRAM MAIN
```

❶ The USE statement accesses the module ARRAY_CALCULATOR. This module contains the function declaration for CALC_AVERAGE (use association).

❷ The 5-element array is passed to the function CALC_AVERAGE, which returns the value to the variable AVERAGE for printing.

Example 1–3 shows the module referenced by the main program. This example program shows more Fortran 95/90 features, including an interface block and an assumed-shape array.

Example 1–3 Sample Module

```
! File: array_calc.f90.
! Module containing various calculations on arrays.

  MODULE ARRAY_CALCULATOR
    INTERFACE
      FUNCTION CALC_AVERAGE(D)
        REAL :: CALC_AVERAGE
        REAL, INTENT(IN) :: D(:)
      END FUNCTION CALC_AVERAGE
    END INTERFACE

  ! Other subprogram interfaces...

  END MODULE ARRAY_CALCULATOR
```

Example 1–4 shows the function declaration CALC_AVERAGE referenced by the main program.

Example 1–4 Sample Separate Function Declaration

```
! File: calc_aver.f90.
! External function returning average of array.

  FUNCTION CALC_AVERAGE(D)
    REAL :: CALC_AVERAGE
    REAL, INTENT(IN) :: D(:)
    CALC_AVERAGE = SUM(D) / UBOUND(D, DIM = 1)
  END FUNCTION CALC_AVERAGE
```

1.3.1 Commands to Create the Executable Program

During the early stages of program development, the sample program files in Example 1–2, Example 1–3, and Example 1–4 might be compiled separately and then linked together, using the following commands:

```
% f90 -c array_calc.f90
% f90 -c calc_aver.f90
% f90 -c main.f90
% f90 -o calc main.o array_calc.o calc_aver.o
```

In this sequence of `f90` (or `fort`) commands:

- The `-c` option (used in the first three commands) prevents linking and retains the `.o` files.
- The first command creates the files `array_calculator.mod` and `array_calc.o` (the name in the MODULE statement in Example 1–3 determines the name of module file `array_calculator.mod`). Module files are written into the current working directory.

- The second command creates the file calc_aver.o.
- The third command creates the file main.o and uses the module file array_calculator.mod.
- The last command links all object files into the executable program named calc. To link files, use the f90 command instead of the ld command.

To allow more optimizations to occur (such as the inline expansion of called subprograms), the entire set of three source files can be compiled and linked together with a single f90 command:

```
% f90 -o calc array_calc.f90 calc_aver.f90 main.f90
```

The order in which the file names are specified is significant. This f90 command:

- Compiles the file array_calc.f90, which contains the module definition, and creates its object file and the file array_calculator.mod.
- Compiles the file calc_aver.f90, which contains the external function CALC_AVERAGE.
- Compiles the file main.f90 (main program). The USE statement references the module file array_calculator.mod.
- Uses ld to link the main program and all object files into an executable program file named calc.

1.3.2 Running the Sample Program

If your path definition includes the directory containing calc, you can run the program by simply typing its name:

```
% calc
```

When running the sample program, the PRINT and READ statements in the main program result in the following dialogue between user and program:

```
Type five numbers:
55.5
4.5
3.9
9.0
5.6
Average of the five numbers is:   15.70000
```

1.3.3 Debugging the Sample Program

To debug a program with the Compaq Ladebug Debugger, compile the source files with the `-g` and `-ladebug` options to request additional symbol table information for source line debugging in the object and executable program files. The following `f90` command also uses the `-o` option to name the executable program file `calc_debug`:

```
% f90 -g -ladebug -o calc_debug array_calc.f90 calc_aver.f90 main.f90
```

The Ladebug debugger has a character-cell interface (`ladebug` command) and a windowing interface.

For instance, to use the character-cell interface to debug an executable program named `calc_debug` on a system running Compaq Tru64 UNIX, type the following command:

```
% ladebug calc_debug
```

The Compaq Tru64 UNIX operating system provides the Ladebug debugger and the `dbx` debugger, both of which can be used to debug Compaq Fortran programs. On Linux Alpha systems, the Compaq Fortran CD–ROM includes the Ladebug debugger.

For more information about Ladebug, see the following Web site:

`http://www.compaq.com/products/software/ladebug/`

For more information on running the program within the debugger, see Chapter 4, Using the Ladebug Debugger.

1.4 f90 or fort Command and Related Software Components

Compaq Fortran provides the standard features of a compiler and linker. Compaq Fortran also supports the use of preprocessors and other compilers.

Compiling and linking are usually done by a single `f90` or `fort` command. The `f90` or `fort` command allows you to use:

- A preprocessor, if needed, such as `cpp` or `fpp`.
- The Compaq Fortran compiler.
- The C compiler by means of the `cc` command. (On Linux systems, the `cc` command is a symbolic link to `ccc` or `gcc`.)
- The `ld` linker.

1.4.1 f90 or fort Driver Program

The f90 or fort command invokes a **driver program** that is the actual user interface to the Compaq Fortran compiler. It accepts a list of command-line options and file names, and causes one or more programs (preprocessor, compiler, assembler, or linker, executed in a sequential manner) to process each file.

After the Compaq Fortran compiler processes the appropriate files to create one or more object files, the driver program passes a list of files, certain options, and other information to the cc (or, on Linux systems, the ccc) compiler.

The cc compiler processes relevant non-Fortran files and information (including running the cpp preprocessor) and passes certain information (such as .o object files) to the ld linker. The cc compiler applies cpp to files that it recognizes, such as any file with a .c suffic. If cpp is used, it is executed once for each file.

If any program does not return a normal status, further processing is discontinued and the f90 command displays a message identifying the program (and its returned status, in hexadecimal) before terminating its own execution.

See Figure 2–1, Driver Programs and Software Components.

1.4.2 cpp, fpp, and Other Preprocessors

When you use a preprocessor for Compaq Fortran source files, the output files the preprocessor creates are used as input source files by the Compaq Fortran compiler. Preprocessors include:

- The C preprocessor cpp, which is provided with the C compiler on Tru64 UNIX and Linux operating systems. To run cpp before Fortran 90 source files are compiled, specify the -cpp option on the f90 or fort command line. With fixed form source files, you can alternatively name a file using a file suffix of .F to run cpp. With free form source files, you can use a file suffix of .F90 to run cpp.
- The Fortran preprocessor fpp, which is provided with Compaq Fortran on Tru64 UNIX and Linux operating systems. To run fpp before Fortran 90 source files are compiled, specify the -fpp option on the f90 or fort command line. With fixed form source files, you can alternatively name a file using a file suffix of .F to run fpp. With free form source files, you can use a file suffix of .F90 to run fpp.

- Preprocessors provided with optional products, such as the Compaq KAP Fortran/OpenMP performance preprocessor (see Section 5.1.1). Such preprocessors usually need to be run to create the appropriate source file (or files) *before* you enter the `f90` command line.

1.4.3 Compaq Fortran Compiler

The Compaq Fortran compiler provides the following primary functions:

- Verifying the correctness of Compaq Fortran source statements and displaying any warnings or error messages.
- Generating machine-level object language instructions from the source statements.

  If the `-omp` option was specified on Tru64 UNIX systems, the compiler generates code enabled for parallel execution, based on OpenMP language constructs in the input source files.

  If the `-hpf` option was specified on Tru64 UNIX systems, the compiler generates code enabled for parallel execution, based on HPF language constructs in the input source files.
- Grouping the instructions to generate an object file that can be processed by the linker.

The object file created by the compiler contains information used by the linker, including the following:

- The object file name.

  This is taken from the name specified in the first PROGRAM, SUBROUTINE, FUNCTION, MODULE, or BLOCK DATA statement in the source program. If a program unit does not have any of these statements, the source file name is used, appended with $MAIN (or $DATA for block data subprograms).
- A list of global symbols declared in the object file.

  The linker uses this information when it binds two or more program units together and must resolve references to the same names in the program units. Such global symbols include entry points and common block names.
- A symbol table (if specifically requested by the `-g`, `-g2`, or `-g3` options on the `f90` command line).

  A symbol table lists the names of all external and internal variables within an object file, with definitions of their locations. The table is of primary use in program debugging.

The file name of the Compaq Fortran compiler is `decfort90`, which may appear in certain messages.

1.4.4 Other Compilers

You can compile and link multilanguage programs using a single `f90` command.

The `f90` command recognizes C or Assembler program files by their file suffix characters and passes them to the `cc` driver and compiler for compilation. Before compilation, `cc` applies the `cpp` preprocessor to files that it recognizes, such as any file with a `.c` suffix, and passes appropriate files to other compilers or the assembler.

Certain options passed to `cc` are passed by `cc` to the `ld` linker.

1.4.5 Linker (ld)

When you enter an `f90` command, the `ld` linker is invoked automatically unless a compilation error occurs or you specify the `-c` option on the command line. The linker produces an executable program image with a default name of `a.out`.

The `ld` linker provides such primary functions as:

- Adding information for virtual memory allocation in the executable program
- Resolving symbol references among object files
- Assigning values to relocatable global symbols
- Performing relocation

For more information, see `ld`(1) and your operating system's programmer's guide.

1.5 Program Development Stages and Tools

This manual primarily addresses the program development activities associated with implementation and testing phases. For information about topics usually considered during application design, specification, and maintenance, see your operating system documentation, appropriate reference pages, or appropriate commercially published documentation.

Table 1–1 lists and describes some of the software tools you can use when implementing and testing a program.

Table 1–1 Main Tools for Program Development and Testing

Task or Activity	Tool and Description
Manage source files	Use `rcs` or `sccs` to manage source files. For more information, see the *Compaq Tru64 UNIX Using Programming Support Tools* or the appropriate reference page.
Create and modify source files	Use a text editor, such as `vi` or `emacs`. For more information, see your operating system documentation.
Analyze source code	Use searching commands such as `grep` and `diff`. For more information, see the *Compaq Tru64 UNIX Using Programming Support Tools* and the appropriate reference page.
Build program (compile and link)	You can use the `f90` command to create small programs, perhaps using shell scripts, or use the `make` command to build your application in an automated fashion using a **makefile**. For more information on `f90`, see Chapter 2. For more information on `make`, see the `make`(1) reference page and the *Compaq Tru64 UNIX Using Programming Support Tools*.
Debug and Test program	Use Ladebug (or dbx) to debug your program or run it for general testing. For more information on debugging, see Chapter 4.
Analyze performance	To perform profiling of code, use the `prof` and `pixie` *(TU*X only)* programs. The `f90` command option needed to use `prof` is `-p` (same as `-p1`).
	To perform call graph profiling, use the `gprof` tool. The `f90` command option needed to use `gprof` is `-pg`.
	Related profiling tools *(TU*X only)* include the use of feedback files and `-cord`.
	For more information on profiling Fortran 90 code, see Chapter 5.
Install program *(TU*X only)*	Use `setld` and related commands such as `tar`. For more information, see the *Compaq Tru64 UNIX Using Programming Support Tools*.

To view information about an object file or an object library, use the following shell commands:

- The `file` command shows the type of a file (such as which programming language, whether it is an object library, ASCII file, and so forth).
- The `strings` command *(TU*X only)* shows whether the object (`.o`) file was compiled by Compaq Fortran 77 or Compaq Fortran and, if it was, the version number used.

- The `nm` command shows symbol table information, including the identification field of each object file.
- The `size` command shows the size of the code and data sections.

For more information on these commands, see the appropriate reference page or the operating system's programmer's guide.

To perform other program development functions at various stages of program development:

- Use the `ar` command to:
  - Create an archive object library.
  - Maintain the object modules in the library.
  - List the object modules in the library.
  - Perform other functions.

  Use `ranlib` to add a table of contents to the object library for linking purposes. For more information, see `ar`(1) or the *Compaq Tru64 UNIX Programmer's Guide*.
- Use `f90` or `ld`, not the `ar` command, to create shared libraries on Tru64 UNIX systems. For more information, see Section 2.6 and the *Compaq Tru64 UNIX Programmer's Guide*.
- Use `fort` or `ld`, not the `ar` command, to create shared libraries on Linux systems. For more information, see Section 2.6.
- Use the `strip` command to remove symbolic and other debugging information to minimize image size. For more information, see `strip`(1).
- Use the `fsplit` command to split a multiple routine Fortran file into multiple, individual files. For more information, see `fsplit`(1).

For More Information:

- On the Compaq Tru64 UNIX programming environment, see the *Compaq Tru64 UNIX Programmer's Guide* and the *Compaq Tru64 UNIX Using Programming Support Tools*.

1.6 Compaq Fortran and Standards It Conforms To

Compaq Fortran conforms to American National Standard Fortran 95 (ANSI X3J3/96-007) [1] and American National Standard Fortran 90 (ANSI X3.198-1992) [2], and includes support for the OpenMP Fortran 1.1 Application Program Interface [3] and the High Performance Fortran Language Specification [4].

The ANSI committee X3J3 is currently answering questions of interpretation of Fortran 95 and Fortran 90 language features. Any answers given by the ANSI committee that are related to features implemented in Compaq Fortran may result in changes in future releases of the Compaq Fortran compiler, even if the changes produce incompatibilities with earlier releases of Compaq Fortran.

Compaq Fortran also includes support for programs that conform to the previous Fortran standards (ANSI X3.9-1978 and ANSI X3.0-1966), the International Standards Organization standard ISO 1539-1980 (E), the Federal Information Processing Institute standard FIPS 69-1, and the Military Standard 1753 Language Specification.

Compaq Fortran provides a number of extensions to the Fortran 95/90 standards. Compaq Fortran extensions to the Fortran 95/90 standards are generally provided for compatibility with Compaq Fortran extensions to the ANSI FORTRAN-77 standard and to support the High Performance Fortran (HPF) Language Specification.

When creating new programs that need to be standard-conforming for portability reasons, you should avoid or minimize the use of extensions to the Fortran 95/90 standards. Extensions to the Fortran 95/90 standards are identified visually in the *Compaq Fortran Language Reference Manual*, which defines the Compaq Fortran language.

---

[1] This is the same as International Standards Organization standard ISO/IEC 1539-1:1997 (E).

[2] This is the same as International Standards Organization standard ISO/IEC 1539:1991 (E).

[3] See the specification at `http://www.openmp.org/specs/`.

[4] See High Performance Fortran Language Specification, Version 1.1, Technical Report CRPC-TR-92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, USA.

2

Compiling and Linking Compaq Fortran Programs

This chapter contains the following topics:

- Section 2.1, f90 Command: Files and Options
- Section 2.2, Driver Programs and Passing Options to cc and ld
- Section 2.3, Compiler Limits, Diagnostic Messages, and Error Conditions
- Section 2.4, Compilation Control: Statements and Directives
- Section 2.5, Linking Object Libraries
- Section 2.6, Creating Shared Libraries

Note

To invoke the Compaq Fortran compiler, use:

- `f90` on Tru64 UNIX Alpha systems
- `fort` command on Linux Alpha systems

This chapter uses `f90` to indicate invoking Compaq Fortran on both systems, so replace this command with `fort` if you are working on a Linux Alpha system.

To invoke the Compaq C compiler, use:

- `cc` on Tru64 UNIX Alpha systems
- `ccc` on Linux Alpha systems

This chapter uses `cc` to indicate invoking Compaq C on both systems, so replace this command with `ccc` if you are working on a Linux Alpha system.

For detailed information on f90 and fort command-line options, see Chapter 3.

2.1 f90 Command: Files and Options

You should almost always use the f90 command (or fort command on Linux systems) to invoke both the Compaq Fortran compiler and the ld linker.

To link one or more object files created by the Compaq Fortran compiler, you should use the f90 command (instead of the ld command), because the f90 command automatically references the appropriate Compaq Fortran Run-Time Libraries when it invokes ld.

When you create your source files using a text editor, use file name suffix conventions expected by the f90 command, as described in Section 2.1.1.

2.1.1 File Suffixes and Source Forms

When creating a source file, choose the file name suffix appropriate for the source form (fixed or free). The f90 command recognizes certain source file suffixes as Fortran 95/90 source files.

Table 2–1 shows the recognized Fortran 95/90 source file suffixes.

Table 2–1 File Suffixes Recognized as Fortran 95/90 Source Files

Suffix	Description
.f90	Identifies Fortran 95/90 files in free source form passed to the Compaq Fortran compiler.
.F90	Identifies Fortran 95/90 files in free source form passed to the cpp preprocessor and then compiled by the Compaq Fortran compiler.
.f, .for	Identifies Fortran files in fixed (or tab) source form passed to the Compaq Fortran compiler but not preprocessed by cpp.
.F, .FOR	Identifies Fortran files in fixed (or tab) source form passed to the cpp preprocessor and then compiled by the Compaq Fortran compiler.

Except for .F, .FOR, and .F90 files, preprocessors are not automatically run before Compaq Fortran compilation. To request that the cpp preprocessor be run before compilation, specify the -cpp option.

You can specify the source file form for all Fortran files on an f90 command line by using the -free option or -fixed option.

Table 2–2 shows other file name suffixes.

Table 2–2 Other File Name Suffixes

Suffix	Description
Library and Object Files	
.a	Identifies archive object libraries passed to cc, which are in turn passed to ld. All routines in the specified object library are searched during linking to resolve external references.
.o	Identifies object files passed to ld.
.so	Identifies shared object libraries passed to cc, which are in turn passed to ld. All routines in the specified object library are searched during linking to resolve external references.
Compaq Fortran Module Files	
.mod	Identifies Fortran 95/90 module files created by the Compaq Fortran compiler. You do not create a .mod file directly and cannot specify a .mod file on the f90 command line (see Section 2.1.3).
Other Source Files	
.c	Identifies C language source files passed to the C compiler driver cc or ccc, which performs additional command-line parsing before invoking the C preprocessor (via the command cpp) and the C language compiler.
.i, .i90	Identifies intermediate files passed from cpp to the Compaq Fortran compiler. The .i or .i90 files are usually created by using the f90 options -P or -K (keep intermediate file) and -cpp (invoke cpp). The .i files are assumed to be in fixed source form. The .i90 file are assumed to be in free source form.
.s	Identifies assembler files passed to cc or to ccc.

To specify libraries (in addition to those automatically searched by the f90 command), you can also use an f90 command-line option, such as -l*string*.

Source file suffixes used by other Compaq languages include:

- On Tru64 UNIX and Linux: C++ (.cxx, .cc, .C)
- On Tru64 UNIX only: Ada (.ada), COBOL (.cob), Pascal (.p) and PL/I (.pli)

For More Information:

On source forms and source coding guidelines that allow the same source file to be used with multiple source forms, see the *Compaq Fortran Language Reference Manual*.

2.1.2 Format of the f90 and fort Commands

The f90 command (on Tru64 UNIX systems) has the following general form:

```
f90
[--options [args]]...
filename
[filename]...
[--options [args]]...
```

The fort command (on Linux systems) has the following general form:

```
fort
[--options [args]]...
filename
[filename]...
[--options [args]]...
```

–*options* [*args*]
Indicates either special actions to be performed by the compiler or linker, or special properties of input or output files. For details about command-line options, see Chapter 3.

If you specify the -l*string* option (which indicates libraries to be searched by the linker) or an object library file name, place it after the file names and after other options.

filename
Specifies the source files containing the program units to be compiled and other files to be used by the Compaq Fortran compiler. The file name has a suffix that indicates the type of file used, such as .f90 or .f (see Section 2.1.1).

If you omit the suffix or it is not one of the preceding types recognized by the f90 command, the file is assumed to be an object file and is passed directly to the linker.

An example f90 command line follows:

```
% f90 -v test.f calc.o -lmnd
```

This command specifies the following:

- The -v option displays the compilation and link passes with their arguments and files, including the libraries passed to ld.
- The file test.f is passed to the Compaq Fortran compiler for compilation. The resulting object file is passed to the linker.
- The object file calc.o is passed directly to the linker. The linker links both object files into an executable program.

- The -lmnd option specifies that the object library libmnd should be searched for unresolved global references. Note that -lmnd is placed *after* the file names and other options.

2.1.3 Creating and Using Module Files

Compaq Fortran creates a **module file** for each module declaration and automatically searches for a module file referenced by a USE statement. A module file contains the equivalent of the module source declaration in a post-compiled, binary form.

2.1.3.1 Creating Module Files

When you compile a Compaq Fortran source file that contains module declarations, Compaq Fortran creates a separate file for each module declaration. The name declared in a MODULE statement becomes the base prefix of the file name and is followed by the .mod suffix.

For example, consider compiling a file that contains the following statement:

```
MODULE MOD1
```

The compiler creates a post-compiled module file mod1.mod in the current directory. An object file is also created for the module.

Compiling a source file that contains multiple module declarations will create multiple module files, but only a single object file. If you need a separate object file for each module, place only one module declaration in each file.

If a source file does not contain the main program and you only need to create module files, specify the -c option to prevent linking.

An object file is not needed if there are only INTERFACE or constant (PARAMETER) declarations. It is needed for all other types of declarations including variables.

2.1.3.2 Using Module Files

Once they are created, you can copy module files into an appropriate shared or private directory. You reference a module file specifying the name in a USE statement (use association). For example:

```
USE MOD1
```

When selecting a directory location for a set of module files, consider how your application will be built, including:

- Whether the module files need to be available privately, perhaps for testing purposes (not shared).

- Whether you want the module files to be available to other users on your project or available system-wide (shared).
- Whether test builds and final production builds will use the same directory or different ones.

To locate module files specified in USE statements, the Compaq Fortran compiler searches the following directories:

- The current working directory
- Each directory specified by one or more `-I`*dir* options on the command line.

Suppose you need to compile a main program `proj_main` that contains one or more USE statements. To request that the compiler look for module files in the additional directories `/usr/proj_module/f90` and then `/usr/common/f90` (after looking in the current directory), enter the following command line:

```
% f90 proj_main.f90 -I/usr/proj_module/f90 -I/usr/common/f90
```

If you specify multiple directories, the order of `-I`*dir* options on the `f90` command line determines the directory search order.

You cannot specify a `.mod` file directly on the `f90` command line.

Module nesting depth is unlimited. If you will use many modules in a program, check the process and system descriptor limit (see Section 1.1).

For More Information:

- On specific `f90` command-line options, including `-I`*dir*, see Chapter 3.
- On an example program that uses a module, see Section 1.3.

2.1.4 INCLUDE Statement and Using Include Files

You can create include files with a text editor. If needed, you can copy include files to a shared or private directory.

When selecting a directory location for a set of include files or text libraries, consider how your application is to be built, including:

- Whether the include files need to be available privately, perhaps for testing purposes (not shared).
- Whether you want the files to be available to other users on your project or available system-wide (shared).
- Whether test builds and final production builds will use the same directory or a different one.

Include file names can have any suffix. Use an INCLUDE statement to request that the specified file containing source lines be included by the compiler in place of the INCLUDE statement.

The INCLUDE statement has the following form:

```
INCLUDE 'name'
INCLUDE 'name.typ'
```

You can also include a file with a pathname specified with the following form:

```
INCLUDE '/pathname/name'
INCLUDE '/pathname/name.typ'
```

If you specify the -vms option, you can specify /LIST or /NOLIST after the file name. For example:

```
INCLUDE '/pathname/name/LIST'
INCLUDE 'name.f90/NOLIST'
```

You can also specify the -show include option to request that source lines from included files appear in the listing file (see Section 3.82).

When the INCLUDE Statement Specifies a Pathname

Specifying *pathname* limits the directory searching done for the named file. For example, *pathname* might specify a directory /usr/users/proj for the file named common_proj.f90:

```
INCLUDE 'usr/users/proj/common_proj.f90'
```

If a directory pathname is specified, only the specified directory is searched.

When the INCLUDE Statement Omits a Pathname

When the INCLUDE statement omits a pathname, one or more directories are searched for the specified file name.

To locate include files specified in INCLUDE statements without a pathname, the Compaq Fortran compiler searches directories in the following order:

1. The directory that the source file resides in (-vms option was omitted) or the current process default directory (-vms option was specified)
2. Each directory specified by one or more -I*dir* options.
3. The /usr/include directory (unless the -noinclude option was specified)

Compaq Fortran allows you to use multiple methods to specify which directories are searched for include files:

- You can specify one or more additional directories for the compiler to search by using one or more -I*dir* options.

- You can request that the standard directory /usr/include *not* be searched by specifying the -noinclude option.
- Depending on whether the -vms option was specified, the Compaq Fortran compiler searches either in the current process default directory or in the directory containing the source file that references the include file.

2.1.5 Output Files: Executable, Object, and Temporary

The output produced by the f90 command includes:

- An object file (such as test.o), if you specify the -c option on the command line
- An executable file (such as a.out), if you omit the -c option
- A listing file (such as test.l), if you specify the -V option
- One or more module files (such as datadef.mod), if the source file contains one or more MODULE statements

You control the production of these files by specifying the appropriate options on the f90 command line. Unless you specify the -c option, the compiler generates a single temporary object file (see Section 2.1.5.2), whether you specify one source file or multiple source files separated by blanks. The ld linker is then invoked to link the object file into one executable program file.

If fatal errors are encountered during compilation, or if you specify certain options such as -c, linking does not occur.

2.1.5.1 Naming Output Files

To specify a file name for the executable program file (other than a.out), use the -o *output* option, where *output* specifies the file name. The following command requests a file name of prog1.out for the source file test1.f:

```
% f90 -o prog1.out test1.f
```

If you specify the -c option with the -o *output* option, you rename the object file (not the executable program file). If you specify -c and omit the -o *output* option, the compiler names the object file using the first specified file name (with a .o suffix substituted for the source file suffix).

You can also use the mv command to rename a file.

2.1.5.2 Temporary Files

Temporary files created by the compiler or a preprocessor reside in the /tmp directory. For example, when an f90 command requests that the compiler create an object file and pass it to the linker, the file is created in, and later deleted from, the /tmp directory (unless you specified the -K option).

You can set the environment variable TMPDIR to specify a directory to contain temporary files if /tmp is not acceptable. For performance reasons, use a local disk (rather than a NFS mounted disk) to contain the temporary files.

For information about the commands used to set and unset environment variables, see Appendix B.

To view the file name and directory where each temporary file is created, use the -v option. To create (and retain) object files in your current working directory, use the -c option. Any object files (.o files) that you specify on the f90 command line are retained.

The TMPDIR environment variable is also used during program execution (run-time) to specify which directory to contain any scratch files your program creates.

2.1.6 Using Multiple Input Files: Effect on Output Files

When you specify multiple source files, the following options control the production of output files and also influence whether Compaq Fortran can apply certain levels of optimizations:

- The -c option prevents linking. This option also creates and retains the object file in the current working directory.
- The -K option keeps temporary files.
- The -o *output* option names the output file.

A description of the interaction of these options follows:

- If you omit both the -c and the -K options, the specified Fortran 95/90 source files are compiled together into a single object file and then linked.

  Since all the Fortran 95/90 source files are compiled together into a single object file, full interprocedure optimizations can occur. Any temporary object files created by the compiler are deleted by the linker after it creates the executable program. The default optimization level is -O4 (unless you specify -g2 or -g).
- If you specify the -c option and you want to allow full interprocedure optimizations to occur, you should also specify the -o *output* option.

The combination of the -c and -o *output* options creates a single object file (*output*) from multiple Fortran 95/90 source files, allowing full interprocedure optimizations. The object file can be linked later.

For example, the following command uses both the -c and -o *output* options to allow interprocedure optimization (explicitly requested by -O4):

```
% f90 -c -o out.o -O4 ax.f bx.f cx.f
```

- If you specify the -c option without specifying the -o *output* option, or if you specify the -K option, each source file is compiled into an object file, creating one object file for each input source file specified. This is acceptable if you specified no optimization (-O0) or local optimization (-O1).

  An information message appears when you specify multiple input files and specify an option that creates multiple object files (such as -c without -o *output*) and specify or imply global optimization (-O2, -O3, -O4, or -O5).

- If you specify the -c option, you must link the object file(s) later by using a separate f90 command. This allows incremental compilation of a large application, perhaps by means of a makefile processed by the make command.

  However, keep in mind that either omitting the -c option or using the -c option in combination with the -o *output* option provides the benefit of full interprocedure optimizations for compiling multiple Fortran source files.

When you request a listing file (-V option), a single listing file is created unless you specify the -c option. If you specify the -c option and the -V option, separate listing files are created.

2.1.7 Examples of the f90 and fort Commands

The following examples show the use of the f90 command. On Linux systems, use the fort command instead of the f90 command.

2.1.7.1 Compiling and Linking Multiple Files

The following f90 command compiles the Compaq Fortran free format source files (aaa.f90, bbb.f90, ccc.f90) into a single temporary object file:

```
% f90 -V aaa.f90 bbb.f90 ccc.f90
```

This f90 command invokes the ld linker and passes the temporary object file to ld, which it uses to produce the executable file a.out. The Compaq Fortran compiler (-V option) creates the listing file aaa.l.

The following f90 command compiles all Compaq Fortran fixed-format (or tab-format) source files with file names that end with .f into a temporary object file:

```
% f90 -V *.f
```

The ld linker produces the a.out file. The listing file (produced when the -V option is specified) assumes the name of the first file, aaa.l.

2.1.7.2 Retaining an Object File and Preventing Linking

The following f90 command compiles, but does not link, the free-format source file typedefs_1.f90, which contains a MODULE TYPEDEFS_1 statement:

```
% f90 -c typedefs_1.f90
```

This command creates files typedefs_1.mod and typedefs_1.o. Specifying the -c option retains the object file typedefs_1.o and prevents linking.

2.1.7.3 Compiling Fortran 95/90 and C Source Files and Linking an Object File

The following f90 command compiles the free-format Compaq Fortran main program (myprog.f90). The main program calls a function written in C and references the module TYPEDEFS_1 with a USE TYPEDEFS_1 statement (uses the object file created in the previous example). The C routine named utilityx_ is declared in a file named utilityx.c. All sources files are compiled and the object files are passed to the linker:

```
% f90 myprog.f90 typedefs_1.o utilityx.c
```

This command does the following:

1. Compiles myprog.f90 with the Compaq Fortran compiler. The module file typedefs_1.mod is read from the current directory.
2. The C compiler compiles utilityx.c.
3. The ld linker links all three object files together into the executable program named a.out.

2.1.7.4 Renaming the Output File

The following f90 command compiles the free-format Compaq Fortran source files circle-calc.f90 and sub.f90 together, producing one object file named circle.o:

```
% f90 -c -o circle.o circle-calc.f90 sub.f90
```

The default optimization level (-O4) applies to both source files during compilation and uses the default loop unrolling. Because the -c option is specified, the object file is not passed to the ld linker and is not deleted. In this case, the named output file is the object file.

Like the previous command, the following f90 command compiles multiple source files:

```
% f90 -o circle.out circle-calc.f90 sub.f90
```

Because the -c option was omitted, an executable program named circle.out is created.

2.1.7.5 Specifying an Additional Linker Library

The following f90 command compiles a free-format source file myprog.f90 using default optimization, and passes an additional library for the linker to search:

```
% f90 typedefs_1.o myprog.f90 -lcxml
```

The file is processed at optimization level -O4 and then linked with the object file typedefs_1.o. The -lcxml option instructs the linker to search in the libcxml library for unresolved references (in addition to the standard list of libraries the f90 command passes to the linker).

2.1.7.6 Requesting Additional Optimizations

The following f90 command compiles the free-format Compaq Fortran source files circle-calc.f90 and sub.f90 together using software pipelining optimizations (-O5):

```
% f90 -O5 -unroll 3 circle-calc.f90 sub.f90
```

The loops within the program are unrolled 3 times (-unroll 3). Loop unrolling occurs at optimization level -O3 or above.

2.1.8 Using Listing Files

If you expect your program to have compilation errors, you should request a separate listing file (-V option).

For example, the following command compiles Compaq Fortran source files with file names that end with .f, and ld creates an executable file named a.out:

```
% f90 -V *.f
```

The listing file assumes the name of the first file. If the first file was named aaa.f, the listing file is named aaa.l.

Using a listing file provides such information as the column pointer (1) that indicates the exact part of the line that caused the error (see Section 2.3.2). Especially for large files, consider obtaining a printed copy of the listing file you can reference while editing the source file.

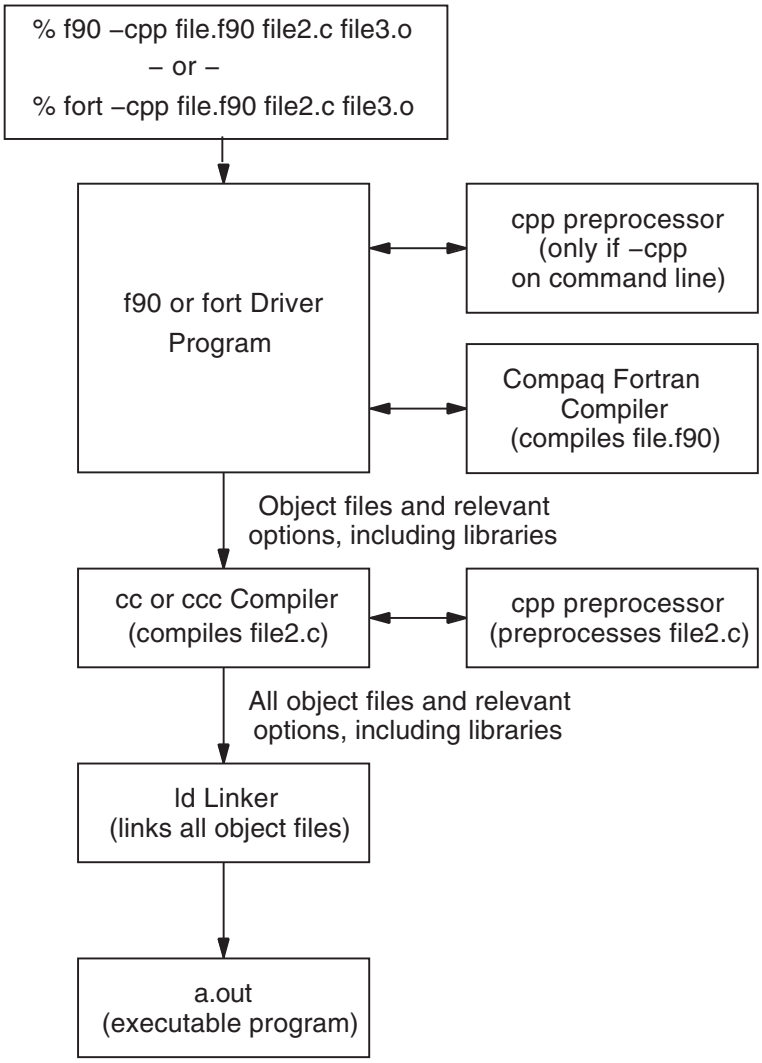
For More Information:

- On calling programs written in other languages, see Chapter 11.
- On options for the `f90` command, see Chapter 3.
- On `f90` command-line options and their categories, see Table 3–1.

2.2 Driver Programs and Passing Options to cc and ld

The `f90` and `fort` driver programs control which software components operate on the files and options specified on the command line and their order of use. See Figure 2–1.

Figure 2–1 Driver Programs and Software Components



ZK–6657A–GE

On Tru64 UNIX Alpha systems, the `f90` driver program passes options and files not intended for the Compaq Fortran compiler to the `cc` driver program.

On Linux Alpha systems, the `fort` driver program passes options and files not intended for the Compaq Fortran compiler to the `cc` driver program. On Linux systems, this is a symbolic link to, generally, `gcc` or `ccc`.

The f90 (or fort) driver program does the following:

1. Examines file name suffix information and groups files and options specified on the command line.
2. Runs the requested preprocessors (if any) on any Fortran files.
3. Runs the Compaq Fortran compiler to process Fortran files.
4. Passes grouped input files, processed source files, and grouped options to the cc driver program in the following order:
   a. All options, except for any -l*string* options, are passed.
   b. All object files are grouped and passed.
   c. All non-Fortran source files (such as a C program with a .c suffix) are grouped and passed.
   d. All archive libraries (.a suffix) are grouped in the same order specified on the f90 command line and passed.
   e. All shared libraries (.so suffix) are grouped in the same order specified on the f90 command line and passed.
   f. All user-specified -l*string* options are grouped in the same order specified on the f90 command line and passed.
   g. All -l*string* options automatically added by the f90 command are grouped with other information and passed.

The cc driver program then does the following:

1. Runs cpp if necessary.
2. Runs the C compiler cc if necessary.
3. Passes library-related information to and runs the ld linker.

Upon return to the command line, the f90 or fort driver program returns one of the following status values:

- 0 (zero)—success
- 1—failure
- 2—subprocess failure (preprocessor, the decfort90 compiler, cc, or ld)
- 3—signal

Because the f90 or fort driver program runs other software components such as the C compiler, error messages may be returned by these other components. For instance, ld may return a message if it cannot resolve a global reference. Using the -v option on the f90 command line can help clarify which component is generating the error.

2.2.1 make Facility

The make facility is often used to automate building large programs. See make(1).

2.2.2 Options Passed to the cc Driver or ld Linker

Certain options are passed directly from the f90 or fort driver program to the cc driver program. These options do not generally apply to compiling Compaq Fortran source files, but might be used to:

- Process C language source files or intermediate files using the f90 or fort command instead of the cc or ccc command
- Pass information to ld.

With the -W*c[c...],arg1[,arg2]...* option, you can pass ld options not otherwise provided by the f90 command directly to ld.

When compiling a program that contains both Compaq Fortran and C language source files, you can usually compile with a single f90 command. Any options that f90 does not recognize are passed to cc, such as the following:

- Compilation environment options, such as -systype *name* and -Y*environment*
- Preprocessor (cpp) options, such as -M
- C language compatibility option, such as -migrate
- Memory location (performance) options, such as -D *num*, -T *num*, and -taso
- Other options listed in cc(1)

Certain options recognized and used by f90 also apply to cc, such as the -O*n* option. If needed, you can compile the C files using the cc command (instead of the f90 command) with the -c option, and then compile the Compaq Fortran files and the (C language) object files using the f90 command.

For more information on the options processed by cc and ccc, see cc(1) and ccc(1) (for most options) or ld(1).

2.3 Compiler Limits, Diagnostic Messages, and Error Conditions

The following sections discuss the compiler limits and error messages from the compiler and linker. Other components can report messages, as described in Section 2.2.

2.3.1 Compiler Limits

Table 2–3 lists the limits to the size and complexity of a single Compaq Fortran program unit and to individual statements contained within it.

Table 2–3 Compiler Limits

Language Element	Limit
Actual number of arguments per CALL or function reference	Limited only by memory constraints.
Arguments in a function reference in a specification expression	255
Array dimensions	7
Array construction nesting	20
Array elements per dimension	9,223,372,036,854,775,807[1] = 2**63-1
Constants; character and Hollerith	7198 characters
Constants; characters read in list-directed I/O	2048 characters
Continuation lines	511
Data and I/O implied DO nesting	7
DO and block IF statement nesting (combined)	128
DO loop index variable	9,223,372,036,854,775,807[1] = 2**63-1
Format group nesting	8
Format statement length	2048 characters
Fortran source line length	fixed form: 72 (or 132 if `-extend_source` is in effect) characters free form: 7200 characters

[1]Also check available process and system virtual memory; see Section 1.1.

(continued on next page)

Table 2–3 (Cont.) Compiler Limits

Language Element	Limit
INCLUDE file nesting	20 levels
Labels in computed or assigned GOTO list	Limited only by memory constraints.
Lexical tokens per statement	20000
Named common blocks	Limited only by memory constraints.
Parentheses nesting in expressions	Limited only by memory constraints.
Structure nesting	30
Symbolic-name length	63 characters

The following are usually limited by the amount of process virtual address space available, as determined by system parameters:

- Amount of data storage
- Size of arrays
- Total size of executable programs

For information on increasing your limits, see Section 1.1.

2.3.2 Compiler Diagnostic Messages and Error Conditions

The Compaq Fortran compiler identifies syntax errors and violations of language rules in the source program. If the compiler finds any such errors, it writes messages to the `stderr` output file and any listing file. If you enter the `f90` command interactively, the messages are displayed on your terminal.

Compiler messages have the following format:

```
f90: severity: filename, line n, message-text
[text-in-error]
--------^
```

The pointer (`---^`) indicates the exact place on the source program line where the error was found. For example, the following error message shows the format and message text in a listing file when an END DO statement was omitted:

```
f90: Severe: echar.f, line 7: Unclosed DO loop or IF block
        DO I=1,5
--------^
```

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

2.3.3 Linker Diagnostic Messages and Error Conditions

If the linker detects any errors while linking object files, it displays messages about their cause and severity. If any errors occur, the linker does not produce an executable program file.

Linker messages are descriptive, and you do not normally need additional information to determine the specific error.

On Tru64 UNIX systems, the general format for `ld` messages is:

```
ld:
message-text
```

On Linux systems, the general format for `ld` messages is:

```
message-text
```

The *message-text* can be on multiple lines and is sometimes accompanied by an `f90` or `fort` error.

Some common errors that occur during linking resemble the following:

- "An object file has compilation errors."

  This error occurs when you attempt to link a file that had warnings or errors during compilation. Although you can usually link compiled files for which the compiler generated messages, you should verify that the files will actually produce the output you expect.

- "The files being linked define more than one transfer address."

  The linker generates a warning if more than one main program has been defined. For example, this can occur when an extra END statement exists in the program. The executable program file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.

- "A reference to a symbol name remains unresolved" or "undefined reference."

  This error occurs when you omit required file or library names from the `f90` or `ld` command and the linker cannot locate the definition for a specified global symbol reference. Use the `-y` option to the `ld` command to determine which files refer to each missing symbol. See `ld`(1) for more information.

If an error occurs when you link files, you may be able to correct it by retyping the command string and specifying the correct routines or libraries (`-l`*string* option, `-L`*dir* option), or by specifying the object library or object files on the command line.

For More Information:

- On specifying object library or object files on the command line and related information on linking, see Section 2.5.
- On linker messages, see your operating system documentation.
- On run-time messages, see Chapter 8.

2.4 Compilation Control: Statements and Directives

In addition to options on the command line, several statements used in the body of a Fortran program and `cpp` directives also influence the compilation process.

- The INCLUDE statement incorporates external source code into your programs during the compilation process.

  If you omit the directory path from the file specification in the INCLUDE statement, the directory searched depends on whether the `-vms` option was specified or omitted. For more information, see Section 2.1.4.
- The USE statement incorporates a post-compiled module file into your programs during the compilation process. For more information, see Section 2.1.3.
- The OPTIONS statement sets compiler options that would otherwise be specified on the `f90` command. The options specified on an OPTIONS statement take precedence over options specified on the `f90` command line if a conflict occurs. Each OPTIONS statement applies only to the program unit in which it appears.
- You can use the general cDEC$ directives to perform tasks during compilation. You can specify the following, for example:
  - The identification field in the object file
  - An alternate name (alias) for external objects, such as subprograms
  - Alternate ways of passing arguments and naming for external objects
  - Common block attributes
  - The `-align` attributes for record structures and common blocks
  - The title and subtitle for listings

- Properties for data objects and procedures

  cDEC$ directives take precedence over the `f90` command-line options.

- A # character in column 1 indicates a `cpp` directive when processing if the `-cpp` option is specified. You can specify that only preprocessing (no compilation) occurs by specifying the `-P` option, perhaps for debugging purposes.

For More Information:

- On Compaq Fortran statements, including INCLUDE, OPTIONS, and USE, see the *Compaq Fortran Language Reference Manual*.
- On cDEC$ directives related to passing arguments and external names, see Section 11.2.
- On `cpp` and its directives, see `cpp`(1).

2.5 Linking Object Libraries

Within a Fortran 95/90 program, references to procedures defined outside your program unit need to be declared as external symbols by using the EXTERNAL statement. (In order for BLOCK DATA statement symbols to be resolved by the linker, the BLOCK DATA symbol must be declared EXTERNAL and not have its data type declared in the source program unit.)

During compilation of multiple source files that will be placed into a single object or executable file, the Compaq Fortran compiler resolves those symbols referenced in one compilation unit and defined in another before linking occurs.

Upon successful compilation of a Compaq Fortran program, the `f90` command specifies certain libraries for `ld` to search for unresolved external symbols in the object file (or files), such as calls to routines in libraries.

Table 2–4 shows the standard `f90` library file names that are searched by `ld`.

Table 2–4 Libraries Automatically Searched When Using the f90 Command

File Name	-l*string* Option Form
libUfor	-lUfor
libfor	-lfor
libFutil	-lFutil
*(TU*X only)* libshpf or libphpf (see text)	-lshpf, -lphpf, or -lphpfp (see text)
libm	-lm
libots	-lots
libc	-lc

*(TU*X only)* If you specify -omp or -mp (requests directed parallel processing), the libots3 library is searched.

Which High Performance Fortran (HPF) library is searched *(TU*X only)* depends on which command-line option is specified:

- If you specify -hpf, libphpf is searched.
- If you omit -hpf, libshpf is searched.

The Compaq Fortran kit provides both shared and archive libraries. For a complete list of Compaq Fortran Run-Time Library files, see f90(1).

2.5.1 Specifying Additional Object Libraries

You can also specify additional object libraries on the command line by using certain options or by providing the file name of the library. These object libraries are also searched by ld for unresolved external references.

When f90 specifies certain libraries to ld, it provides a standard list of f90 library file names to ld. The ld linker tries to locate each of these library file names in a standard list of library directories; ld attempts to locate each object library file name first in one directory, then in the second, and then in the third directory on its search list of directories.

To display a list of the compilers invoked, files processed, and libraries accessed during linking, specify the -v option on the f90 command line.

In addition to an object file created by the compiler, any linker options and object files specified on the f90 command are also passed to the ld linker. The linker loads object files according to the order in which they are specified on the command line. Because of this, you must specify object libraries *after* all source and object files on the f90 command line.

For more details on the interaction of the f90 command with other components, see Section 2.2.

To help you identify undefined references to routines or other symbols in an object file, consider using the nm command. For instance, the following nm command filtered by the grep command lists all undefined (U) symbols:

```
% nm -o ex.o | grep U
```

If the symbol is undefined, a "U" appears in the column before the symbol name. Any symbols with a U in their names are also displayed by this use of grep.

You can control the libraries to be searched with these methods:

- To specify additional object library file names for ld to search, use the -l*string* option at the end of the f90 command line. Each occurrence of the -l*string* option specifies an additional file name that is added to the list of object libraries for ld to locate.
- In addition to the standard directories in which ld tries to locate the library file names, you can use the -L*dir* option to specify another directory.

  Unlike the -l*string* option that adds an object library *file name* for ld to search, the -L*dir* option adds an additional *directory* in which ld will look for library file names.

  The standard ld directories are searched after the directories specified by the -L*dir* option.

  The following example specifies the additional object library path /usr/lib/mytest:

  ```
  % f90  simtest.f -L/usr/lib/mytest
  ```

  For a list of the standard ld directories searched, see Section 3.58.
- You can specify the pathname and file name of an object library as you would specify any file. Specifying each object library that resides in special directories in this manner is an alternative to specifying the library by using the -l*string* or -L*dir* option. Specifying the pathname and file name of an object library can reduce the amount of searching the linker must do to locate all the needed object files.

  For instance, instead of specifying the options -L/usr/jones and -lfft, this example specifies the directory path and file name of the library file:

  ```
  % f90 main.o more.o rest.o /usr/jones/libfft.a
  ```

In certain cases, you may need to specify the pathname and file name instead of using the -l*string* or -L*dir* options for the linker to resolve global symbols with shared libraries.

- You can indicate that ld should not search its list of standard directories at all by specifying the -L option. When you do so, you must specify all libraries on the f90 command line in some form, including the directory for f90 standard libraries.

  To specify all libraries, you might use the -L option in combination with the -L*dir* option on the same f90 command line.

2.5.2 Specifying Types of Object Libraries

External references found in an **archive library** result in the referenced routine being included in the resulting executable program file at link time.

External references found in a **shared object library** result in a special link to that library being included in the resulting executable program file, instead of the actual routine itself. When you run the program, this link gets resolved by either using the shared library in memory (if it already exists) or loading it into memory from disk.

Certain f90 options influence whether ld searches for an archive (.a) or shared object (.so) library on the standard list of f90 libraries, as well as any additional libraries specified by using the -l*string* or -L*dir* options:

- The -call_shared option is the default. It indicates that .so files are searched before .a files. As ld attempts to resolve external symbols, it looks at the shared library before the corresponding archive library. For instance, /usr/shlib/libc.so is searched before /usr/lib/libc.a.

  References to symbols found in .so libraries are dynamically loaded into memory at run time.

  References to symbols found in .a libraries are loaded into the executable program file at link time.

- The -non_shared option indicates that *only* .a files are searched, so the object file created contains static references to external routines. These static references are loaded into the executable program at link time, not at run time. Corresponding .so files are not searched.

  The following example requests that the standard f90 .a files be searched instead of the corresponding .so files:

  ```
  % f90 -non_shared main.f rest.o
  ```

- The -shared option requests the creation of an executable program to be included in a shared library.

If you specify the -c option to inhibit linking, an object file (.o file) is created that can subsequently be processed by ld to create a shared library.

If you omit the -c option, the f90 command creates a shared library (.so file).

In either case, use the -o option to name the resulting object file or shared library with the correct file name and suffix.

For more information about creating a shared library using either the f90 command or the ld command, see Section 2.6, Creating Shared Libraries.

2.5.3 Specifying Shared Object Libraries

When you link your program with a shared library, all symbols must be referenced before ld searches the shared library. You should always specify libraries at the end of the f90 command line, after all file names. Unless you specify the -non_shared option, shared libraries will be searched before the corresponding archive libraries.

For instance, the following command generates an error if the file rest.o references routines in the library libX:

```
% f90 -call_shared test.f -lX rest.o
```

The correct command specifies the library at the end of the line, as follows:

```
% f90 -call_shared test.f rest.o -lX
```

Link errors can occur with symbols that are defined twice, as when both an archive and a shared object library are specified on the same command line. In general, specify any archive libraries after the last file name, followed by any shared libraries at the end of the command line.

Before you reference a shared library at run time, it must be installed. See Section 2.6, Creating Shared Libraries.

2.6 Creating Shared Libraries

To create a shared library from a Fortran source file, process the files using the f90 command:

- You must specify the -shared option to create the .so file.
- You can specify the -o *output* option to name the output file.
- If you omit the -c option, you will create a shared library (.so file) directly from the f90 command line in a single step.

If you also omit the -o *output* option, the file name of the first Fortran file on the command line is used to create the file name of the .so file. You can specify additional options associated with shared library creation.

- If you specify the -c option, you will create an object file (.o file) that you can name with the -o option. To create a shared library, process the .o file with ld, specifying certain options associated with shared library creation.

You can specify multiple source and object files when creating a shared library by using the f90 command.

2.6.1 Creating a Shared Library with a Single f90 Command

You can create a shared library (.so) file with a single f90 command:

```
% f90 -shared  octagon.f90
```

The -shared option is required to create a shared library. The name of the source file is octagon.f90. You can specify multiple source files and object files.

The -o option was omitted, so the name of the shared library file is octagon.so.

2.6.2 Creating a Shared Library with f90 and ld Commands

You first must create the .o file, such as octagon.o in the following example:

```
% f90 -O3 -c octagon.f90
```

The file octagon.o is then used as input to the ld command to create the shared library named octagon.so:

```
% ld -shared -no_archive octagon.o \
          -lUfor -lfor -lFutil -lm -lots -lc
```

Note that the -no_archive option is available only on Tru64 UNIX systems.

- The -shared option is required to create a shared library.
- The -no_archive option *(TU*X only)* indicates that ld should not search archive libraries to resolve external names (only shared libraries).
- The name of the object file is octagon.o. You can specify multiple object (.o) files.
- The -lUfor and subsequent options are the standard list of libraries that the f90 command would have otherwise passed to ld. When you create a shared library, all symbols must be resolved. For more information about the standard list of libraries used by Compaq Fortran, see Section 2.5.

2.6.3 Choosing How to Create a Shared Library

Consider the following when deciding whether to use a single `f90` command (`-c` omitted) or both the `f90` (`-c` present) and `ld` commands to create a shared library:

- Certain `ld` options may not be available from the `f90` command line. If you need to use those options, use the two-command method (specify `f90 -c` and subsequently use `ld`). Such options include `-check_registry` and `-update_registry` (see `ld`(1)) that are available only on Tru64 UNIX systems.
- If you use a single `f90` command with `-shared` and omit `-c`, you do not need to specify the standard list of `f90` libraries by using the `-l`*string* option.

In addition to the options shown in Section 2.6.1 and Section 2.6.2, certain other `ld` options may be needed. For instance, to optimize shared library startup, use the `-update_registry` and `-check_registry` options *(TU*X only)*, which preassign a starting address in virtual memory to a shared library using the file `/usr/shlib/so_locations`.

For More Information:

- On the relevant `ld` options, see the `ld`(1) reference page.
- On the standard list of libraries used by Compaq Fortran, see Section 2.5.

2.6.4 Shared Library Restrictions

When creating a shared library with `ld`, be aware of the following restrictions:

- Shared libraries must not be linked with archive libraries.

  When creating a shared library, you can only depend on other shared libraries for resolving external references. If you need to reference a routine that currently resides in an archive library, either put that routine in a separate shared library or include it in the shared library being created. You can specify multiple object (`.o`) files when creating a shared library.

  To put a routine in a separate shared library, obtain the source or object file for that routine, recompile if necessary, and create a separate shared library. You can specify an object file when recompiling with the `f90` command or when creating the shared library with the `ld` command.

  To include a routine in the shared library being created, put the routine (source or object file) with other source files that make up the shared library and recompile if necessary.

Then create the shared library, making sure that you specify the file containing that routine either during recompilation or when creating the shared library. You can specify an object file when recompiling with the `f90` command or when creating the shared library with the `ld` command.

- When creating shared libraries, all symbols must be defined (resolved).

  Because all symbols must be defined to `ld` when you create a shared library, you must specify the shared libraries on the `ld` command line, including all standard Compaq Fortran libraries. The list of standard Compaq Fortran libraries might be specified by using the `-l`*string* option, as in the previous example in this section.

For other restrictions imposed by the operating system, see the *Compaq Tru64 UNIX Programmer's Guide*.

2.6.5 Installing Shared Libraries *(TU*X only)*

Once the shared library is created, it must be installed for private or system-wide use before you run a program that refers to it:

- To install a *private* shared library (when you are testing, for example), set the environment variable LD_LIBRARY_PATH, as described in `loader`(5).
- To install a *system-wide* shared library, place the shared library file in one of the standard directory paths used by `ld` (see `loader`(5) or Section 3.58).

For complete information on installing shared libraries, see the *Compaq Tru64 UNIX Programmer's Guide*.

3

f90 and fort Command-Line Options

This chapter describes `f90` and `fort` command-line options in detail.

Note

To invoke the Compaq Fortran compiler, use:

- `f90` on Tru64 UNIX Alpha systems
- `fort` command on Linux Alpha systems

This chapter uses `f90` to indicate invoking Compaq Fortran on both systems, so replace this command with `fort` if you are working on a Linux Alpha system.

To invoke the Compaq C compiler, use:

- `cc` on Tru64 UNIX Alpha systems
- `ccc` on Linux Alpha systems

This chapter uses `cc` to indicate invoking Compaq C on both systems, so replace this command with `ccc` if you are working on a Linux Alpha system.

3.1 Overview of Command-Line Options

Options to the `f90` command affect how the compiler processes a file in conjunction with the file name suffix information described in Section 2.1.1. The simplest form of the `f90` command is often sufficient.

You can override some options specified on the command line by using the OPTIONS statement in your Fortran source program. The options specified by the OPTIONS statement affect only the program unit where the statement occurs.

If you compile parts of your program by using multiple `f90` commands, options that affect the execution of the program should be used consistently for all compilations, especially if data is shared or passed between procedures. For example:

- The same data alignment needs to be used for data passed or shared by module definition (such as user-defined structures) or common block. Use the same version of the `-align` option for all compilations (see Section 3.3).
- The program might contain INTEGER, LOGICAL, REAL, COMPLEX, or DOUBLE PRECISION declarations without a kind parameter or size specifier that is passed or shared by module definition or common block. You must consistently use the options that control the size of such numeric data declarations. For information about these options, see:
  - Section 3.53 for INTEGER and LOGICAL declarations
  - Section 3.78 for REAL and COMPLEX declarations
  - Section 3.34 for DOUBLE PRECISION declarations
- On Tru64 UNIX systems, you can specify the `-hpf` *num* option (to request code generation for parallel HPF execution) for multiple compilation units.

Some options consist of two words separated by a space, while others may have words joined by an underscore (_). Most options can be abbreviated. For example, you can abbreviate `-check output_conversion` to `-check out` (usually four characters or more).

3.2 f90 and fort Command Categories and Options

Table 3–1 lists categories of command-line options, the `f90` and `fort` command-line options, and the sections in which they are described in detail.

Table 3–1 f90 and fort Command Categories and Options

Category	Option Name and Section in this Manual
Assembler File	`-S` (see Section 3.81)
Data Size	`-i2`, `-i4`, `-i8`, `-integer_size` *num* (see Section 3.53) `-r8`, `-r16`, `-real_size` *num*, (see Section 3.78) `-double_size` *num*, (see Section 3.34) For a summary of available data types, see Chapter 9.

(continued on next page)

Table 3–1 (Cont.) f90 and fort Command Categories and Options

Category	Option Name and Section in this Manual
Compaq FUSE Cross-Reference File	`-fuse_xref`, (see Section 3.47) *(TU*X only)*
Debugging and Symbol Table	`-d_lines` (see Section 3.33) `-g0`, `-g1`, `-g2`, `-g`, `-g3`, and `-ladebug` (see Section 3.48) Also see Chapter 4.
Data Accuracy and Floating-Point Exceptions	`-assume noaccuracy_sensitive` (see Section 3.12) `-check nopower` (see Section 3.25) `-double_size` *num* (see Section 3.34) `-fast` (see Section 3.40) `-fpe`*n* (see Section 3.44) `-fpconstant` (see Section 3.43) `-fp_reorder` (see Section 3.12) `-fprm` *keyword* (see Section 3.46) `-intconstant` (see Section 3.55) `-math_library` *keyword* (see Section 3.61) `-r8`, `-r16`, `-real_size` *num* (see Section 3.78) `-speculate` *keyword* *(TU*X only)* (see Section 3.84) `-synchronous_exceptions` (see Section 3.86)
Language and Run-Time Compatibility	`-1`, `-66`, `-f66`, `-onetrip` (see Section 3.37) `-altparam` (see Section 3.66) `-assume dummy_aliases` (see Section 3.9) `-assume noprotect_constants` (see Section 3.13) `-f77rtl` (see Section 3.39) `-mixed_str_len_arg` (see Section 3.62) `-std, std95` (see Section 3.85) `-vms` (see Section 3.98)
Linking, Loading, and Output File Naming	`-assume nounderscore` (see Section 3.15) `-assume no2underscores` (see Section 3.16) `-c` (see Section 3.19) `-call_shared`, `-non_shared`, `-shared` (see Section 3.20) `-L` (see Section 3.57) `-L`*dir* (see Section 3.58) `-l`*string* (see Section 3.59) `-names` *keyword* (see Section 3.65) `-o` *output* (see Section 3.71) `-U` (see Section 3.91) `-v` (see Section 3.96) `-Wl,`*-xxx* (see Section 3.99) Also see Section 2.5.

(continued on next page)

Table 3–1 (Cont.) f90 and fort Command Categories and Options

Category	Option Name and Section in this Manual
Listing File	`-show code`, `-show include`, `-machine_code` (see Section 3.82) `-source_listing` (see Section 3.83) `-V` (see Section 3.95) `-vms` (applies to INCLUDE statements with /LIST or /NOLIST) (see Section 3.98) Also see Appendix C.
Module and Include File Directory Searching	`-I`*dir*, `-noinclude` (see Section 3.52) `-module` *directory* (see Section 3.63), Also see Section 2.1.3 (module files) and Section 2.1.4 (include files).
Miscellaneous	`-nohpf_main` *(TU*X only)* (see Section 3.68) `-norun` (see Section 3.70) `-version` (see Section 3.97) `-what` (see Section 3.97)
Nonnative Unformatted Data File Conversion and Use	`-assume byterecl` (see Section 3.7) `-convert` *keyword* (see Section 3.30) Also see Chapter 10.
Parallel HPF Execution Performance *(TU*X only)*	`-hpf`, `-hpf` *n*, `-assume nozsize`, `-nearest_neighbor` *num*, `-nonearest_neighbor`, `-show hpf` (see Section 3.50)
Parallel Directives Execution *(TU*X only)*	`-mp` (see Section 3.64) `-omp` (see Section 3.74) `-assume cc_omp` (see Section 3.8)

(continued on next page)

Table 3–1 (Cont.) f90 and fort Command Categories and Options

Category	Option Name and Section in this Manual
Performance Optimization (Nonparallel and Parallel)	-align *keyword* (see Section 3.3) -annotations *keyword* (see Section 3.4) -arch *keyword* (see Section 3.5) -assume noaccuracy_sensitive (see Section 3.12) -assume dummy_aliases (see Section 3.9) -fast (see Section 3.40) -fpe*n* (see Section 3.44) -fp_reorder (-assume noaccuracy_sensitive) (see Section 3.12) -fprm *keyword* (see Section 3.46) -inline *type*, -noinline (see Section 3.54) -math_library *keyword* (see Section 3.61) -om (see Section 3.73) -O0, -O1, -O2, -O3, -O4, -O, -O5 (see Section 3.72) -pipeline (see Section 3.76) -speculate *keyword* *(TU*X only)* (see Section 3.84) -synchronous_exceptions (see Section 3.86) -transform_loops (see Section 3.89) -tune *keyword* (see Section 3.90) -unroll *num* (see Section 3.94) Also see Section 5.1.2.
Preprocessor	-cpp, -D*name*, -fpp (see Section 3.45), -I, -I*dir*, -K, -M, -P, -U*name* (see Section 3.31), -Wp,-*xxx* (see Section 3.31.3)
Profiling, Feedback Files, and cord	-p0, -p1 or -p, -pg (see Section 3.77) -gen_feedback, -feedback *file*, -cord *(TU*X only)* (see Section 3.41)
Record Output	-assume buffered_io (see Section 3.6) -ccdefault *keyword* (see Section 3.21)
Recursion and Data Storage	-automatic, -static (see Section 3.18) -recursive (see Section 3.79)

(continued on next page)

Table 3–1 (Cont.) f90 and fort Command Categories and Options

Category	Option Name and Section in this Manual
Run-Time Messages and Checking	`-check arg_temp_created`, (see Section 3.22) `-C`, `-check bounds` (see Section 3.23) `-check noformat` (see Section 3.24) `-check nooutput_conversion` (see Section 3.27) `-check overflow` (see Section 3.28) `-check nopower` (see Section 3.25) `-check underflow` *(TU*X only)* (see Section 3.29) `-fpe`*n* (see Section 3.44) `-synchronous_exceptions` (see Section 3.86) Also see the options related to "Floating-point exceptions, rounding, and accuracy" in this table.
Shared Access to Data	`-granularity` *keyword*, (see Section 3.49)
Source Form and Column Use	`-d_lines` (see Section 3.33) `-extend_source` (see Section 3.36) `-fixed`, `-free` (see Section 3.42) `-pad_source` (see Section 3.75)
Threaded Applications	`-granularity` *keyword*, (see Section 3.49) `-pthread` *(TU*X only)* (see Section 3.88) `-reentrancy` *keyword* (see Section 3.80) `-threads` *(TU*X only)* (see Section 3.88)
VMS Run Time	`-vms` (see Section 3.98)
Warning Messages at Compile Time	`-error_limit` *num*, `-noerror_limit` (see Section 3.35) `-v` (see Section 3.96) `-warn` *keyword* `-w1` (see Section 3.100) `-warning_severity` (see Section 3.101)

3.3 -align *keyword* — Data Alignment

Use the `-align` *keyword* options to control the alignment of fields associated with common blocks, derived-type structures, and record structures.

If you omit the `-align` *keyword*, `-fast`, and `-vms` options:

- Individual data items (not part of a common block or other structure) are naturally aligned.
- Fields in derived-type (user-defined) structures (where the SEQUENCE statement is omitted) are naturally aligned.
- Fields in Compaq Fortran record structures are naturally aligned.

- Data items in common blocks *are not* naturally aligned, unless data declaration order has been planned and checked to ensure that all data items *are* naturally aligned.

Although Compaq Fortran always aligns local data items on natural boundaries, certain data declaration statements and unaligned arguments can force unaligned data.

Note

For optimal performance on Alpha systems, make sure your data is aligned naturally (see Section 5.4).

The compiler issues messages when it detects unaligned data (unless you specify `-warn noalignments`). For information about the causes of unaligned data and detection at run time, see Section 5.4, Data Alignment Considerations.

If necessary, you can specify either `-align records` or `-align norecords` on the same command line as one of the following: `-align nocommons`, `-align commons`, or `-align dcommons`. For example:

```
% f90 -align dcommons -align norecords main.f90
```

If you specify `-fast` and omit the `-align` *keyword* options, the compiler uses:
`-align dcommons -align records -align sequence`

If you specify `-vms` and omit the `-fast` and `-align` *keyword* options, the compiler uses:
`-align commons -align norecords -align nosequence`

If no `-align` option is specified, the default is `-align records`, which is the same as specifying `-align rec8byte`.

The following options apply:

-align all
Tells the compiler to add padding bytes wherever possible to obtain the natural alignment of data items in common blocks and structures.

`-align all` essentially turns on these options: `-align nocommons`, `-align dcommons`, `-align records`, `-align sequence`, `-align norec1byte`, `-align norec2byte`, `-align norec4byte`, and `-align norec8byte`.

-align commons
Aligns the data items of all common blocks on natural boundaries up to *4 bytes*. The default is `-align nocommons`.

-align dcommons
Aligns the data items of all common blocks on natural boundaries up to 8 bytes instead of the default byte boundary.

The default is `-align nodcommons`.

If your command line includes the `-std`, `-std90`, or `-std95` options, then the compiler ignores `-align dcommons`. See Section 3.85.

-align none
Tells the compiler not to add padding bytes anywhere in common blocks or structures.

`-align none` essentially turns on these options: `-align nocommons`, `-align nodcommons`, `-align norecords`, `-align nosequence`, `-align norec1byte`, `-align norec2byte`, `-align norec4byte`, and `-align norec8byte`.

-align rec*N*byte
Aligns fields of records and components of derived types on the smaller of the size boundary specified (*N* can be 1, 2, 4, or 8) or the boundary that will naturally align them.

Specifying `-align rec`*N*`byte` does not affect whether common blocks are naturally aligned or packed.

-align norecords
Aligns fields witin record structures and components of derived types on the next available byte boundary instead of the default natural boundaries.

The default is `-align records`, which requests alignment of the fields of all derived-type data and RECORD data blocks on natural boundaries up to 8 bytes.

-align sequence
Tells the compiler that components of derived types with the SEQUENCE attribute will obey whatever alignment rules are currently in use. The default alignment rules align unsequenced components on natural boundaries.

The default value of `-align nosequence` means that components of derived types with the SEQUENCE attribute will be packed, regardless of whatever alignment rules are currently in use.

Specifying `-fast` sets `-align sequence` so that components of derived types with the SEQUENCE attribute will be naturally aligned for improved performance. Specifying `-align all` also sets `-align sequence`. If your command line includes the `-std`, `-std90`, or `-std95` options, then the compiler ignores `-align sequence`. See Section 3.85.

For More Information:

- On general performance guidelines, see Section 5.7, Additional Source Code Guidelines for Run-Time Efficiency.
- On alignment, see Section 5.4, Data Alignment Considerations.
- On intrinsic data types, see Chapter 9, Data Types and Representation.

3.4 -annotations *keyword* — Place Optimization Information in Source Listing

The `-annotations` *keyword* option specifies that annotations will be added to the source listing file. These annotations indicate which of a set of optimizations the compiler applied to particular parts of the source file. Note that some of the values of *keyword* below may exist for optimizations that are not supported on your platform. If so, the source listing file contains no corresponding annotations.

You can view the resulting annotations in the source listing file to see what optimizations the compiler performed or why the compiler was not able to optimize a particular code sequence.

The default is `-noannotations`, which places no annotations in the source listing file.

The `-annotations` option requires a keyword.

The following options apply:

-annotations none
Same as `-noannotations`, which is the default value.

-annotations all
Selects all of the annotations.

-annotations code
Annotates the machine code listing with descriptions of special instructions used for prefetching, alignment, and so on.

-annotations detail
Provides, where available, an additional level of annotation detail.

-annotations feedback
Annotates the source listing if profile-directed feedback optimizations were used.

-annotations inlining
Indicates where code for a called procedure was expanded inline.

-annotations loop_transforms
Indicates where advanced loop nest optimizations have been applied to improve cache performance (unroll and jam, loop fusion, loop interchange, and so on).

-annotations loop_unrolling
Indicates where a loop was unrolled (contents expanded multiple times).

-annotations prefetching
Indicates where special instructions were used to reduce memory latency.

-annotations shrinkwrapping
Annotates the source listing if code establishing routine context was removed.

-annotations software_pipelining
Indicates where instructions have been rearranged to make optimal use of the processor's functional units. See Section 5.8.6, Software Pipelining.

-annotations tail_calls
Indicates an optimization where a call from one routine to another can be replaced with a jump.

-annotations tail_recursion
Indicates an optimization that eliminates unnecessary routine context for a recursive call.

3.5 -arch *keyword* — Specify Type of Code Instructions Generated

Use the `-arch` *keyword* option to specify the type of Alpha architecture code instructions to be generated for the program unit being compiled.

Compaq Tru64 UNIX provides an operating system kernel that includes an instruction emulator. This emulator allows new instructions, not implemented on the host processor chip, to execute and produce correct results. Applications using emulated instructions will run correctly, but may incur significant software emulation overhead at run time.

All Alpha processors implement a core set of instructions. Certain Alpha processor versions include additional instruction set extensions, such as:

BWX - Byte/Word manipulation instructions
MAX - Multimedia instructions

FIX - Square root and floating-point conversion
CIX - Count

If you omit the `-arch` *keyword*, `-arch generic` is used.

The following options apply:

-arch generic
Generates instructions that are appropriate for most Alpha processors. This is the default.

Programs compiled with the `generic` keyword run on all implementations of the Alpha architecture without any instruction emulation overhead.

-arch host
Generates instructions for the machine the compiler is running on (for example, ev56 instructions on an ev56 processor and ev4 instructions on an ev4 processor).

Programs compiled with the `host` keyword run on other implementations of the Alpha architecture might encounter instruction emulation overhead.

-arch ev4
Generates instructions for ev4 processors (for 21064, 20164A, 21066, and 21068 chips).

Applications compiled with this option will not incur any emulation overhead on any Alpha processor.

-arch ev5
Generates instructions for ev5 processors (some 21164 chips that use only the base set of Alpha instructions, with no extensions).

Applications compiled with this option will not incur any emulation overhead on any Alpha processor.

-arch ev56
Generates instructions for ev56 processors (the 21164 chips that use the BWX extension).

This option permits the compiler to generate any ev4 instruction plus any instruction contained in the BWX extension.

Applications compiled with this option might incur emulation overhead on ev4 and ev5 processors, but will still run correctly.

-arch pca56
Generates instructions for pca56 processors (21164PC chips).

This option permits the compiler to generate any ev4 instruction plus any instruction contained in the BWX or MAX extension.

Applications compiled with this option may incur emulation overhead on ev4, ev5, and ev56 processors, but will still run correctly.

-arch ev6
Generates instructions for ev6 processors (21264 chips).

This option permits the compiler to generate any ev6 instruction, including instructions contained in the BWX, MAX, and FIX extensions.

Applications compiled with this option may incur emulation overhead on ev4, ev5, ev56, and pca56 processors, but will still run correctly.

-arch ev67
Generates instructions for ev67 processors (21264A chips).

This option permits the compiler to generate any ev67 instruction, including instructions contained in the BWX, MAX, FIX, and CIX extensions. Applications compiled with this option may incur emulation overhead on ev4, ev5, ev56, ev6, and pca56 processors, but will still run correctly.

3.6 -assume buffered_io — Buffered Output

Use the `-assume buffered_io` option so that, at run time, the default value BUFFERED='YES' applies to opening sequential output files. As a result, write statements to these disk files will first fill buffers with data before the run-time library routines move it to the files and empty the buffers.

The default value is `-assume nobuffered_io`. Its effect is to have the run-time library routines move data to sequential disk files immediately.

For More Information:

- See Section 5.6.7, Efficient Use of Record Buffers and Disk I/O.

3.7 -assume byterecl — Units for Unformatted File Record Length

Specifying the `-assume byterecl` option:

- Indicates that the OPEN statement RECL value for unformatted files is in byte units. If you omit `-assume byterecl`, Compaq Fortran expects the OPEN statement RECL value for unformatted files to be in longword (4-byte) units.

- Returns the record length value for an INQUIRE by output item list (unformatted files) in byte units. If you omit `-assume byterecl`, Compaq Fortran returns the record length for an INQUIRE by output item list in longword (4-byte) units.
- Returns the record length value for an INQUIRE by unit or file name (unformatted files) in byte units if *all* of the following occur:
  - You had specified `-assume byterecl` for the code being executed
  - The file was opened with an OPEN statement and a RECL specifier
  - The file is still open (connected) when the INQUIRE occurs.

  If any one of the preceding conditions are not met, Compaq Fortran returns the RECL value for an INQUIRE in longword (4-byte) units.

The default is `-assume nobyterecl`.

For More Information:

- On converting and using nonnative unformatted data files, see Chapter 10.
- On the INQUIRE statement, see Section 7.6.

3.8 -assume cc_omp — Enable Conditional Compilation for OpenMP

*(TU*X only)* This option enables conditional compilation as defined by the OpenMP Fortran API. (When `!$space` appears in free-form source or `!$spaces` appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.)

If `-omp` is specified, the default is `-assume cc_omp`; otherwise, the default is `-assume nocc_omp`.

3.9 -assume dummy_aliases — Dummy Variable Aliasing

If you specify the `-assume dummy_aliases` option, the compiler must assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use.

These program semantics do not strictly obey the Fortran 95/90 standards and they slow performance.

If you omit -assume dummy_aliases, the compiler does not need to make these assumptions, which results in better run-time performance. However, omitting -assume dummy_aliases can cause some programs that depend on such aliases to fail or produce wrong answers.

You only need to compile the called subprogram with -assume dummy_aliases.

If you compile a program that uses dummy aliasing with -assume nodummy_aliases in effect, the run-time behavior of the program becomes unpredictable. In such programs, the results depend on the exact optimizations that are performed. In some cases, normal results occur; however, in other cases, results differ because the values used in computations involving the offending aliases differ.

The default is -assume nodummy_aliases.

For More Information:

- See Section 5.9.8, Dummy Aliasing Assumption.

3.10 -assume gfullpath — Source File Path for Debugging

The -assume gfullpath option includes the full source file path in debugger information.

The default is -assume nogfullpath.

3.11 -assume minus0 — Standard Semantics for Minus Zero

The -assume minus0 option tells the compiler to use Fortran 95 standard semantics for the treatment of IEEE floating value −0.0 in the SIGN intrinsic. This applies if the processor is capable of differentiating −0.0 from +0.0.

The default is -assume nominus0, which tells the compiler to use Fortran 90/77 standard semantics in the SIGN intrinsic to treat −0.0 and +0.0 as 0.0.

3.12 -assume noaccuracy_sensitive, -fp_reorder — Reorder Floating-Point Calculations

Specifying -assume noaccuracy_sensitive allows the compiler to reorder code based on algebraic identities (inverses, associativity, and distribution) to improve performance. The numeric results can be slightly different from the default (-assume accuracy_sensitive) because of the way intermediate results are rounded.

The -assume noaccuracy_sensitive and -fp_reorder options are synonymous.

If you omit -assume noaccuracy_sensitive and omit -fast, the compiler uses a limited number of rules for calculations, which might prevent some optimizations.

The default is -assume accuracy_sensitive or -no_fp_reorder (unless -fast was specified).

For More Information:

- See Section 5.9.7, Arithmetic Reordering Optimizations.

3.13 -assume noprotect_constants — Remove Protection from Constants

The -assume noprotect_constants option tells the compiler to pass a copy of a constant actual argument. As a result, the called routine can modify this copy, even though the Fortran standard prohibits such modification. The calling routine does not modify the constant.

The default is -assume protect_constants, which results in passing of a constant actual argument. Any attempt to modify it results in an error.

3.14 -assume nosource_include — INCLUDE file search

This option determines where the compiler searches for INCLUDE files.

If you specify -assume nosource_include, the compiler searches the default directory for INCLUDE files (the same as if -vms is used).

The default is -assume source_include, which instructs the compiler to search the directory where the source files are located.

3.15 -assume nounderscore — Underscore on External Names

Specifying -assume nounderscore prevents the compiler from appending an underscore (_) to the following external user-defined names:

- The main program name
- Named common blocks
- BLOCK DATA blocks
- Names implicitly or explicitly declared external

The name of a blank (unnamed) common block remains _BLNK__ and Compaq Fortran intrinsic names remain the same.

You can also use the cDEC$ ATTRIBUTES ALIAS directive to change how individual external names are spelled.

The default is `-assume underscore`, in which case Compaq Fortran appends an underscore to calls to most external names.

3.16 -assume no2underscores — Two Underscores on External Names

*(L*X only)* Specifying `-assume no2underscores` prevents the compiler from appending two underscores (`__`) to the following external user-defined names:

- The main program name
- Named common blocks
- BLOCK DATA blocks
- Names implicitly or explicitly declared external

The name of a blank (unnamed) common block remains `_BLNK__` and Compaq Fortran intrinsic names remain the same.

You can also use the cDEC$ ATTRIBUTES ALIAS directive to change how individual external names are spelled.

The default is `-assume 2underscores`, in which case Compaq Fortran appends two underscores to calls to *most* external names.

If `-assume nounderscore` is specified, `-assume 2underscores` and `-assume no2underscores` are ignored.

3.17 -assume pthreads_lock — Thread Lock Selection for Parallel Execution

*(TU*X only)* The `-assume pthreads_lock` option lets you select the kind of locking used for an unnamed critical section (under `-mp` and `-omp`). In the rare event that your program should need more restrictive locking, use this option to set `_OtsPthreadLock`, which locks out other pthreads in addition to all critical sections.

The default is `-assume nopthreads_lock`, which results in much faster compile times. Using the default creates a `bifcall (enter_critical)` to set `_OtsGlobalLock`, which does not lock out other pthreads, but does provide a single lock for all unnamed critical sections.

3.18 -automatic, -static — Local Variable Allocation

The -automatic and -static options control how local variables are allocated.

Specifying -automatic puts local variables on the run-time stack. Specifying -automatic sets -recursive.

Specifying -static causes all local variables to be statically allocated.

A subprogram declared with the RECURSIVE keyword is always recursive (whether you specify or omit the -static option).

The default is -static or -noautomatic.

3.19 -c — Inhibit Linking and Retain Object File

Use the -c option to suppress the loading phase of the compilation and force an object file to be produced even if only one program is compiled. If you omit -c, the linker creates an executable program and any temporary object files are deleted.

If you specify the -o *output* option with the -c option and multiple Fortran files, a single .o file is created, allowing full interprocedure optimizations.

However, if you specify multiple source files and the -c option without the -o *output* option, multiple object files are created and interprocedure optimizations do not occur.

3.20 -call_shared, -non_shared, -shared — Shared Library Use

These options relate to shared libraries.

The default is -call_shared.

The following options apply:

-call_shared
Causes the linker to search for unresolved references in shared libraries (.so files) before archive libraries (.a files).

If the unresolved reference is found in the shared library, the unresolved reference is resolved when the program is run (during program loading), reducing executable program size. For more information on related options, see Section 2.5.2.

-non_shared
Requests that the linker search only in archive libraries (`.a` files) for unresolved references; shared library (`.so`) files are not searched. Object files (`.o` suffix) from archives are included in the executable produced.

-shared
Produces a dynamic shareable object that can be included into a shared library. This includes creating all of the tables for run-time linking and resolving references to other specified shared objects. The object created can be used by the linker to produce a shareable object that other dynamic executables can use at run time. To explicitly name the resulting object file or shared library, use the `-o` option.

If you also specify the `-c` option when you use `-shared`, a `.o` file is created; otherwise, a `.so` file is created.

For More Information:

- See Section 2.5.2, Specifying Types of Object Libraries.
- See Section 2.6, Creating Shared Libraries.

3.21 -ccdefault *keyword* — Carriage Control for Terminals

The `-ccdefault` *keyword* options specify default carriage control when a terminal displays a file.

The following options apply:

-ccdefault fortran
Results in normal Fortran interpretation of the first character, such as the character "0" resulting in a blank line before output.

-ccdefault list
Results in one line feed between records.

-ccdefault none
Results in no carriage-control processing.

-ccdefault default
Specifies that the compiler is to use the default carriage-control setting. This is the default value of `-ccdefault`.

This default setting can be affected by the -vms option as follows:

- If -vms -ccdefault default is specified and the file is formatted and the unit is connected to a terminal, then normal Fortran interpretation of the first character occurs (same as -ccdefault fortran).
- If -novms -ccdefault default is specified, then LIST carriage control occurs (same as -ccdefault list).

3.22 -check arg_temp_created — Check for Copy of Temporary Arguments

Specifying -check arg_temp_created generates code to check if actual arguments are copied into temporary storage before routine calls. If a copy is made at run time, an informative message is displayed.

The default is -check noarg_temp_created.

3.23 -check bounds, -C, -check_bounds — Boundary Run-Time Checking

Specifying -check bounds or -C or -check_bounds (all are synonymous) generates code to perform run-time checks on array subscript and character substring expressions. A run-time message is reported if the expression is outside the addresses of the dimension of the array or outside the length of the string.

The default (-check nobounds) suppresses range checking.

For array bounds, each individual dimension is checked. Array bounds checking is *not* performed for arrays that are dummy arguments in which the last dimension bound is specified as * or when both upper and lower dimensions are 1.

Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.

3.24 -check format — Format Mismatches at Run Time

Specifying -check format issues the run-time FORVARMIS fatal error when the data type of an item being formatted for output does not match the format descriptor being used (for example, a REAL*4 item formatted with an I edit descriptor).

Specifying -check noformat disables the run-time message associated with format mismatches (number 61). It also requests that the data item be formatted using the specified descriptor, unless the length of the item cannot accommodate the descriptor. (For example, it is still an error to pass an INTEGER*2 item to an E edit descriptor.) Specifying -check noformat allows such format mismatches as a REAL*4 item formatted with an I edit descriptor.

If you specify the -vms option, the default is -check format. Otherwise, the default is -check noformat.

3.25 -check nopower — Allow Special Floating-Point Expressions

Specifying -check nopower allows certain arithmetic expressions containing floating-point numbers and exponentiation to be evaluated and return a result rather than causing the compiler to display a run-time message and stop the program. The specific arithmetic expressions include:

- 0.0 ** 0.0
- *negative-value ** integer-value-of-type-real*

For example, if you specify -check nopower, the calculation of the expression 0.0 ** 0.0 results in 1. The expression (–3.0) ** 3.0 results in –27.0.

The default is -check power, which means that for such expressions, an exception occurs, error message number 65 is displayed, and the program stops.

3.26 -check omp_bindings — OpenMP Fortran API Binding Rules Checking

*(TU*X only)* Specifying -check omp_bindings provides run-time checking to enforce the binding rules for OpenMP Fortran API compiler directives. When this option is in effect, the Compaq Fortran compiler issues an error message if your program attempts any of the following actions:

- Enter a DO, SINGLE, or SECTIONS directive if already in a work-sharing construct, a CRITICAL SECTION, or a MASTER.
- Execute a BARRIER directive if already in a work-sharing construct, a CRITICAL SECTION, or a MASTER.
- Execute a MASTER directive if already in a work-sharing construct.
- Execute an ORDERED directive if already in a CRITICAL SECTION.
- Execute an ORDERED directive unless already in an ORDERED DO.

Additionally, the following conditions apply:

- `-omp` implies `-check omp_bindings`
- `-fast -omp` implies `-check noomp_bindings`, regardless of the placement of `-fast`
- If you want the checking done on `-mp`, specify `-check omp_bindings` explicitly

The default is `-check noomp_bindings`.

For More Information:

- On using OpenMP Fortran API compiler directives, see Section 6.1.
- On rules that apply to dynamic binding of these directives, see the *Compaq Fortran Language Reference Manual*.

3.27 -check output_conversion — Truncated Format Mismatches at Run Time

Specifying `-check output_conversion` issues the run-time OUTCONERR continuable error message when a data item is too large to fit in a designated format descriptor field. The field is filled with asterisks (*) and execution continues.

Specifying `-check nooutput_conversion` disables the run-time message (number 63) associated with format truncation. Error number 63 occurs when a number could not be output in the specified format field length without loss of significant digits (format truncation).

If you specify the `-vms` option, the default is `-check output_conversion`. Otherwise, the default is `-check nooutput_conversion`.

3.28 -check overflow — Integer Overflow Run-Time Checking

Specifying `-check overflow` requests that all integer calculations (INTEGER, INTEGER with a kind parameter, or INTEGER with a length specifier) be checked for arithmetic overflow at run time. Real and complex calculations are always checked for overflow and are not affected by `-check overflow`.

If the check reveals that the code caused arithmetic overflow, an error message is displayed at run time, program execution stops, and a core dump occurs.

Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.

The default is `-check nooverflow`.

3.29 -check underflow — Floating-Point Underflow Run-Time Checking

*(TU*X only)* Specifying `-check underflow` displays a run-time warning message (maximum of twice) when a floating-point underflow occurs.

Floating-point underflow is allowed to continue if the `-fpe3` or `-fpe4` option was specified; otherwise, the underflow value is replaced by zero. A count of how many occurrences of each type of exception is displayed on program completion.

The default is `-check nounderflow`.

For More Information:

- On underflow ranges for floating-point data types, see Section 9.4, Native IEEE Floating-Point Representations and Exceptional Values.
- On controlling floating-point exception handling, see Section 3.44.

3.30 -convert *keyword* — Unformatted Numeric Data Conversion

The `-convert` *keyword* options determine which format is used to read and write unformatted files.

Numeric data in an unformatted file is expected to be in native little endian integer and little endian IEEE S_float, T_float, and X_float floating-point formats, unless handled otherwise by the application.

You can specify the unformatted data format for specific unit numbers by using the OPEN statement CONVERT specifier or by setting the appropriate `FORT_CONVERT`*n* environment variable.

The default is `-convert native`.

The following options apply:

-convert big_endian
Specifies that unformatted data will be in big endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
IEEE floating point format for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32

Note that INTEGER*1 data is the same for little endian and big endian.

-convert cray
Specifies that unformatted data will be in big endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
CRAY floating-point format for REAL*8 or COMPLEX*16

-convert fdx
Specifies that unformatted data will be in little endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
Compaq VAX™ floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and X_floating for REAL*16 or COMPLEX*32

-convert fgx
Specifies that unformatted data will be in little endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
Compaq VAX floating-point format F_floating for REAL*4 or COMPLEX*8, G_floating for REAL*8 or COMPLEX*16, and X_floating for REAL*16 or COMPLEX*32

-convert ibm
Specifies that unformatted data will be in big endian format of the appropriate size:

INTEGER*1, INTEGER*2, or INTEGER*4
IBM System\370 floating-point format for REAL*4 or COMPLEX*8 (IBM short 4) and for REAL*8 or COMPLEX*16 (IBM long 8)

-convert little_endian
Specifies that unformatted data will be in native RISC little endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
IEEE floating-point format for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32

Note that INTEGER*1 data is the same for little endian and big endian.

Using `-convert little_endian` produces the same results as `-convert native`.

-convert native
Specifies that unformatted data should not be converted.

This is the default.

-convert vaxd
Specifies that unformatted data will be in little endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
Compaq VAX floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and H_floating for REAL*16 or COMPLEX*32

-convert vaxg
Specifies that unformatted data will be in little endian format of the appropriate size:

INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8
Compaq VAX floating-point format F_floating for REAL*4 or COMPLEX*8, floating-point format G_floating for REAL*8 or COMPLEX*16, and H_ floating for REAL*16 or COMPLEX*32

For More Information:

- On converting unformatted files, including using the OPEN statement CONVERT specifier and using FORT_CONVERT*n* environment variables, see Chapter 10.
- On native data types, see Chapter 9.
- On OpenVMS floating-point data types, see Section A.4.3.

3.31 -cpp and Related Options — Run C Preprocessor

Specifying `-cpp` runs the C preprocessor `cpp` on all Fortran source files before compiling.

The default is `-nocpp`.

The following `cpp` macros are defined by the `f90` command when any `.f90`, `.F90`, `.f`, `.F`, `.for`, or `.FOR` file is being compiled:

`LANGUAGE_FORTRAN_90` *(UN*X only)*
`__LANGUAGE_FORTRAN_90__` *(UN*X only)*
`LANGUAGE_FORTRAN` *(UN*X only)*
`__LANGUAGE_FORTRAN__` *(UN*X only)*
`linux` *(L*X only)*
`__linux__` *(L*X only)*
`__alpha`
`__osf__` *(UN*X only)*

unix *(UN*X only)*
__unix__ *(UN*X only)*
__OPENMP if -omp is specified *(UN*X only)*

If you omit -cpp and -P, the cpp preprocessor is not run unless the file name suffix is .F or .FOR (for fixed-form source files) or .F90 (for free-form source files).

If you specify -cpp or -P, you can also specify these options:

- Section 3.32, -Dname, -Dname=def, -Dname="string" — Define Symbol Names
- Section 3.51, -I — Remove Directory from Include Search Path
- Section 3.52, -Idir — Add Directory for Module and Include File Search
- Section 3.56, -K — Keep Temporary Files
- Section 3.92, -Uname — Undefine Preprocessor Symbol Name

For More Information:

- On cpp and its options, see cpp(1).
- On the options the f90 command passes to the cc driver program (which do not usually apply to Compaq Fortran files), see Section 2.2.2.
- On recognized file name suffix characters and their meanings, see Section 2.1.1, File Suffixes and Source Forms.

3.31.1 -M — Request cpp Dependency Lists for make

Specifying -M requests that cpp generate dependency lists suitable for make, instead of the normal output.

3.31.2 -P — Retain cpp Intermediate Files

Specifying -P runs only the cpp preprocessor and puts the result for each source file in an intermediate file, after processing by any appropriate preprocessors. The intermediate file does not have line numbers (#) in it.

The following naming convention is used: .f file results are put into .i files. .f90 file results are put into .i90 files.

This option sets the -cpp option.

3.31.3 -Wp,*-xxx* — Pass Specified Option to cpp

The `-Wp,-xxx` option allows you to pass an option (specified by *xxx*) directly to the `cpp` preprocessor. This is useful for options that the driver does not normally pass to the preprocessor.

For example, to pass `-C -M` to `cpp`, use:

```
-Wp,-C,-M
```

The `-Wp` option does not invoke `cpp`. Use `-cpp` to invoke `cpp`.

3.32 -D*name*, -D*name=def*, -D*name="string"* — Define Symbol Names

Specifying `-Dname` or `-Dname=def` or `-Dname="string"` defines *name* for use by either the `cpp` preprocessor or by Fortran conditional compilation.

When *name=def* is used for Fortran conditional compilation, *def* can be an integer string or a character string delimited by double quotation marks as in `-Dvehicle_type="red convertible"`. When *name=def* is used for the `cpp` preprocessor, the definition is interpreted as if by `#define`. Regardless of the usage, if no definition is given, then *name* is defined as "1".

For example:

```
% f90 -Dnum=4 -Dber -Dclock="the time" foo.f90
```

In this example, *num* is the integer 4, *ber* is "1", and *clock* is the string "the time".

Preprocessor symbols can be used for conditional compilation by using the cDEC$ IF directive construct (see the *Compaq Fortran Language Reference Manual*) or by using `cpp`. Predefined preprocessor symbols are defined in Section 3.31.

Specifying `-noD` prevents the compiler from receiving any `-Dname` or `-Uname` options. (The default is to allow the compiler to receive these options.) However, the `cpp` preprocessor always receives these options.

For More Information:

On the C preprocessor, see `cpp`(1).

3.33 -d_lines — Debugging Statement Indicator, Column 1

Use the -d_lines option to request that lines in fixed-format source files that have a D or a d character in column 1 be compiled instead of treated as comment lines. Such lines might print the values of variables or otherwise provide useful debugging information.

The -d_lines option does not apply to free-format files.

The default (-nod_lines) treats all lines with a D or a d in column 1 as comment lines.

3.34 -double_size 128, -double_size 64 — Double Precision Data Size

Use the -double_size 128 option to specify the size of DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL*16 and DOUBLE COMPLEX declarations, constants, functions, and intrinsics as COMPLEX*32.

The default is -double_size 64, where DOUBLE PRECISION declarations, constants, functions, and intrinsics are specified as REAL*8 and DOUBLE PRECISION COMPLEX declarations, constants, functions, and intrinsics as COMPLEX*16.

3.35 -error_limit *num*, -noerror_limit — Limit Error Messages

Use the -error_limit *num* and -noerror_limit options to specify the maximum number of error-level or fatal-level compiler errors allowed for a given file specified on the command line.

If -c is specified on the command line and the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

If -c is not specified, a warning message is issued and compilation terminates.

If you specify -noerror_limit, there is no limit on the number of errors that are allowed.

The default is -error_limit 30, or a maximum of 30 error-level and fatal-level messages.

3.36 -extend_source — Line Length for Fixed-Format Source

Specifying `-extend_source` treats the statement field of each source line as ending in column 132 instead of column 72.

The default (`-noextend_source` or `-col72`) specifies 72-column lines.

Specifying `-extend_source` sets the `-fixed` option.

Specifying `-extend_source` or `-col132` sets the source to fixed format even if implicit naming rules set it to free format.

For More Information:

- On recognized file name suffix characters and their relationship to fixed and free source formats, see Section 2.1.1.
- On column positions and more complete information on the fixed and free source formats, see the *Compaq Fortran Language Reference Manual*.

3.37 -f66, -66, -nof77, -onetrip, -1 — Use FORTRAN 66 Semantics

Specifying `-f66`, `-66`, `-nof77`, `-onetrip`, or `-1` (all are synonymous) tells the compiler to select FORTRAN 66 (FORTRAN IV) interpretations in cases of incompatibility.

This option applies the following FORTRAN 66 semantics:

- DO loops are always executed at least once.
- FORTRAN 66 EXTERNAL statement syntax and semantics are allowed. See the *Compaq Fortran Language Reference Manual*.
- In the OPEN statement:
  - If the BLANK specifier is omitted, the default is BLANK='ZERO'.
  - If the STATUS specifier is omitted, the default is STATUS='NEW'.

The default is `-nof66`, which specifies that FORTRAN 77 interpretation rules are used for those statements that have a meaning incompatible with FORTRAN 66. Thus, DO loops whose lower range exceeds the upper range will not be executed.

3.38 -f77, -nof66 — Use FORTRAN 77 Semantics

Specifying `-f77` or `-nof66` (they are synonymous) enforces FORTRAN 77 semantics instead of FORTRAN 66 semantics.

In Fortran F90, the default is to apply Fortran 95/90 semantics.

3.39 -f77rtl — Use Fortran 77 Run-Time Behavior

Specifying `-f77rtl` tells the compiler to create an executable file whose run-time behavior is that of Compaq Fortran 77 instead of Compaq Fortran.

Specifying this option controls control the following run-time behavior:

- When the unit is not connected to a file, some INQUIRE specifiers will return different values:
  - NUMBER returns 0
  - ACCESS returns 'UNKNOWN'
  - BLANK returns 'UNKNOWN'
  - FORM returns 'UNKNOWN'
- List-directed input for character strings must be delimited by apostrophes or quotation marks, or an error will result.
- When processing NAMELIST input:
  - Column 1 of each record is skipped.
  - The '$' or '&' that appears prior to the group-name must appear in column 2 of the input record.

The default is `-nof77rtl`.

3.40 -fast — Set Options to Improve Run-Time Performance

Specifying `-fast` sets the following options:

- `-align dcommons` (see Section 3.3). However, if any of the `-std` options are set (see Section 3.85), then the compiler ignores `-align dcommons`.
- `-align sequence` (see Section 3.3). However, if any of the `-std` options are set (see Section 3.85), then the compiler ignores `-align sequence`.
- `-arch host` (see Section 3.5)
- `-assume bigarrays` *(TU*X only)* (see Section 3.50.1)
- `-assume noaccuracy_sensitive` (same as `-fp_reorder`; see Section 3.12)

- `-assume nozsize` *(TU*X only)* (see Section 3.50)
- `-math_library fast` (see Section 3.61)
- `-O4` (reinforced as the default; see Section 3.72)
- `-tune host` (see Section 3.90)

Avoid using the `-fast` option unless you understand the options that `-fast` sets. For example, the `-fast` option sets the `-assume noaccuracy_sensitive` and `-math_library fast` options, which can change the calculated results of a program.

Also, before you use `-fast` with the `-hpf` option, first make sure your program does not use any zero-sized arrays or array sections (see Section 3.50). And, while the setting of `-arch host` and `-tune host` generates optimal code for the computer architecture on which the compiler is running, this code may run slowly on another version of the computer architecture.

The default is `-nofast`.

3.41 -feedback *file*, -gen_feedback, -cord — Create and Use Feedback Files

Note

These options apply only to Tru64 UNIX systems.

These options allow the creation and use of a feedback file, which can improve run-time performance. You can optionally use `cord` to rearrange procedures.

You create a feedback file by using a series of commands, including `f90` with the `-gen_feedback` option, `pixie` *(TU*X only)*, and `prof`. (See Section 5.3.5.)

The options are:

-cord
Runs the `cord` procedure-rearranger `cord` after the linker creates the executable program. This rearrangement reduces the cache conflicts of the program's text. The output of `cord` is left in the file specified by the `-o` *output* option (or `a.out` by default). At least one `-feedback` *file* must be specified.

-feedback *file*
Specifies the *file* to be used by `-cord` or the compiler for further optimizations.

This *file* is produced by the prof command with its -feedback option from an execution of the program produced by the pixie command *(TU*X only)*.

-gen_feedback
Directs the compiler to generate code that will produce accurate feedback information when profiled.

For example:

```
f90 -gen_feedback -o x x.f
pixie x
x.pixie
prof x -pixie -feedback x.fb x.Addrs x.Counts
f90 -feedback x.fb -O5 -fast -o x x.f
```

Using -gen_feedback changes the default optimization level from -O4 to -O0. Avoid using -gen_feedback with optimizations higher than -O3.

For More Information:

- On using cord, see Section 5.3.5.
- On profiling options, see Section 3.77.
- On timing program execution, see Section 5.2.

3.42 -fixed, -free — Fortran Source Format

Use the -fixed and -free options to specify the source format:

-fixed
Specifies that the source file is fixed format, regardless of the file name suffix.

Note that source files with a suffix of .f, .for, .F, .FOR, or .i are assumed to be fixed format.

-free
Specifies that the source file is free format, regardless of the file name suffix.

Note that source files with a suffix of .f90, .F90, or .i90 are assumed to be free format.

You cannot specify both -free and -fixed on the same command line.

Table 3–2 summarizes how the -free and -fixed options interact with the file suffix of a source file.

Table 3–2 Interaction of File Suffix and the -free and -fixed Options on Source Form

Suffix	-free Option	-fixed Option	Expected Source Form
.f90, .F90, .i90	Not specified	Not specified	Free form
.f90, .F90, .i90	Specified	Not specified	Free form
.f90, .F90, .i90	Not specified	Specified	Fixed form
.f, .F, .for, .FOR, .i	Not specified	Not specified	Fixed form
.f, .F, .for, .FOR, .i	Specified	Not specified	Free form
.f, .F, .for, .FOR, .i	Not specified	Specified	Fixed form

For More Information:

- On recognized file name suffix characters and their relationship to fixed and free source formats, see Section 2.1.1, File Suffixes and Source Forms.
- On column positions and more complete information on the fixed and free source formats, see the *Compaq Fortran Language Reference Manual*.

3.43 -fpconstant — Handling of Floating-Point Constants

Specifying `-fpconstant` requests that a single-precision constant assigned to a double-precision variable be evaluated in double precision.

If you omit `-fpconstant`, a single-precision constant assigned to a double-precision variable is evaluated in single precision. The Fortran standard requires that the constant be evaluated in single precision.

Certain programs created for FORTRAN-77 compilers (including Compaq Fortran 77) may show different results, because they rely on single-precision constants assigned to a double-precision variable to be evaluated in double precision.

In the following example, if you specify `-fpconstant`, identical values are assigned to D1 and D2. If you omit the `-fpconstant` option, Compaq Fortran will obey the standard and assign a less precise value to D1.

```
REAL (KIND=8) D1, D2
DATA D1 /2.71828182846182/    ! REAL (KIND=4) value expanded to double
DATA D2 /2.71828182846182D0/  ! Double value assigned to double
```

3.44 -fpe*n* — Control Arithmetic Exception Handling and Reporting

Use the `-fpe0`, `-fpe1` *(TU*X only)*, `-fpe2` *(TU*X only)*, `-fpe3`, or `-fpe4` *(TU*X only)* options to control floating-point exception handling at run time for the main program. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported.

Using these options, you can do the following:

- For floating-point calculations that result in overflow, division by zero, or invalid data, you can request that the compiler:
  - Either stop program execution and perform a core dump (faster run-time performance) or let the program continue with the generated Infinity (+ or –) or NaN exceptional value (slower run-time performance).
  - *(TU*X only)* Display or not display a message when the arithmetic exception occurs.
  - Report exceptions closer to the instruction that caused the exception.
- For floating-point calculations that result in underflow (a denormalized number; see Section 9.4), you can request that the compiler:
  - Either set the calculated underflowed (denormalized) value to zero (0) or leave the denormalized value as is (slowest run-time performance).
  - *(TU*X only)* Display or not display a message when the arithmetic exception occurs (you can also specify `-check underflow`).

  For information on the underflow ranges for floating-point data types, see Section 9.4.
- When a denormalized number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression, you can request that the compiler:
  - Either stop program execution and perform a core dump (faster run-time performance) or let the program continue with slower run-time performance.
  - *(TU*X only)* Display or not display a message when the arithmetic exception occurs.

To associate an exception with the instruction that causes the exception, specify any value other than `-fpe0` (or specify `-synchronous_exceptions`).

*(TU*X only)* Specifying `-fpe2` or `-fpe4` displays error messages indicating the cause of the exceptions.

To allow source line correlation when using a debugger to locate the exception, specify the `-g` option when compiling the program (see Chapter 4).

The default is `-fpe0` or `-fpe`, which are synonymous.

The following options apply:

-fpe0, -fpe
Terminates a program if a floating-point operation results in overflow or division by zero, or if the operands are exceptional values. Before termination, a message is issued and a core dump file is created. If an operand is a denormalized number, it is set to zero.

In the case of floating-point underflow, the program does not terminate, but continues with the underflow value set to zero. On Tru64 UNIX systems, a warning message is issued if `-check underflow` is set.

Examining the `core` file shows the exception one or more instructions *after* the instruction that caused the exception (unless you also specified `-synchronous_exceptions`).

To obtain the fastest run-time performance, use `-fpe0`. This is the default. Using other `-fpe`*n* values will slow run-time performance.

-fpe1
*(TU*X only)* Continues program execution and generates an Infinity or Nan if a floating-point operation results in overflow, division by zero, or invalid data.

If an operand is a denormalized number, it is set to zero. In the case of floating-point underflow, the underflow value is set to zero. A warning message is issued if `-check underflow` is set.

-fpe2
*(TU*X only)* Continues program execution if a floating-point operation results in overflow, division by zero, or invalid data. A warning message is generated for the first two occurrences. If an operand is a denormalized number, it is set to zero.

In the case of floating-point underflow, the underflow value is set to zero. On program completion, a count of how many times each exception occurred is displayed.

-fpe3
Continues program execution if a floating-point operation results in overflow, division by zero, invalid data, or underflow. Calculated denormalized numbers are left as is.

For underflow, the underflow value is not set to zero, and gradual underflow occurs. On Tru64 UNIX systems, a warning message is issued if `-check underflow` is set.

-fpe4
*(TU*X only)* Continues program execution if a floating-point operation results in overflow, division by zero, invalid data, or underflow. A warning message is generated for the first two occurrences. Calculated denormalized numbers are left as is.

In the case of floating-point underflow, the underflow value is not set to zero and gradual underflow occurs. On program completion, a count of how many times each exception occurred is displayed.

3.44.1 Hints on Using These Options

Table 3–3 summarizes the floating-point exception handling options.

Table 3–3 Summary of Floating-Point Exception Command-Line Options

Option	Handling of Underflow	Handling of Overflow, Division by Zero, and Invalid Data
`-fpe0`, `-fpe`	Sets any calculated denormalized value (result) to zero and lets the program continue. Use of a denormalized number in an arithmetic expression uses a value of zero and execution continues. *(TU*X only)* A message is displayed only if `-check underflow` is also specified.	Exceptional values are *not* allowed. The program terminates after displaying a message and creating a core dump file.
`-fpe1` *(TU*X only)*	Sets any calculated denormalized value to zero and lets the program continue. Use of a denormalized number in an arithmetic expression uses a value of zero and execution continues. A message is displayed only if `-check underflow` is also specified.	The program continues (no core dump). No message is displayed. A NaN or Infinity (+ or –) exceptional value is generated.
`-fpe2` *(TU*X only)*	Sets any calculated denormalized value to zero and lets the program continue. Use of a denormalized number in an arithmetic expression uses a value of zero and execution continues. A message is displayed (`-check underflow` is not needed).	The program continues (no core dump). A message is displayed a maximum of twice for each type of exception. A NaN or Infinity (+ or –) is generated.
`-fpe3`	Leaves any calculated denormalized value as is. The program continues, allowing gradual underflow. Use of a denormalized number in an arithmetic expression results in program continuation, but with slower performance. *(TU*X only)* A message is displayed only if `-check underflow` is also specified.	The program continues (no core dump). No message is displayed. A NaN or Infinity (+ or –) is generated.
`-fpe4` *(TU*X only)*	Leaves any calculated denormalized value as is. The program continues, allowing gradual underflow. Use of a denormalized number in an arithmetic expression results in program continuation, but with slower performance. A message is displayed (`-check underflow` is not needed).	The program continues (no core dump). A message is displayed a maximum of twice for each type of exception. A NaN or Infinity (+ or –) is generated.

On Tru64 UNIX systems, the exception message reporting specified by the `-fpe`*n* options applies only to the main program and cannot be changed during program execution.

To help you debug a routine, you can associate an exception with the instruction that causes it by specifying any value other than `-fpe0` (such as `-fpe3`) or specify `-synchronous_exceptions`.

When compiling different routines in a program separately, you should use the same -fpe*n* value. For example, assume:

- Routine A is compiled with -fpe3. Routine A can create and use an exceptional value.
- Routine B is compiled with -fpe0. Routine B cannot create or use an exceptional value (program stops).

If routine A passes an exceptional value to routine B and routine B uses that exceptional value in an arithmetic expression, program execution stops and a core file is created.

You can call the for_set_fpe routine to set the floating-point exception handling for subprograms (including C functions) or perhaps to change the setting for the main program.

To use for_set_fpe in most cases, you should recompile the program with -fpe1 or higher. You should not change the exception settings to request program continuation with for_set_fpe if you compiled the program using -fpe0.

Both the for_set_fpe and for_get_fpe routines can be used by Fortran programs using the INTEGER parameter values located in /usr/include/for_fpe_flags.f. Programs written in C can call the for_rtl_init_ routine prior to calling the for_get_fpe or for_get_fpe routines, and must include an equivalent header file, /usr/include/for_fpe_flags.h.

For programs that use a number of denormalized values (such as those that allow gradual underflow with -fpe3, the impact on run-time performance can be significant.

If you use the -math_library fast (or -fast) option along with an -fpe*n* option, the -fpe*n* option is ignored when arithmetic values are evaluated by math library routines.

If you specify the -speculate all or -speculate by_routine options along with one of the -fpe*n* options that request exception reporting, exceptions are not reported as expected (see Section 3.84).

For More Information:

- On IEEE floating-point exception handling, see *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985) and ieee(3).
- On Compaq Fortran floating-point exception handling, see Chapter 14.
- On native IEEE floating-point data representation and IEEE exceptional values, see Section 9.4.

- On the `for_set_fpe` and `for_get_fpe` routines, see `for_set_fpe`(3f) and Chapter 12 and Section 14.3.

3.45 -fpp — Run Fortran Preprocessor

Use the `-fpp` option to run the Fortran preprocessor on all Fortran source files in the command line before the Fortran compiler executes. This option has no effect on any C source files in the command line. The Fortran preprocessor has a subset of the functionality of the C preprocessor (which the `-cpp` option invokes).

The default is `-nofpp`, which means that neither the Fortran preprocessor nor the C preprocessor runs on the Fortran source files in the command line.

3.46 -fprm *keyword* — Control Floating-Point Rounding Mode

The `-fprm nearest`, `-fprm dynamic` *(TU*X only)*, `-fprm chopped`, and `-fprm minus_infinity` options allow you to control how rounding occurs during floating-point operations.

The rounding mode applies to each program unit being compiled.

The following options apply:

-fprm nearest
Causes the compiler to round results of floating-point operations toward the nearest representable value.

If the unrounded value is halfway between two representable values, the even value is chosen.

This is the default.

-fprm chopped
Causes the compiler to round results of floating-point operations toward zero.

-fprm minus_infinity
Causes the compiler to round results of floating-point operations toward the next smallest representative value.

-fprm dynamic
*(TU*X only)* Allows run-time selection of a rounding mode. You can modify your program to:

- Call the read_rnd Compaq Tru64 UNIX routine to obtain the current rounding mode. The current mode is stored in the floating-point control register (fpcr).
- Call the write_rnd Compaq Tru64 UNIX routine to set the rounding mode and obtain the previous rounding mode.

When you call write_rnd, you can set the rounding mode to one of the following settings (see write_rnd(3)):

- Round toward zero or truncate (same as -fprm chopped)
- Round toward nearest (same as -fprm nearest)
- Round toward plus infinity
- Round toward minus infinity (same as -fprm minus_infinity)

If you compile with -fprm dynamic and do not call write_rnd, the default setting (-fprm nearest) is used.

For the fastest run-time performance, avoid using -fprm dynamic.

For More Information:

- On IEEE floating-point rounding modes, see *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985), write_rnd(3), ieee_functions(3), and ieee(3).
- On the floating-point control register and Alpha architecture, see *Alpha Architecture Reference Manual*.

3.47 -fuse_xref — Cross-Reference Information for Compaq FUSE

Use the -fuse_xref option *(TU*X only)* to request that Compaq Fortran generate a data file that the Compaq FUSE Database Manager uses to create a cross-reference database file. This improves the performance of the Compaq FUSE Call Graph Browser and Cross-Referencer, which use the database file for their operations.

3.48 -g0, -g1, -g2 or -g, -g3, -ladebug — Traceback and Symbol Table Information

Use the -g0, -g1, -g2, or -g, -g3, and -ladebug options to control the amount of symbol table information in the object file.

If you intend to use the Compaq Ladebug debugger, specify the -ladebug option and one of the following: -g, -g2, or -g3.

The default is -g1.

The following options apply:

-g0
Provides no traceback or symbol table information needed for debugging or profiling. Only symbol information needed for linking (global symbols) is produced. The size of the resulting object file is the minimum size.

-g1
Produces traceback information, which allows program counter to source file line correlation, but not symbol table information needed for debugging. Specifying -g1 produces the global symbol information needed for linking or profiling.

The object file size is somewhat larger than if -g0 was specified, but is smaller than if either -g2 or -g3 was specified.

This is the default.

-g2, -g
Produces the following:

- Traceback information
- Symbol table information needed for full symbolic debugging of unoptimized code
- Global symbol information needed for linking or profiling

If you use this option and do not specify an -O*n* option, the default optimization level changes to -O0, which disables nearly all optimizations to make debugging more accurate (the default optimization level is usually -O4). If you use this option and specify an -O*n* option other than -O0, a warning message is displayed.

-g2 and -g are synonymous.

-g3
Produces traceback information, symbol table information needed for symbolic debugging of optimized code, and global symbol information needed for linking or profiling. This option can provide additional debugging information to describe the effects of optimization, but debugging inaccuracies may occur as a result of optimizations.

-ladebug
Produces additional symbolic debugger information for the Compaq Ladebug debugger, if you specify `-g`, `-g2`, or `-g3`. This allows use of standard Fortran syntax when printing dynamic arrays using Compaq Ladebug, including array sections.

If you specify `-g0` or `-g1` (the default) with `-ladebug`, the `-ladebug` option is ignored.

For More Information:

- See Section 4.2, Compaq Fortran Options for Debugging.
- See Section 4.12, Debugging Optimized Programs.
- See Section 5.8, Optimization Levels: the -On Option.

3.49 -granularity *keyword* — Control Shared Memory Access to Data

The `-granularity` *keyword* options allow you to control the size of shared data in memory that can be safely accessed from different threads. You do not need to specify this option for local data access by a single process, unless asynchronous write access from outside the user process might occur.

The default is `-granularity quadword`.

To be written from multiple threads, data must be naturally aligned and declared as VOLATILE (so it is not held in registers). To ensure alignment in common blocks, derived-type structures, and record structures, use the `-align` *keyword* option.

The following options apply:

-granularity byte
Ensures that all data 1 byte or greater can be accessed from different threads sharing data in memory. This option will slow run-time performance.

-granularity longword
Ensures that naturally aligned data of 4 bytes or greater can be accessed safely from different threads sharing access to that data in memory.

When this option is in effect, attempts to access smaller size data or misaligned data can result in data items that are inconsistently updated for multiple threads.

-granularity quadword
Ensures that naturally aligned data of 8 bytes can be accessed safely from different threads sharing data in memory.

When this option is in effect, attempts to access smaller size data or misaligned data can result in data items that are inconsistently updated for multiple threads.

For More Information:

- On the Alpha architecture, see the *Alpha Architecture Reference Manual*.
- On allowing multiple processes to access common block data in a shared library (using memory mapping), see `shcom_connect`(3f).
- On intrinsic data types, see Chapter 9.

3.50 -hpf, -hpf *num*, and Related Options — Compile HPF Programs for Parallel Execution

Note

These options apply only to Tru64 UNIX systems.

`-wsf` is the nonpreferred synonym for `-hpf`.

`-wsf` *num* is the nonpreferred synonym for `-hpf` *num*.

Use the `-hpf` option to specify that the HPF program will be compiled to run in parallel on multiple processors using the Message Passing Interface (MPI).

If you omit the `-hpf` option, the executable program will run in a nonparallel (serial) run-time environment and HPF directives will only be checked for correct syntax.

The optional *num* parameter specifies the number of processors on which the program is intended to run, for example, `-hpf 3`. If a number is not specified, the compiler generates code that can run on any number of processors. More efficient code is generated when *num* is specified.

When the HPF parallel programs are compiled with the -c option and linked separately, specify -hpf both when compiling and linking the program.

The following options are not compatible with the -hpf option: -cord, -double_size 128, -feedback, -fpe1, -fpe2, -fpe3, -fpe4, -gen_feedback, -om, -omp, -p1, and -pg.

The -check bounds option is compatible with the -hpf option only when the optional parameter *num* is set to 1.

When -hpf is used, the HPF target must be specified either by using the -hpf_target option or by setting the DECF90_HPF_TARGET environment variable. If neither is used, the driver will issue an error message and abort the compilation. See Section 3.50.5.

For More Information:

- On compiling an HPF global routine that will be linked with a main program that was not compiled with the -hpf option, see Section 3.68.
- On enabling conditional compilation for OpenMP, see Section 3.8.
- On using the -warn hpf option for syntax checking, see Section 3.100.

The following options are relevant to the -hpf option:

3.50.1 -assume bigarrays — Run-time Checking for Distributed Small Array Dimensions

*(UN*X only)* Specifying -assume bigarrays suppresses run-time checking for distributed small array dimensions. This allows for increased run-time performance and reduced compile time when using the -hpf option.

In programs compiled with both the -hpf and -assume bigarrays options, nearest-neighbor computations that reference small arrays will fail. An array is big enough if, for every dimension with BLOCK distribution, the shadow edge width is no bigger than the block size.

Specifying the -fast option sets the -assume bigarrays option.

The default is -assume nobigarrays.

3.50.2 -assume nozsize — Omit Zero-Sized Array Checking

*(UN*X only)* Specifying `-assume nozsize` suppresses run-time checking for zero-sized array sections. This allows for increased performance when using the `-hpf` option.

An array (or array section) is zero-sized when the extent of any of its dimensions takes the value zero or less than zero. When the `-hpf` option is specified, the compiler ordinarily inserts a series of checks to guard against irregularities (such as division by zero) in its internal computations that zero-sized arrays can cause. Depending on the particular application, these checks can cause noticeable (or even major) degradation of performance.

Specifying the `-fast` option sets the `-assume nozsize` option.

Avoid using the `-assume nozsize` option with `-hpf` when a program references any zero-sized arrays or array sections, because the resulting executable might fail to execute or might produce incorrect program results.

You can insert a run-time check into your program to ensure that a given line is not executed if an array or array section referenced there is zero-sized.

The default is `-assume zsize`.

3.50.3 -nearest_neighbor, -nearest_neighbor *num*, or -nonearest_neighbor — Nearest Neighbor Optimization

Use the `-nearest_neighbor` option *(TU*X only)* to enable the nearest neighbor parallel optimization. The compiler automatically determines the correct shadow-edge widths on an array-by-array, dimension-by-dimension basis.

This option is valid only if the `-hpf` option is specified.

Use the optional *num* field to specify the maximum width of the shadow edge, which limits how much extra storage the compiler can allocate for nearest-neighbor arrays.

When *num* is specified, the compiler will only recognize nearest-neighbor constructs that need shadow-edge widths less than or equal to the value of *num*.

If *num* is not specified, the compiler uses a *num* value of 10.

You can also set shadow-edge widths manually, using the !HPF$ SHADOW directive.

The `-nonearest_neighbor` option turns off the nearest-neighbor optimization. It is equivalent to specifying `-nearest_neighbor` with *nn* set to 0.

3.50.4 -show hpf — Show HPF Parallelization Information

The -show hpf options *(TU*X only)* show information about HPF parallelization by displaying the information to standard error and to the listing (if one is generated using the -V option). These options are valid only if the -hpf option is specified.

The -show option can take only one argument. However, the -show options can be combined by specifying -show multiple times. For example:

```
% f90 -hpf -hpf_target cmpi -show hpf_near -show hpf_punt foo.f90
```

Specifying -show hpf selects a subset of the messages generated by all of the following -show hpf_* options. Try using -show hpf first, with the following variations only when a more detailed listing is needed.

The following options apply:

-show hpf_all, -show hpfinfo, -show wsfinfo
The -show hpf_all option is the same as specifying all the other -show hpf_* options.

The -show wsfinfo option is the nonpreferred synonym of -show hpfinfo.

-show hpf_comm
This instructs the compiler to display information about statements that cause interprocessor communication to be generated.

-show hpf_indep
This instructs the compiler to display information about the optimization of loops marked with the INDEPENDENT directive.

-show hpf_nearest
This instructs the compiler to display information about arrays and statements involved in optimized nearest-neighbor computations.

-show hpf_punt
This instructs the compiler to display information about distribution directives that were ignored and statements that were not handled in parallel.

-show hpf_temps
This instructs the compiler to display information about temporaries that were created at procedure interfaces.

3.50.5 -hpf_target — Message Passing Protocol for Parallel Programs

This option specifies the message passing protocol to use between processors for programs running in parallel. The format is:

```
% -hpf_target cmpi | gmpi | smpi
```

If this option is used, -hpf must also be set.

The following options apply:

-hpf_target cmpi
Causes the compiler to target Compaq MPI, a version of Message Passing Interface (MPI) specifically tuned for Alpha systems. Compaq MPI is distributed as a Compaq layered product. Compaq MPI supports only Memory Channel clusters and shared-memory (SMP) machines.

For more information about Compaq MPI, see:

http://www.compaq.com/hpc/software/dmpi.html

-hpf_target gmpi
Causes the compiler to target generic Message Passing Interface (MPI), specifically MPICH 1.2.0 or other libraries compatible with that version of MPICH. MPICH is a public-domain implementation of the MPI specification that is available for many platforms. It can be obtained from:

http://www-unix.mcs.anl.gov/mpi/mpich/

MPICH supports many interconnection networks, including Ethernet, FDDI, and other hardware. Note: The use of the gmpi target is officially unsupported.

When you use this option, you should set the DECF90_GMPILIB environment variable to specify a path to the MPI library to link against. If you don't set the DECF90_GMPILIB environment variable, then you must specify the MPI library to link against on the command line.

-hpf_target smpi
Causes the compiler to target the Message Passing Interface (MPI) that comes installed on SC-series systems. It is used in conjunction with the SC's "RMS" software, which provides a set of commands for launching MPI jobs, scheduling those jobs on SC clusters, and other miscellaneous tasks.

3.51 -I — Remove Directory from Include Search Path

Use the `-I` option to prevent the search for `cpp` and `fpp` `#include` files or to prevent Compaq Fortran from searching for files specified in an INCLUDE statement in the standard directory `/usr/include`.

This option is not related to the Fortran USE statement.

3.52 -I*dir* — Add Directory for Module and Include File Search

Use the `-I`*dir* option to direct the search for `cpp` and `fpp` `#include` files, files specified in an INCLUDE statement, and module files (USE statement). The file names must not begin with a slash (/).

If you use this option, directories are searched in the following order:

1. The directory preceding the input file name on the command line
2. The directory (*dir*) specified by the `-I`*dir* option (see Section 3.52)
3. The standard directory `/usr/include`

You can repeat the `-I`*dir* option as many times as needed to specify multiple additional search directories.

To prevent the Compaq Fortran compiler from searching for include files in the `/usr/include` directory, use the `-noinclude` option.

For More Information:

- On creating and using module files, see Section 2.1.3.
- On creating and using include files, see Section 2.1.4.
- On recognized file name suffix characters and their meanings, see Section 2.1.1.

3.53 -i2, -i4, -i8, -integer_size *num* — Integer and Logical Data Size

Use the following options to control the size of INTEGER and LOGICAL declarations (where no kind parameter or size specifier is indicated).

For optimal performance on Alpha systems, use 4- or 8-byte integer or logical values instead of 2-byte values.

The default is `-i4` or `-integer_size 32`.

The following options apply:

-i2, -noi4, -integer_size 16
Specifying `-i2`, `-noi4`, or `-integer_size 16` (all are synonymous) makes the default integer and logical variables 2 bytes long. That is, INTEGER and LOGICAL declarations are treated as INTEGER*2 (KIND=2) and LOGICAL*2 (KIND=2).

-i4, -integer_size 32
Specifying `-i4` or `-integer_size 32` makes default integer and logical variables 4 bytes long. That is, INTEGER and LOGICAL declarations are treated as INTEGER*4 (KIND=4) and LOGICAL*4 (KIND=4).

-i8, -integer_size 64
Specifying `-i8` or `-integer_size 64` makes default integer and logical variables 8 bytes long. That is, INTEGER and LOGICAL declarations are treated as INTEGER*8 (KIND=8) and LOGICAL*8 (KIND=8).

3.54 -inline *keyword*, -noinline — Control Procedure Inlining

Specifying `-inline` *keyword* or `-noinline` controls the type of procedure calls that are inlined by the optimizer.

The following options apply:

-inline all
Inlines every call that can be inlined while still generating correct code, including:

- Statement functions
- Any procedures that Compaq Fortran expects will improve run-time performance with a likely significant increase in program size.
- Any other procedures that can possibly be inlined and generate correct code. Certain recursive routines are not inlined to prevent infinite expansion.

-inline speed
Inlines calls that will likely improve run-time performance, even where it might significantly increase the size of the program.

This option is meaningful only at optimization levels `-O1` and higher.

This is the default for optimization levels `-O4` and `-O5`.

-inline size
Inlines calls where inlining will not significantly increase program size, plus any additional calls that the compiler determines will improve run-time performance.

This option was previously called `-inline automatic` or `-inline space`.

This option is meaningful only at optimization levels `-O1` and higher.

-inline none, -inline manual, -noinline
Suppresses all inlining of routines. However, statement functions are always inlined.

All three options are synonymous.

This is the default for optimization levels `-O0`, `-O1`, `-O2`, and `-O3`.

For More Information:

- See Section 5.9.3, Controlling the Inlining of Procedures.
- See Chapter 5, Performance: Making Programs Run Faster.
- See Section 5.8, Optimization Levels: the -On Option.

3.55 -intconstant — Handling of Integer Constants

Specify this option to use Compaq Fortran 77 instead of Fortran 95/90 semantics to determine the kind of integer constants. If you don't specify `-intconstant`, Fortran 95/90 semantics are used.

With Fortran 77 semantics, all constants are kept internally by the compiler in the highest precision possible. For example, if you specify `-intconstant`, the compiler stores an integer constant of 14 internally as INTEGER(KIND=8) and converts the constant upon reference to the corresponding proper size. Fortran 95/90 specifies that integer constants with no explicit KIND are kept internally in the default INTEGER kind (KIND=4 by default).

Note that the internal precision for floating-point constants is controlled by the `-fpconstant` option, described in Section 3.43.

3.56 -K — Keep Temporary Files

Specifying `-K` requests that temporary files created by `cpp` or the Compaq Fortran compiler not be automatically deleted.

This option does not affect the naming of temporary files. To see the names and locations of the temporary files, use the `-v` option.

Specifying `-K` also creates, in the current working directory, one temporary file for each input source file specified. This file creation may not be desirable when compiling multiple source files (see Section 2.1.6).

All other temporary files reside in `/tmp`, unless the environment variable `TMPDIR` is set to indicate an alternate directory.

3.57 -L — Remove ld Directory Search Path

Use the `-L` option to prevent the linker from searching for libraries in the standard directories (see Section 3.58).

3.58 -L*dir* — Add Directory to ld Search Path

Use the `-L`*dir* option to specify the directory path *dir* as a search directory for `ld`, which is searched before the standard directories.

When you specify the `-non_shared` option, the following directories are searched:

- `/lib`
- `/usr/lib`
- `/usr/local/lib`

When searching for shared libraries (`-call_shared` or `-shared`), the following directories are searched:

- `/usr/shlib`
- `/usr/ccs/lib`
- `/usr/lib/cmplrs/cc`
- `/usr/lib`
- `/usr/local/lib`

3.59 -l*string* — Add Library Name to ld Search

Use the -l*string* option to specify additional search libraries for ld, using *string* as an abbreviation of the library name.

Using this option causes ld to search for unresolved references to run-time object files in the specified libraries, in addition to the appropriate Compaq Fortran libraries and related libraries (see Section 2.5.1).

This option should be placed after the file name(s) at the end of the command line.

You can use -l*string* multiple times to specify multiple search libraries. The order in which the multiple libraries are specified determines the search order used by ld.

For example, specifying -lgraph causes ld to search the standard library directories (and any specified using the -L*dir* option) for the library libgraph.

To view libraries accessed during compile and link operation, use the -v option, described in Section 3.96.

For More Information:

- On the standard directories searched by ld, see Section 3.58.
- On the standard list of library names searched when using the f90 command, see Section 2.5.1.
- On recognized file name suffix characters and their meanings, see Section 2.1.1.
- On specifying and using archive or shared libraries, see Section 2.5.
- On requirements and other information related to creating a shared library, see Section 2.6.

3.60 -machine_code

See Section 3.82, -show keyword, -machine_code — Control Listing File Content.

3.61 -math_library *keyword* — Fast or Accurate Math Library Routines

The -math_library *keyword* option lets you choose which math library routines to use.

The default is -math_library accurate.

The following options apply:

-math_library accurate
Specifies that the compiler is to select the version of the math library routine that provides the most accurate result for mathematical intrinsic functions.

For certain ranges of input values, the selected routine may execute more slowly than if you used -math_library fast.

The standard math library routines are designed to obtain very accurate "near correctly rounded" results and provide the robustness needed to check for IEEE exceptional argument values, rather than achieve the fastest possible run-time execution speed. Using -math_library accurate allows user control of arithmetic exception handling with the -fpe*n* option and the for_set_fpe routine.

-math_library fast
Specifies that the compiler is to select the version of the math library routine that provides the highest execution performance for certain mathematical intrinsic functions, such as EXP and SQRT.

For certain ranges of input values, the selected routine may not provide a result that is as accurate as -math_library accurate provides.

This option is set by default if you use the -fast option.

Using -math_library fast allows certain math library functions to get significant performance improvements when the applicable intrinsic function is used.

If you specify -math_library fast, the math library routines do not necessarily check for IEEE exceptional values and the -fpe*n* option and calls to the for_set_fpe routine are ignored.

When you use -math_library fast, you should carefully check the calculated output from your program. Check the program's calculated output to verify that it is not relying on the full fractional accuracy of the floating-point data

type[1] to produce correct results or producing unexpected exceptional values (exception handling is indeterminate).

Programs that do not produce acceptable results with `-math_library fast` and single-precision data might produce acceptable results with `-math_library fast` if they are modified (or compiled) to use double-precision data.

The specific intrinsic routines that have special fast math routines depend on the version of the Compaq Tru64 UNIX operating system in use. Allowed error bounds vary with each routine.

For More Information:

- On controlling arithmetic exception handling, including using the `-fpe`*n* option, see Section 3.44 and Chapter 14.
- On requesting double-precision data during compilation for REAL data declarations (`-real_size` option), see Section 3.78.
- On native IEEE floating-point formats, see Section 9.4.

3.62 -mixed_str_len_arg — Specify Length of Character Arguments

Use the `-mixed_str_len_arg` option to tell the compiler that the hidden length passed for a character argument is to be placed immediately after its corresponding character argument in the argument list.

The default is `-nomixed_str_len_arg`, which places the hidden lengths in sequential order at the end of the argument list.

3.63 -module *directory* — Specify Directory for Creating Modules Files

The `-module` *directory* option tells the compiler to create module files in the specified *directory* instead of the current directory.

[1] Single-precision S_float, double-precision T_float, and extended precision X_float data types provide 24, 53, and 113 bits respectively for the normalized fractional part (see Section 9.4.)

3.64 -mp — Enable Parallel Processing Using Directed Decomposition

Use the -mp option *(TU*X only)* to enable parallel processing using directed decomposition. Parallel processing is directed by inserting parallel directives using the !$PAR prefix in your source code. The compiler recognizes these directives only if you specify the -mp option. This kind of parallel processing is intended for shared-memory multiprocessor systems.

To enable parallel processing with with !$OMP directives, use the -omp compiler option.

To enable parallel processing across clusters of servers or workstations with !HPF$ directives, use the -hpf compiler option.

The -mp option sets the -automatic option.

The default is -nomp.

For More Information:

- See Chapter 6, Parallel Compiler Directives and Their Programming Environment.
- See the *Compaq Fortran Language Reference Manual*.
- See Section 3.74.
- See Section 3.50.

3.65 -names *keyword* — Case Control of Source and External Names

The -names *keyword* option controls how the case sensitivity of letters in source code identifiers and external names is handled.

The naming convention applies whether names are being defined or referenced.

The default is -names lowercase.

The following options apply:

-names as_is, -U
Causes the compiler to distinguish case differences in identifiers and to preserve the case of external names.

The -names as_is and -U options are synonymous.

-names lowercase
Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase.

-names uppercase
Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase.

3.66 -noaltparam — Alternative PARAMETER Syntax

Use the `-noaltparam` option to disallow the alternate nonstandard syntax for PARAMETER statements.

The nonstandard PARAMETER statement syntax is:

```
PARAMETER par1=exp1 [, par2=exp2] ...
```

This form has no parentheses surrounding the list, and the form of the constant, rather than implicit or explicit typing, determines the data type of the variable.

The default is `-altparam`, which allows use of the nonstandard syntax.

3.67 -nofor_main — Allow Non-Fortran Main Program

Use the `-nofor_main` option when the main program is not written in Fortran. For example, if the main program is written in C and calls a Compaq Fortran subprogram, specify `-nofor_main` when compiling the program with the `f90` command. Specifying `-nofor_main` prevents linking `for_main.o` into applications.

If you omit `-nofor_main`, the main program must be a Fortran program.

The default is `-for_main`.

3.68 -nohpf_main, -nowsf_main — Compile HPF Global Routine for Nonparallel Main Program

*(TU*X only)* Use the `-nohpf_main` option (the `-nowsf_main` option is synonymous) to specify that the HPF global routine being compiled will be linked with a main program that was not compiled with the `-hpf` option. The main program can be a Fortran program compiled without `-hpf`, or it can be written in an entirely different language.

3.69 -noinclude — Omit Standard Directory Search for INCLUDE Files

Specifying -noinclude prevents the compiler from searching in the /usr/include directory for files specified in an INCLUDE statement. This option does *not* apply to the directories searched for module files or preprocessor #include files.

To request that the cpp preprocessor not search for #include files in the /usr/include directory, use the -I option (see Section 3.51).

3.70 -norun — Do Not Run the Compiler

Specifing the -norun option directs the driver not to execute the compiler and other phases of the process. If you use this option with the -v option, described in Section 3.96, you can see what would have been executed.

The default is for the driver to execute the compiler and other phases of the process.

3.71 -o *output* — Name Output File

When you specify -c with -o *output*, this names the object file *output* instead of *source-file-name*.o in the current working directory.

If you omit -c and specify -o *output*, this names the executable program file *output* instead of a.out in the current working directory.

An already existing a.out file is unaffected by this command.

For More Information:

- On output files and their names, see Section 2.1.5.
- On using multiple input files, see Section 2.1.6.

3.72 -O0, -O1, -O2, -O3, -O4 or -O, -O5 — Specify Optimization Level

Use the -O0, -O1, -O2, -O3, -O4 (same as -O), and -O5 options to specify the level of optimization performed during compilation.

The default level of optimization is -O4 unless you specify the -g2, -g, or -gen_feedback option (in which case the default is -O0).

At optimization levels lower than -O4, the compiler issues “uninitialized variable” warnings.

In most cases, the higher the level of optimization you specify, the faster the program will execute. However, the faster execution speeds that result from using -O3 or higher usually produce larger object files and longer compile times.

The following options apply:

-O0
Disables all optimizations. Does not check for unassigned variables.

If you specify -g2 or -g, this is the default.

-O1
Enables local optimizations and recognition of common subexpressions. Optimizations include integer multiplication and division expansion using shifts. The call graph determines the order of compilation of procedures.

-O2
Enables global optimization and all -O1 optimizations. This global optimization includes code motion, strength reduction and test replacement, split-lifetime analysis, code scheduling, and inlining of arithmetic statement functions.

-O3
Enables global optimizations that improve speed (at the cost of increased code size) and all -O2 optimizations. Optimizations include:

- Loop unrolling (also set by -unroll, described in Section 3.94)
- Prefetching
- Code replication to eliminate branches

-O4, -O
Enables interprocedure analysis and automatic inlining of small procedures (with heuristics limiting the amount of extra code), software pipelining (also set by -pipeline, described in Section 3.76), and all -O3 optimizations.

This is the default. However, if you specify -g2 or -g or -gen_feedback, the default becomes -O0.

-O5
Enables loop transformation optimizations (also set by -transform_loops, described in Section 3.89), all -O4 optimizations, and other optimizations, including byte vectorization and insertion of additional NOPs (No Operations) for alignment of multi-issue sequences. (See also Section 3.84.)

To determine whether using -O5 benefits your particular program, you should time program execution for the same program (or subprogram) compiled at levels -O4 and -O5.

For More Information:

- On the effects of the -O5 option, see Section 3.89 and Section 3.76.
- On limiting loop unrolling with optimization level -O3 or higher (-unroll *num*), see Section 3.94.
- On speculative execution optimization, see Section 3.84.
- On timing program execution, see Section 5.2.
- On the related -fp_reorder option, see Section 3.12.
- On the related -fast option, see Section 3.40.
- On improving and measuring run-time performance, see Chapter 5.
- On the optimizations performed at each level, see Section 5.8.

3.73 -om — Request Nonshared Object Optimizations

*(TU*X only)* Use the -om option with the -non_shared option to request certain code optimizations after linking, including NOP (No Operation) removal, .lita removal, and reallocation of common symbols. This option also positions the global pointer register so the maximum addresses fall in the global-pointer window.

Pass -om options to the linker using the -WL,*arg* form:

- -WL,-om_compress_lita

  Removes unused .lita entries after optimization, and then compresses the .lita section.

- -WL,-om_dead_code

  Removes dead code (unreachable instructions) generated after applying optimizations. The .lita section is not compressed.

- -WL,-om_no_inst_sched

  Turns off instruction scheduling.

- -WL,-om_no_align_labels

  Turns off alignment of labels. Normally, the -om option aligns the targets of all branches on quadword boundaries to improve loop performance.

- -WL,-om_Gcommon,*num*

Sets the size threshold of common symbols. Every common symbol whose size is less than or equal to *num* will be allocated close to each other. This option can be used to improve the probability that the symbol can be accessed directly from the global pointer register. Normally, `-om` tries to collect all common symbols together.

For more information, see your operating system documentation.

3.74 -omp — Enable OpenMP Parallel Processing Using Directed Decomposition

*(UN*X only)* Use the `-omp` option to enable parallel processing using directed decomposition following the OpenMP application program interface (API). Parallel processing is directed by inserting !$OMP directives in your source code. This kind of parallel processing is intended for shared-memory multiprocessor systems.

The !$OMP directives are described in Section 6.1.2, Summary Descriptions of OpenMP Fortran API Compiler Directives.

The `-omp` option sets the `-automatic` option.

The default is `-noomp`.

For More Information:

- Chapter 6, Parallel Compiler Directives and Their Programming Environment.
- See the *Compaq Fortran Language Reference Manual*.

3.75 -pad_source — Pad Short Source Records with Spaces

For fixed-form source files, specify the `-pad_source` option to request that source records shorter than the statement field width are to be padded with spaces on the right, out to the end of the statement field. This affects the interpretation of character and Hollerith literals that are continued across source records.

The default is `-nopad_source`. This causes a warning message to be displayed if a character or Hollerith literal that ends before the statement field ends is continued onto the next source record. To suppress this warning message, specify the `-warn nousage` option.

Specifying `-pad_source` can prevent warning messages associated with `-warn usage`.

3.76 -pipeline — Activate Software Pipelining Optimization

Specifying -pipeline (or -O4 or -O5) activates the software pipelining optimization. The software pipelining optimization applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution.

Software pipelining is a subset of the optimizations activated by -O4 or -O5. Instead of specifying both -pipeline and -transform_loops, you can specify -O5.

This optimization is not performed at optimization levels below -O2.

You must specify -nopipeline if you want this type of optimization disabled and you are also specifying -O4 or -O5.

For More Information:

- On details about using this option, see Section 5.8.6, Software Pipelining.
- On the -O4 and -O5 options, see Section 3.72.

3.77 -p0, -p1 or -p, and -pg — Profiling Support

Nonparallel programs (with the -hpf option omitted) and *(TU*X only)* parallel HPF programs (-hpf option specified) use different profiling tools, which need different profiling options. Profiling information identifies those parts of your program where improving source code efficiency would most likely improve run-time performance.

If you omit the -hpf option, you can use the prof and pixie *(TU*X only)* tools if you specify the -p0 and -p1 or -p options to control the level of profiling support provided during compilation. When you specify -hpf, omit the -p0, -p1, and -p options.

The default is -p0.

The following options apply:

-p0
Does not permit profiling. If loading occurs, the standard run-time startup routine (crt0.o) is used, and profiling libraries are not searched.

-p1, -p
Sets up profiling by periodically sampling the value of the program counter for use with the postprocessor prof(1).

This option only affects loading. When loading occurs, this option replaces the standard run-time startup routine option with the profiling run-time startup routine (`mcrt0.o`) and searches the level 1 profiling library (`libprof1`).

When profiling occurs, the startup routine calls `monstartup`(3) and produces the file `mon.out`, which contains execution-profiling data for use with the postprocessor `prof`(1) command. (See also `monitor`(3).)

If you specify this option, you should also specify `-g1` or higher.

-pg
Sets up profiling for `gprof`(1), which produces a call graph showing the execution of the program. With this option, the standard run-time startup routine is replaced by the `gcrt0.o` routine, and `ld`(1) inserts calls to `_mcount` at each entry label.

When programs are linked with the `-pg` option and then run, these files produced:

`gmon.out` contains a dynamic call graph and profile.
`gmon.sum` contains a summarized dynamic call graph and profile.

To display the output, run `gprof` on the `gmon.out` file.

For More Information:

- On using profiling tools `prof` and `pixie` *(TU*X only)*, see Section 5.3.
- On the `-gen_feedback`, `-feedback` *file*, and `-cord` options, *(TU*X only)* see Section 3.41.

3.78 -real_size *number*, -r8, -r16 — Floating-Point Data Size

Use these options to control the size of REAL and COMPLEX declarations that do not have an explicit KIND parameter or size specifier.

The default is `-real_size 32`.

The following options apply:

-real_size 32
Defines:

- REAL declarations, constants, functions, and intrinsics as REAL*4
- COMPLEX declarations, constants, functions, and intrinsics as COMPLEX*8

-real_size 64, -r8
Defines:

- REAL declarations, constants, functions, and intrinsics as DOUBLE PRECISION (REAL*8)
- COMPLEX declarations, constants, functions, and intrinsics as DOUBLE COMPLEX (COMPLEX*16)

-real_size 128, -r16
defines:

- REAL and DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL*16
- COMPLEX and DOUBLE COMPLEX declarations, constants, functions, and intrinsics as COMPLEX*32

If you omit `-real_size_64` and `-real_size 128`, then:

- REAL declarations, constants, functions, and intrinsics are defined as REAL*4 (KIND=4).
- DOUBLE PRECISION declarations, constants, functions, and intrinsics are defined as REAL*8 (KIND=8).
- COMPLEX declarations, constants, functions, and intrinsics are defined as COMPLEX*8 (KIND=4).
- DOUBLE COMPLEX declarations, constants, functions, and intrinsics are defined as COMPLEX*16 (KIND=8).

For More Information:

- On data types, see Chapter 9.
- On intrinsic functions, see the *Compaq Fortran Language Reference Manual*.

3.79 -recursive — Request Recursive Execution

This option compiles all FUNCTION and SUBROUTINE procedures for possible recursive execution.

This option:

- Changes the default allocation class for all local variables from STATIC to AUTOMATIC, except for variables that are data-initialized or named in a SAVE statement, or for variables declared as STATIC.

- Permits references to a routine name from inside the routine.

A subprogram declared with the RECURSIVE keyword is always recursive (whether you specify or omit the `-static` option).

Variables declared with the AUTOMATIC statement or attribute always use stack-based storage for all local variables (whether you specify or omit the `-recursive` or `-automatic` options).

Specifying `-recursive` sets `-automatic` (puts local variables on the run-time stack).

The default is `-norecursive`.

3.80 -reentrancy *keyword* — Control Use of Threaded Run-Time Library

*(UN*X only)* The `-reentrancy` *keyword* option specifies whether code generated for the main program and any Fortran procedures it calls will be relying on threaded or asynchronous reentrancy.

To use the threaded libraries, also specify the `-threads` option (see Section 3.88).

The default is `-reentrancy none`. (The option `-noreentrancy` is the same as `-reentrancy none`.)

The following options apply:

-reentrancy asynch
Tells the Compaq Fortran run-time library (RTL) that the program may contain asynchronous handlers that could call the RTL. This causes the RTL to guard against asynchronous interrupts inside its own critical regions.

-reentrancy none
Tells the Compaq Fortran RTL that the program will not be relying on threaded or asynchronous reentrancy. Therefore, the RTL will not guard against such interrupts inside its own critical regions.

-reentrancy threaded
Tells the Compaq Fortran RTL that the program is multithreaded, such as programs using the POSIX threads library. This causes the RTL to use thread locking to guard its own critical regions.

3.81 -S — Create Assembler File

Specifying `-S` creates an assembler file from the compiled source. The name of the assembler file is the base name of the source file with a `.s` file suffix. Linking does not occur.

3.82 -show *keyword*, -machine_code — Control Listing File Content

Use the `-show` *keyword* options in conjunction with the `-V` option to control the amount of information in a listing file.

The following options apply:

-show code, -machine_code
Specifying `-show code` or `-machine_code` includes a machine-language representation of the compiled code if a listing file is being generated. This machine language cannot be assembled.

The default is `-show nocode` or `-nomachine_code`.

-show hpf
*(TU*X only)* When used with the `-hpf` *num* option, specifying `-show hpf` shows information about HPF parallelization. For more information, see Section 3.50.4, -show hpf — Show HPF Parallelization Information.

-show include
Specifying `-show include` lists the contents of any text file specified with INCLUDE in the source file if a listing is generated.

The default is `-show noinclude`.

-show nomap
Specifying `-show nomap` excludes information about the symbols used in the source program if a listing is generated.

The default is `-show map`.

If you omit these options, the listing file contains the minimum amount of information.

For More Information:

- On requesting a listing file, see Section 3.95.
- On examples and explanations of the various listing options, see Appendix C.

3.83 -source_listing — Create a Source Listing File

Specifying `-source_listing` tells the compiler to create a listing file of the source program. The file also contains compiler-generated information such as that specified by the `-V` option. The name of the listing file is the base name of the source file with a `.lis` suffix.

The default is `-nosource_listing`.

3.84 -speculate *keyword* — Speculative Execution Optimization

Specifying the `-speculate all` option *(TU*X only)* or the `-speculate by_routine` option *(TU*X only)* requests the compiler to perform speculative execution optimization on all routines in the application.

Speculation occurs when a conditionally executed instruction is moved to a position before a test instruction so that the moved instruction is then executed unconditionally. This reduces instruction latency stalls to improve run-time performance for certain applications or routines.

Speculative execution affects code most noticeably at optimization level `-O3` and higher.

Performance improvements may be reduced because the run-time system must dismiss exceptions caused by speculative instructions. For certain applications, longer execution times may result. To determine whether using `-speculate all` or `-speculate by_routine` benefits your particular program, you should time program execution for the same program compiled with `-speculate by_routine` or `-speculate all` with `-speculate none` (default).

Any exception (for example, SIGSEGV, SIGBUS, or SIGFPE), anywhere in the entire program, is assumed to be speculative. All of these exceptions are quietly dismissed without calling any user-mode signal handler. If a module is compiled using `-speculate all`, it cannot be linked with any other module or library that does its own exception processing.

Since speculation turns off some run-time error checking, this option should not be used while debugging or while testing for errors.

The following options apply:

-speculate all
Perform speculative execution optimization on all routines in the program. All exceptions within the entire program are dismissed without calling any user-mode signal handler or reporting exceptions.

If a compilation unit is compiled with `-speculate all`, then it may not be linked with any other object or library that does its own exception processing. Do *not* use `-speculate all` if your program does any of the following:

- Generates any exceptional values other than underflow values
- Uses a user-supplied signal handler
- Was compiled with any of the `-check` *keyword* options

-speculate by_routine
Specifies that speculative execution optimization should be performed on all routines in the current compilation unit (set of routines being compiled). However, other compilation units in the application will not be affected.

All exceptions within the routines being compiled with `-speculate by_routine` are quietly dismissed without calling any user-mode signal handler, but the object files created can be linked with other objects or libraries that perform exception processing. For example, using `-speculate by_routine` allows the Compaq Fortran Run-Time Library to report exceptions.

-speculate none, -nospeculate
Suppresses all speculative execution optimization.

The `-speculate none` and `-nospeculate` options are synonymous.

This is the default.

3.85 -std, -std90, -std95 — Perform Fortran Standards Checking

Use these options to request that the compiler produce warnings for syntax that is not standard in the language.

If your command line includes any of these options, then the compiler ignores `-align dcommons` and `-align sequence`.

The default is `-nostd`.

The options are:

-std
Produces warnings for things that are not standard.

When `-std` is specified, the compiler being used determines the standard checked.

For example, on Tru64 UNIX systems, the Compaq Fortran compiler recognizes the following equivalencies:

`f95 -std` is the same as `f90 -std95`
`f90 -std` is the same as `f90 -std90`

On Linux systems, the Compaq Fortran compiler recognizes the following equivalency:

`fort -std` is the same as `fort -std95`

Source statements that do not conform to Fortran 90 or Fortran 95 language standards are detected by the Compaq Fortran compiler under the following circumstances:

- The statements contain ordinary syntax and semantic errors.
- A source program containing nonconforming statements is compiled with the `-std` option.

Given these circumstances, the compiler is able to detect *most* instances of nonconforming usage. It does not detect all instances because the `-std` options do not produce checks for all nonconforming usage at compile time. In general, the unchecked cases of nonconforming usage arise from the following situations:

- The standard violation results from conditions that cannot be checked at compile time.
- The compile-time checking is prone to false alarms.

Most of the unchecked cases occur in the interface between calling and called subprograms. However, other cases are not checked, even within a single subprogram.

The following items are known to be unchecked:

- Use of a data item prior to defining it
- Use of the SAVE statement to ensure that data items or common blocks retain their values when reinvoked
- Association of character data items on the right and left sides of character assignment statements

- Mismatch in order, number, or type in passing actual arguments to subprograms with implicit interfaces
- Association of one or more actual arguments with a data item in a common block when calling a subprogram that assigns a new value to one or more of the arguments

On Tru64 UNIX systems only, the `-std` options interact with the `-hpf` option (requests generation of parallel HPF code) as follows:

- If you omit both `-std` and `-hpf`, HPF directives are checked, but not processed.
- If you omit `-std` and specify `-hpf`, HPF directives are checked and processed.
- If you specify `-std` and omit `-hpf`, HPF directives are ignored (treated as comments and not checked).

You should not specify both `-std` and `-hpf`.

-std90
Produces warnings for things that are not standard in the Fortran 90 language. This includes:

- Fortran 90 standard-conforming statements that become nonstandard due to the way in which they are used. Data type information and statement locations are considered when determining semantic extensions.
- For fixed-format source files, lines that use tab formatting.

-std95
Produces warnings for things that are not standard in the Fortran 95 language. This includes all the checks performed by the `-std90` option with the following exceptions:

- The FORALL statement is part of Fortran 95; thus it is not flagged as an extension by the `-std95` option
- User-defined PURE functions are part of Fortran 95; thus they are also not flagged as extensions.
- The following features that were obsolescent in F90 are deleted in Fortran 95 (`-std95` flags them as deleted) but the Compaq Fortran compiler fully supports them:
  - REAL and DOUBLE PRECISION DO variables
  - Branching to an ENDIF statement from outside the paired IF statement

- PAUSE statement
- ASSIGN statement, assigned GOTO, and assigned FORMAT statements
- H edit descriptor

For More Information:

On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

3.86 -synchronous_exceptions — Report Exceptions More Precisely

This option causes the compiler to generate TRAPB instructions after every floating-point instruction, resulting in precise exception reporting.

To use this option, you must also enable `-fpe0`.

This is a very expensive but effective way to synchronize the instruction stream containing floating-point exceptions so the failing instruction can be accurately located by the debugger or a handler. You should use this option only when debugging a specific problem, such as locating the source of an exception.

If you omit the `-synchronous_exceptions` option and `-fpe0` is in effect, exceptions can be reported one or more instructions *after* the instruction that caused the exception. If you specify `-fpe1`, `-fpe2`, `-fpe3`, or `-fpe4`, exceptions are reported precisely (this is the same as specifying `-synchronous_exceptions`).

The default is `-nosynchronous_exceptions`.

3.87 -syntax_only — Do Not Create Object File

The `-syntax_only` option specifies that the source file will be checked only for correct syntax. No code is generated, no object file is produced, and some error checking done by the optimizer is bypassed (for example, checking for uninitialized variables).

This option lets you do a quick syntax check of your source file.

The default is `-nosyntax_only`.

3.88 -threads, -pthread — Link Using Threaded Run-Time Library

*(UN*X only)* Specifying the -threads or -pthread option (they are synonymous) requests that the linker use threaded libraries. This is usually used with the -reentrancy threaded option (see Section 3.80).

3.89 -transform_loops — Activate Loop Transformation Optimizations

Specifying -transform_loops (or -O5) activates a group of loop transformation optimizations that apply to array references within loops. These optimizations can improve the performance of the memory system and usually apply to multiply nested loops.

The loop transformation optimizations are a subset of optimizations activated by -O5. Instead of specifying both -pipeline and -transform_loops, you can specify -O5.

To determine whether using -transform_loops benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without loop transformation optimizations (such as with -transform_loops and -notransform_loops).

For More Information:

- On the -O5 option, see Section 3.72.
- On loop transformations, see Section 5.8.7, Loop Transformation.

3.90 -tune *keyword* — Specify Alpha Processor Implementation

Use the -tune *keyword* option to specify the types of processor-specific instruction tuning for implementations of the Alpha architecture.

Regardless of the setting of -tune *keyword* option you use, the generated code runs correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can improve run-time performance; it is also possible that code tuned for a specific Alpha processor may run more slowly on another Alpha processor.

If you omit -tune *keyword*, -tune generic is used.

When -arch *name* is specified and no -tune is specified, the following occurs: if -fast is specified, then -tune is *host* and -arch is *name*; if -fast is not specified, then both -tune and -arch are *name*.

If you set -tune to an architecture that has fewer features than what -arch specifies, the compiler will reset -tune to the same architecture as -arch.

The following options apply:

-tune generic
Generates and schedules code that will execute well for all implementations of the Alpha architecture. This provides generally efficient code for those cases where different processor generations are likely to be used.

This is the default.

-tune host
Generates and schedules code optimized for the implementation of the Alpha architecture in use on the system being used for compilation.

-tune ev4
Generates and schedules code optimized for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture.

-tune ev5
Generates and schedules code optimized for the 21164 implementation of the Alpha chip. This implementation of the Alpha architecture is faster and more recent than the implementations of the Alpha architecture associated with -tune ev4 (21064, 21064A, 21066, and 21068).

-tune ev56
Generates and schedules code optimized for some 21164 Alpha architecture implementations that use the byte and word manipulation instruction extensions of the Alpha architecture.

-tune pca56
Generates and schedules code optimized for the 21164PC Alpha architecture implementation that uses the byte and word manipulation instruction extensions and multimedia instruction extensions.

-tune ev6
Generates and schedules code optimized for the 21264 Alpha architecture implementations that use the byte and word manipulation instruction extensions, multimedia instruction extensions, and square root and floating-point convert extensions.

-tune ev67
Generates and schedules code optimized for the 21264A Alpha architecture implementations that use the byte and word manipulation instruction extensions, multimedia instruction extensions, square root and floating-point convert extensions, and count extensions.

For More Information:

- On improving and measuring run-time performance, see Chapter 5.

3.91 -U — Activates Case Sensitivity

This option causes the compiler to distinguish between uppercase and lowercase letters in identifiers and external names.

The `-U` option and `-names as_is` option are synonymous.

3.92 -U*name* — Undefine Preprocessor Symbol Name

Specifying `-U`*name* tells `-cpp` to remove the definition of *name*, such as a predefined symbol. Predefined preprocessor symbols are defined in Section 3.31.

3.93 -u

This option causes the compiler to produce messages about undeclared symbols.

The `-u` option is the same as `-warn declarations`.

3.94 -unroll *num* — Specify Number for Loop Unroll Optimization

The `-unroll` *num* option sets the depth of loop unrolling done by the optimizer to *num*. *num* must be an integer in the range 0 through 16.

Specify this option only with `-O3` or higher optimization levels, at which loop unrolling occurs.

If you omit `-unroll` *num* or specify `-unroll 0`, the optimizer determines how many times loops are unrolled. Unless you specify a value, the optimizer will choose an unroll amount that minimizes the overhead of prefetching while also limiting code size expansion.

The option `-nounroll` is not allowed.

For More Information:

- On loop unrolling optimization, see Section 5.8.4.1.
- On software pipelining, which is related to loop unrolling, see Section 3.76.
- On options related to optimization levels, see Section 3.72.

3.95 -V — Create Listing File

Specifying `-V` creates a listing of the source file with various compile-time information appended. The name of the listing file is the base name of the file with the `.l` suffix.

A file name whose suffix is `.l` may conflict with `lex`. If you want the listing file to have the suffix `.lis`, then use the option `-source_listing`.

If you expect your program to get compilation errors, use `-V` to request a separate listing file or use the `error` command to insert the error messages into the appropriate place in your source program.

Using a listing file provides slightly more information and includes the column pointer (1) that indicates the exact part of the line that caused the error. Especially for large files, consider obtaining a printed copy of the listing file you can reference while editing the source file.

If you use `-V` without other options (such as `-show code`), the listing file will not show the code of included source files or machine code and will not include a cross-reference table.

If a source line of length 1 contains a form-feed character, the source code listing begins a new page with the following line; the line containing the form-feed does not appear.

If a source line of length greater than 1 contains a form-feed character, that line is printed but the form-feed character is ignored (does *not* generate a new page).

Any other nonprinting ASCII characters encountered in source files are replaced by a space character, and a warning message appears.

If you compile several source files together and specify `-V`, a single listing file (using the name of the first input file) is created.

If you compile several source files one at a time and specify `-V`, multiple listing files are created.

The default is `-noV`.

For More Information:

- On the options that control the contents of the listing file, see Section 3.82.
- On sample listing file content, see Appendix C.

3.96 -v — Verbose Command Processing Display

Specifying `-v` displays the preprocessor (if requested), compiler, and linker passes as they execute, with their arguments and their input and output files, as well as final resource usage in the C shell `time` command format. The Compaq Fortran compiler is `decfort90`, the C compiler is `cc` on Tru64 UNIX systems but `ccc` on Linux systems, the linker is `ld`, and so forth.

The default is `-nov`, which does not display any command processing information.

3.97 -version, -what — Show Compaq Fortran Version Information

Specifying the `-version` (or `-what`) option prints the version information of the Fortran driver and compiler.

If `-version` appears alone on the command line, the compiler is not executed.

The default is `-noversion`.

3.98 -vms — OpenVMS Fortran Compatibility

Specifying `-vms` causes the run-time system to behave like Compaq Fortran on OpenVMS Alpha systems and VAX systems (VAX FORTRAN) in the following ways:

- Certain defaults

  In the absence of other options, `-vms` sets the `f90` defaults as `-check format` and `-check output_conversion`.

- Alignment

  The `-vms` option does not affect the alignment of fields in records or items in common blocks. Use `-align norecords` to pack fields of records on the next byte boundary for compatibility with Compaq Fortran on OpenVMS systems.

- Carriage control default

If -vms -ccdefault default is specified, carriage control defaults to FORTRAN if the file is formatted and the unit is connected to a terminal. See Section 3.21.

- INCLUDE qualifiers

  /LIST and /NOLIST are recognized at the end of the file name in an INCLUDE statement at compile time (see Section 2.1.4).

  If the file name in the INCLUDE statement does not specify the complete path, the path used is the current directory.

  Note that if -vms is not specified, the path used is the directory where the file that contains the INCLUDE statement resides.

- Quotation mark character

  A quotation mark (") character is recognized as starting an octal constant ("0..7) instead of a character literal ("...").

- Deleted records in relative files

  When a record in a relative file is deleted, the first byte of that record is set to a known character (currently '@'). Attempts to read that record later result in ATTACCNON errors. The rest of the record (the whole record, if -vms is not specified) is set to nulls for unformatted files and spaces for formatted files.

- ENDFILE records

  When an ENDFILE is performed on a sequential unit, an actual 1-byte record containing a Ctrl/Z is written to the file. If -vms is not specified, an internal ENDFILE flag is set and the file is truncated.

  The -vms option does not affect ENDFILE on relative files: these files are truncated.

- Implied logical unit numbers

  The -vms option enables Compaq Fortran to recognize certain environment variables at run time for ACCEPT, PRINT, and TYPE statements and for READ and WRITE statements that do not specify a unit number (such as READ (*,1000)). See Section 7.5.7, Using Environment Variables.

- Treatment of blanks in input

  The -vms option causes the defaults for the keyword BLANK in OPEN statements to become 'NULL' for an explicit OPEN and 'ZERO' for an implicit OPEN of an external or internal file. For more information, see the description of the OPEN statement in the *Compaq Fortran Language Reference Manual*.

- OPEN statement effects

  Carriage control defaults to FORTRAN if the file is formatted, and the unit is connected to a terminal (checked by means of `isatty(3)`). Otherwise, carriage control defaults to LIST.

  The `-vms` option affects the record length for direct access and relative organization files. The buffer size is increased by 1 to accommodate the deleted record character.

- Reading deleted records and ENDFILE records

  The run-time direct access READ routine checks the first byte of the retrieved record. If this byte is `'@'` or NULL ("\0"), then an ATTACCNON error is returned.

  The run-time sequential access READ routine checks to see if the record it just read is one byte long and contains a Ctrl/Z. If this is true, it returns EOF.

The default is `-novms`.

For More Information:

- On alignment, see Section 3.3 and Section 5.4.

3.99 -Wl,-*xxx* — Pass Specified Option to ld

The `-Wl,-`*xxx* option allows you to pass an option (specified by *-xxx*) directly to the `ld` linker.

For example, to set the linker `-taso` option *(TU*X only)* to help port 32-bit programs that assume addresses can be stored into 32-bit variables, specify:

```
-Wl,-taso
```

If the *-xxx* option takes an argument *yyy*, include *yyy* after the option separated by a comma.

For example:

```
-Wl,-xxx,yyy
```

results in the passing of

```
-xxx yyy
```

to `ld`.

For another example, if you want to specify the linker option *-VS 3*, then a possible command is:

```
% f90 -Wl,-VS,3 test123.f90
```

For More Information:

On linker options, see ld(1).

3.100 -warn *keyword*, -u, -nowarn, -w, -w1 — Warning Messages and Compiler Checking

You can prevent the display of some or all warning messages and request that additional compile-time checking be performed (can issue additional warning messages):

- The following options prevent the display of warning messages:

  ```
  -warn noalignments
  -warn nogeneral
  -warn nouncalled
  -warn nouninitialized
  -warn nounused
  -warn nousage
  -nowarn
  -w
  ```

- The `-warn declarations` (or `-u`) and `-warn argument_checking` options request that additional checking be performed and can display additional warning messages.

If you specify `-syntax_only`, some warning options are ignored (see Section 3.87).

The following options apply:

-warn noalignments

Disables warning messages about data that is not naturally aligned.

The default is `-warn alignments`.

To control alignment of common blocks, derived-type structures, and record structures, specify the `-align` *keyword* options.

-warn argument_checking

Enables warnings about mismatched procedure arguments. Specifying `-warn argument_checking` applies to calls with an implicit interface (such as routines declared as EXTERNAL).

The default is -warn noargument_checking.

When an explicit interface is present between calling and called procedures, warning messages about argument mismatches are reported whether or not you specify -warn argument_checking.

-warn declarations, -u
Sets the default type of a variable as undefined (IMPLICIT NONE), which enables warnings about any undeclared symbols. This behavior differs from default Fortran 95/90 rules.

The default is -warn nodeclarations.

-warn nogranularity
Disables warnings when the compiler cannot generate code for a requested granularity.

The default is -warn granularity.

-warn hpf
*(TU*X only)* Tells the compiler to do both syntax and semantics checking on HPF directives.

The default vis -warn nohpf, unless -hpf is specified, in which case -warn hpf is assumed.

-warn ignore_loc
Enables warnings when %loc is stripped from an argument.

The default is -warn noignore_loc.

-warn truncated_source
Enables warnings at compile time about source characters to the right of column 72 (or column 132 if -extend_source is specified) in a non-comment line.

This option has no effect on truncation; lines that exceed the maximum column width are always truncated.

This option does not apply to free-format source files.

The default is -warn notruncated_source.

-warn nouninitialized
Disables warning messages for a variable used before a value could be assigned to it.

The default is -warn uninitialized.

-warn nogeneral, -nowarn, -w
Disables all warning messages.

The default is `-warn general`, which enables all warning messages.

The options `-warn nogeneral`, `-nowarn`, and `-w` are synonymous.

-warn nouncalled
Disables warning messages about a statement function that is never called.

The default is `-warn uncalled`.

-warn unused
Requests warning messages for variables that are declared but not used.

The default is `-warn nounused` or `-w1`, which are synonymous.

-warn nousage
Disables warning messages about questionable programming practices which, although allowed, are often the result of programming errors. Examples are a continued character or Hollerith literal whose first part ends before the statement field and appears to end with trailing spaces.

The default is `-warn usage`.

3.101 -warning_severity *keyword* — Elevate Severity of Warning Messages

Use this option to elevate warning-level messages to error-level status.

The default is `-warning_severity warning`, which leaves all compiler warning messages at warning-level status.

The following options apply:

-warning_severity error
Turns all compiler warning-level messages into error-level messages.

-warning_severity stderror
Turns all standards-checking compiler warning-level messages into error-level messages.

Standards checking is enabled when the `-std` option is specified.

3.102 -what

`-what` is the same as `-version`. See Section 3.97.

3.103 -wsf

`-wsf` is the nonpreferred spelling of `-hpf`. See Section 3.50.

4

Using the Ladebug Debugger

This chapter contains the following topics:

- Section 4.1, Overview of Ladebug and dbx Debuggers
- Section 4.2, Compaq Fortran Options for Debugging
- Section 4.3, Running the Debugger
- Section 4.4, Sample Program and Debugging Session
- Section 4.5, Summary of Debugger Commands
- Section 4.6, Displaying Variables
- Section 4.7, Expressions in Debugger Commands
- Section 4.8, Debugging Mixed-Language Programs with Ladebug
- Section 4.9, Debugging a Program that Generates an Exception
- Section 4.10, Locating Unaligned Data
- Section 4.11, Using Alternate Entry Points
- Section 4.12, Debugging Optimized Programs

4.1 Overview of Ladebug and dbx Debuggers

You can use these debuggers to debug Compaq Fortran programs:

- On Tru64 UNIX: Ladebug and dbx
- On Linux: Ladebug

Ladebug supports Compaq Fortran data types, syntax, and use. Ladebug is a source-level, symbolic debugger that lets you:

- Control the execution of individual source lines in a program.
- Set stops (breakpoints) at specific source lines or under various conditions.
- Change the value of variables in your program.

- Refer to program locations by their symbolic names, using the debugger's knowledge of the Compaq Fortran language to determine the proper scoping rules and how the values should be evaluated and displayed.
- Print the values of variables and set a **tracepoint** (trace) to notify you when the value of a variable changes. Another term for a tracepoint is a **watchpoint**.
- Perform other functions, such as examining `core` files, examining the call stack, or displaying registers.

The command names and command syntax used by Ladebug and dbx are almost identical. However, Ladebug provides significantly more Compaq Fortran language support than dbx, especially Fortran 95/90 features not found in the FORTRAN-77 standard.

This chapter primarily describes the Ladebug debugger.

4.2 Compaq Fortran Options for Debugging

The `-g`*n* options control the amount of information placed in the object file for debugging. To use Ladebug, you should specify the `-ladebug` option along with the `-g`, `-g2`, or `-g3` options.

Table 4–1 summarizes the information provided by the `-g`*n* options and their relationship to the `-O`*n* options.

Table 4–1 Command-Line Options Affecting Traceback and Symbol Table Information

Option	Traceback Information	Debugging Symbol Table Information	Effect on -O*n* Options
`-g0`	No	No	Default is `-O4`.
`-g1` (default)	Yes	No	Default is `-O4`.
`-g` or `-g2`[1]	Yes	Yes, but for unoptimized code.	Changes default to `-O0`.
`-g3`	Yes	Yes, but for unoptimized code.	Default is `-O4`.
`-ladebug`	Yes	Allows access to Fortran 95/90 dynamic arrays using standard Fortran 95/90 syntax, including array sections.	No effect.

[1]The `-g` and `-g2` options are equivalent.

Traceback information and symbol table information are both necessary for debugging. As shown in Table 4–1, if you specify `-g`, `-g2`, or `-g3`, the compiler provides the symbol table and traceback information needed for symbolic debugging. Unless you specify `-g0`, the compiler supplies traceback information in the object file.

To use the Ladebug debugger, you should specify the `f90` (on Tru64 UNIX systems) or the `fort` (on Linux systems) command and the command option `-ladebug` along with `-g`, `-g2`, or `-g3`.

Likely uses of these options at the various stages of program development are as follows:

- During early stages of program development, use the `-g` (`-g2`) option to create unoptimized code (optimization level `-O0`). This option also might be chosen later to debug reported problems from later stages.
- During the later stages of program development, use `-g0` or `-g1` to minimize the object file size and, as a result, the memory needed for program execution, usually with optimized code. (See Chapter 5, Performance: Making Programs Run Faster.)

Traceback and symbol table information result in a larger object file. When you have finished debugging your program, you can recompile and relink to create an optimized executable program or remove traceback and symbol table information with the `strip` command. (See `strip`(1).)

If your program generates an exception, see Section 4.9, Debugging a Program that Generates an Exception.

4.3 Running the Debugger

The Ladebug debugger provides the following user interfaces in the Compaq Tru64 UNIX operating system Programmer's Development Toolkit:

- A graphical Ladebug windowing interface *(TU*X only)*
- A character-cell interface

The examples in this chapter show the character-cell interface to the Ladebug debugger.

4.3.1 Creating the Executable Program and Running the Debugger

Use the `f90` command with certain options to create an executable program for debugging. To invoke the debugger, enter the debugger shell command and the name of the executable program.

4.3.1.1 Invoking Ladebug

The following commands create (compile and link) the executable program and invoke the character-cell interface to the Ladebug debugger:

```
% f90 -g -ladebug -o squares squares.f90
% ladebug squares
Welcome to the Ladebug Debugger Version x.x-xx
---------------
object file name: squares
reading symbolic information ... done
(ladebug)
```

In this example, the `f90` command:

- Compiles and links the program `squares.f90`.
- Requests symbol table information needed for symbolic debugging and no optimization (`-g` and `-ladebug`).
- Names the executable file `squares` instead of `a.out` (`-o squares`).

The `ladebug` shell command runs the debugger, specifying the executable program `squares`. The `ladebug` command accepts various options. (See `ladebug`(1).)

At the debugger prompt `(ladebug)`, you can enter a debugger command.

4.3.1.2 Invoking dbx

The following commands create the executable program and invoke the dbx debugger *(TU*X only)*. Note that the `-ladebug` option is omitted, which causes dbx to be used:

```
% f90 -g -o squares squares.f90
% dbx squares
dbx version x.x.x
Type 'help' for help.
squares:   1  PROGRAM SQUARES
(dbx)
```

In this example, the `f90` command:

- Compiles and links the program `squares.f90`.
- Requests symbol table information needed for symbolic debugging and no optimization (`-g`).
- Names the executable file `squares` instead of `a.out` (`-o squares`).

The `dbx` shell command runs the debugger, specifying the executable program `squares`. The `dbx` command accepts various options. (See `dbx`(1).)

At the debugger prompt `(dbx)`, you can enter a debugger command.

4.3.2 Debugger Commands and Breakpoints

To find out what happens at critical points in your program, you need to stop execution at these points and look at the contents of program variables to see if they contain the correct values. Points at which the debugger stops program execution are called **breakpoints**.

To set a breakpoint, use one of the forms of the `stop` or `stopi` commands.

Using the program created in Section 4.3.1, the following debugger commands set a breakpoint at line 4, run the program, continue the program, delete the breakpoint, rerun the program, and return to the shell:

```
(ladebug) stop at 4
[#1: stop at "squares.f90":4 ]
(ladebug) run
[1] stopped at [squares:4 0x120001880]
>   4      OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
(ladebug) cont
Process has exited with status 0
(ladebug) delete 1
(ladebug) rerun
Process has exited with status 0
(ladebug) quit
%
```

1. The `stop at 4` command sets a breakpoint at line 4.

   To set a breakpoint at the start of a subprogram (such as calc), use the `stop in` (such as `stop in calc`).

2. The `run` command begins program execution and stops at the first breakpoint. The program is now active, allowing you to view the values of variables with `print` commands and perform related functions.

3. The `cont` (continue) command resumes program execution.

   In addition to the `cont` command, you can also use the `step`, `next`, `run`, or `rerun` commands to resume execution.

4. The `delete 1` command shows how to delete a previously set breakpoint (with event number 1). For instance, you might need to delete a previously set breakpoint before you use the `rerun` command.

5. The `rerun` command runs the program again. Since there are no breakpoints, the program runs to completion.

6. The `quit` command exits the debugger and returns to the shell.

Other debugger commands include the following:

- To get help on debugger commands, enter the `help` command.

- To display previously typed debugger commands, type the `history` command.
- To look at, or examine, the contents of a location, use the `print` or `dump` commands.
- You can use the debugger `sh` command (followed by the desired shell command) to execute a shell command. For instance, if you cannot recall the name of a FUNCTION statement, the following `grep` shell command displays the lines containing the letters FUNCTION, allowing use of the function name (SUBSORT) in the `stop in` command:

```
(ladebug) sh grep FUNCTION data.for
   INTEGER*4 FUNCTION SUBSORT (A,B)
(ladebug) stop in subsort
(ladebug)
```

4.3.3 Ladebug Limitations

For a list of known Ladebug limitations, see the *Compaq Tru64 UNIX Ladebug Debugger Manual*.

4.4 Sample Program and Debugging Session

Example 4–1 shows a program called SQUARES that requires debugging. The program was compiled and linked without diagnostic messages from either the compiler or the linker. However, this program contains a logic error in an arithmetic expression.

Compiler-assigned line numbers have been added in the example so that you can identify the source lines to which the explanatory text refers.

Example 4–1 Sample Program SQUARES

```
1         PROGRAM SQUARES
2         INTEGER INARR(20), OUTARR(20)
3   C  ! Read the input array from the data file.
4         OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
5         READ(8,*,END=5) N, (INARR(I), I=1,N)
6       5  CLOSE (UNIT=8)
7   C ! Square all nonzero elements and store in OUTARR.
8         K = 0
9         DO I = 1, N
10          IF (INARR(I) .NE. 0) THEN
11             OUTARR(K) = INARR(I)**2
12          ENDIF
13        END DO
14
15  C ! Print the squared output values.  Then stop.
16        PRINT 20, K
17    20  FORMAT (' Number of nonzero elements is',I4)
18        DO I = 1, K
19          PRINT 30, I, OUTARR(I)
20    30    FORMAT(' Element',I4,' has value',I6)
21        END DO
22        END PROGRAM SQUARES
```

The program SQUARES performs the following functions:

1. Reads a sequence of integer numbers from a data file and saves these numbers in the array INARR (lines 4 and 5). The file `datafile.dat` contains one record with the integer values 4, 3, 2, 5, and 2. The first number (4) indicates the number of data items that follow.
2. Enters a loop in which it copies the square of each nonzero integer into another array OUTARR (lines 8 through 13).
3. Prints the number of nonzero elements in the original sequence and the square of each such element (lines 16 through 21).

Note: This example assumes that the program was executed without array bounds checking (set by the Section 3.23 command-line option). When executed with array bounds checking, a run-time error message appears.

When you run SQUARES, it produces the following output, regardless of the number of nonzero elements in the data file:

```
% squares
Number of nonzero elements is   0
```

The logic error occurs because variable K, which keeps track of the current index into OUTARR, is not incremented in the loop on lines 9 through 13. The statement K = K + 1 should be inserted just before line 11.

Example 4–2 shows how to start the debugging session and how to use the character-cell interface to the Ladebug debugger to find the error in the sample program in Example 4–1. Comments keyed to the callouts follow the example.

Example 4–2 Sample Debugging Session Using Program Squares

```
% f90 -g -ladebug -o squares squares.f90  ❶
% ladebug squares                         ❷
Welcome to the Ladebug Debugger Version x.x-xx
---------------
object file name: squares
reading symbolic information ... done
(ladebug) list 1,9                        ❸
      1   PROGRAM SQUARES
      2     INTEGER INARR(20), OUTARR(20)
      3 C  ! Read the input array from the data file.
>     4      OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
      5      READ(8,*,END=5) N, (INARR(I), I=1,N)
      6    5 CLOSE (UNIT=8)
      7 C  ! Square all nonzero elements and store in OUTARR.
      8      K = 0
      9      DO 10 I = 1, N
(ladebug) stop at 8                       ❹
[#1: stop at "squares.f90":8]
(ladebug) run                             ❺
[1] stopped at ["squares.f90":4 0x120001a88]
>   8      K = 0
(ladebug) step                            ❻
stopped at [squares:9 0x120001a90]
      9        DO 10 I = 1, N
(ladebug) print n, k                      ❼
4 0
(ladebug) step                            ❽
stopped at [squares:10 0x120001ab0]]
     10       IF(INARR(I) .NE. 0)  THEN
```

(continued on next page)

Example 4–2 (Cont.) Sample Debugging Session Using Program Squares

```
(ladebug) s
stopped at [squares:11 0x1200011acc]
     11       OUTARR(K) =   INARR(I)**2
(ladebug) print i, k                                    ❾
1 0
(ladebug) assign k = 1                                  ❿
(ladebug) watch variable k                              ⓫
[#2: watch variable (write) k 0x1400002c0 to 0x1400002c3 ]
(ladebug) cont                                          ⓬
Number of nonzero elements is   1
Element   1 has value     4
Process has exited with status 0
(ladebug) quit                                          ⓭
% vi squares.f90                                        ⓮
   .
   .
   .
10:       IF(INARR(I) .NE. 0) THEN
11:           K = K + 1
12:           OUTARR(K) = INARR(I)**2
13:       ENDIF
   .
   .
   .
% f90 -g -ladebug -o squares squares.f90               ⓯
% ladebug squares
Welcome to the Ladebug Debugger Version x.x-xx
Reading symbolic information ...done
(ladebug) when at 12 {print k}                          ⓰
[#1: when at "squares.f90":12 { print K} ]
```

(continued on next page)

Example 4–2 (Cont.) Sample Debugging Session Using Program Squares

```
(ladebug) run                                    ⓱
[1] when [squares:12 0x120001ae0]
1
[1] when [squares:12 0x120001ae0]
2
[1] when [squares:12 0x120001ae0]
3
[1] when [squares:12 0x120001ae0]
4
Number of nonzero elements is   4
Element   1 has value     9
Element   2 has value     4
Element   3 has value    25
Element   4 has value     4
Process has exited with status 0
(ladebug) quit                                   ⓲
%
```

❶ On the `f90` command line, the `-g` and `-ladebug` options direct the compiler to write the symbol information associated with SQUARES into the object file for the debugger. It also disables most optimizations done by the compiler to ensure that the executable code matches the source code of the program. On a Linux system, the `fort` command and the same options would give the compiler the same directions.

❷ The shell command `ladebug squares` runs the debugger, which displays its banner and the debugger prompt, `(ladebug)`. This command specifies the executable program as a file named `squares`. You can now enter debugger commands.

After the `ladebug squares` command, execution is initially paused before the start of the main program unit (before program SQUARES, in this example).

To use the dbx debugger, enter the following shell command instead of `ladebug squares`:

```
% dbx squares
dbx version x.x
Type 'help' for help.
squares:   4  OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
(dbx)
```

❸ The `list 1,9` command prints lines 1 through 9.

❹ The command `stop at 8` sets a breakpoint (1) at line 8.

❺ The `run` command begins program execution. The program stops at the first breakpoint, line 8, allowing you to examine variables N and K before program completion.

❻ The `step` advances the program to line 9.

The `step` command ignores source lines that do not result in executable code; also, by default, the debugger identifies the source line at which execution is paused.

To avoid stepping into a subprogram, use the `next` command instead of `step`

❼ The command `print n, k` displays the current values of variables N and K. Their values are correct at this point in the execution of the program.

❽ The two `step` commands continue executing the program into the loop (lines 9 to 11) that copies and squares all nonzero elements of INARR into OUTARR.

Certain commands can be abbreviated. In this example, the `s` command is an abbreviation of the `step` command.

❾ The command `print i, k` displays the current values of variables I and K.

Variable I has the expected value, 1. But variable K has the value 0 instead of the expected value, 1. To fix this error, K should be incremented in the loop just before it is used in line 11.

❿ The `assign` command assigns K the value 1.

⓫ The `watch variable k` command sets a watchpoint that is triggered every time the value of variable K changes. In the original version of the program, this watchpoint is never triggered, indicating that the value of variable K never changes (a programming error).

⓬ To test the patch, the `cont` command (an abbreviation of continue) resumes execution from the current location.

The program output shows that the patched program works properly, but only for the first array element. Because the watchpoint (`watch variable k` command) does not occur, the value of K did not change and there is a problem. The Ladebug message "Process has exited with status 0" shows that the program executed to completion.

When using the dbx debugger, the message "Program terminated normally" shows that the program executed to completion.

⓭ The `quit` command returns control to the shell so that you can correct the source file and recompile and relink the program.

⓮ The shell command `vi` runs a text editor and the source file is edited to add K = K + 1 after line 10, as shown. (Compiler-assigned line numbers have been added to clarify the example.)

⓯ The revised program is compiled and linked.

The shell command `ladebug squares` (or `dbx squares`) starts the debugger, using the revised program so that its correct execution can be verified.

⓰ The `when at 12 {print k}` command reports the value of K at each iteration through the loop.

⓱ The `run` command starts execution.

The displayed values of K confirm that the program is running correctly.

⓲ The `quit` command ends the debugging session, returning control to the shell.

4.5 Summary of Debugger Commands

Table 4–2 lists some of the more frequently used debugging commands available in Ladebug and dbx. Many of these commands can be abbreviated (for example, you can enter `c` instead of `cont`). When using the debugger with a windowing interface, you can access these commands by using the windowing interface (command buttons and command menu). For a complete list of available debugger commands, see the `ladebug`(1) and `dbx`(1) reference pages.

Table 4–2 Summary of Debugger Commands

Command Example	Description
`catch`	Displays all signals that the debugger is currently set to catch (see also `ignore`).
`catch fpe`	Tells the debugger to catch the fpe signal (or the signal specified). This prevents the specified signal from reaching the Compaq Fortran RTL. The signals that the Compaq Fortran RTL arranges to catch are listed in Section 8.3.
`catch unaligned`	Tells the debugger to catch the unaligned signal, as further described in Section 4.10. The signals that the Compaq Fortran RTL arranges to catch are listed in Section 8.3.

(continued on next page)

Table 4–2 (Cont.) Summary of Debugger Commands

Command Example	Description
`cont`	Resumes execution of the program that is being debugged. Note that there is no Ladebug command named `continue`.
`delete 2`	Removes the breakpoint or tracepoint identified by event number 2 (see also `status`).
`delete all`	Removes all breakpoints and tracepoints.
`help`	Displays debugger help text.
`history 5`	Displays the last 5 debugger commands.
`ignore`	Displays the signals the debugger is currently set to ignore. The ignored signals are allowed to pass directly to the Compaq Fortran RTL. (see also `catch`).
`ignore fpe`	Tells the debugger to ignore the fpe signal (or the signal specified). This allows the specified signal to pass directly to the Compaq Fortran RTL, allowing message display. The signals that the Compaq Fortran RTL arranges to catch are listed in Section 8.3.
`ignore unaligned`	Tells the debugger to ignore the unaligned signal (the default).
`kill`	Terminates the program process, leaving the debugger running and its breakpoints and tracepoints intact for when the program is rerun.
`list`	Displays source program lines. To list a range of lines, add the starting line number, a comma (,), and the ending line number, such as `list 1,9`.
`print k`	Displays the value of the specified variable, such as K.
`printregs`	Displays all registers and their contents.
`next`	Steps one source statement but does *not* step into calls to subprograms (compare with `step`).
`quit`	Ends the debugging session.
`run`	Runs the program being debugged. You can specify program arguments and redirection.
`rerun`	Runs the program being debugged again. You can specify program arguments and redirection.

(continued on next page)

Table 4–2 (Cont.) Summary of Debugger Commands

Command Example	Description
`return [`*`routine-name`*`]`	When using the `step` command, if you step into a subprogram that does not require further investigation, use the `return` command to continue execution of the current function until it returns to its caller. If you include the name of a routine with the `return` command, execution continues until control is returned to that routine. The *routine-name* is the name of the routine, usually named by a PROGRAM, SUBROUTINE, or FUNCTION statement. If there is no PROGRAM statement, the debugger refers to the main program with a prefix of `main$` followed by the file name.
`sh more progout.f90`	Executes the shell command `more` to display file `progout.f90`, then returns to the debugger environment.
`show thread`	Lists all threads known to the Ladebug debugger.
`status`	Displays breakpoints and tracepoints with their event numbers (see also `delete`).
`step`	Steps one source statement, including stepping *into* calls of a subprogram. For Compaq Fortran I/O statements, intrinsic procedures, 3f library routines, or other subprograms, use the `next` command instead of `step` to step over the subprogram call. Compare with `next`; also see `return`.
`stop in foo`	Stops execution (breakpoint) at the beginning of routine foo.
`stop at 100`	Stops execution at line 100 (breakpoint) of the current source file.
`stopi at` *xxxxxxx*	Stops execution at address *xxxxxxx* of the current executable program.
`thread [`*`n`*`]`	Identifies or sets the current thread context (ladebug).
`watch` *location*	Displays a message when the debugger accesses the specified memory location. For example, `watch 0x140000170`.
`watch variable` *m*	Displays a message when the debugger accesses the variable specified by *m*.
`whatis` *sym*	Displays data type of specified symbol.

(continued on next page)

Table 4–2 (Cont.) Summary of Debugger Commands

Command Example	Description
`when at` *9* {*command*}	When a specified line (such as 9) is reached, the *command* or commands are executed. For example, `when at 9 {print k}` prints the value of variable K when the program executes source code line 9.
`when in` *name* {*command*}	When a procedure specified by *name* is reached, the *command* or commands are executed. For example, `when in calc_ave {print k}` prints the value of variable K when the program begins executing the procedure named calc_ave.
`where`	Displays the call stack.
`where thread all`	Displays the stack traces of all threads.

The debuggers support other special-purpose commands. For example:

- You might use the `attach` and `detach` commands for programs with very long execution times.
- The `listobj` command may be helpful when debugging programs that depend on shared libraries. The `listobj` debugger command displays the names of executables and shared libraries currently known to the debugger.

For More Information:

- On debugger command syntax, use the `help` command.
- On the debugger windowing interface, use online windowing help or see Section 4.3.
- On examples of debugger commands, see Section 4.4 and Section 4.5.
- On Ladebug commands, see the *Compaq Tru64 UNIX Ladebug Debugger Manual* or `ladebug`(1) reference page.
- On dbx commands, see *Compaq Tru64 UNIX Programmer's Guide* and `dbx`(1).

4.6 Displaying Variables

To refer to a variable, use either the uppercase or lowercase letters. For example:

```
(ladebug) print J
(ladebug) print j
```

You can enter command names in uppercase:

```
(ladebug) PRINT J
```

If you compile the program with the `f90` command option `-names as_is` and you need to examine case-sensitive names, you can control whether Ladebug is case sensitive by setting the `$lang` environment variable to the name of a case-sensitive language (see Section 4.8).

4.6.1 Compaq Fortran Module Variables

To refer to a variable defined in a module, insert a dollar sign ($), the module name, and another dollar sign ($) before the variable name. For example, with a variable named J defined in a module named modfile (statement MODULE MODFILE), enter the following command to display its value:

```
(ladebug) list 5,9
     5     USE MODFILE
     6      INTEGER*4 J
     7      CHARACTER*1 CHR
     8      J = 2**8
(ladebug) PRINT $MODFILE$J
256
```

4.6.2 Compaq Fortran Common Block Variables

You can display the values of variables in a Fortran common block by using debugger commands such as `print` or `whatis`.

To display the entire common block, use the common block name.

To display a specific variable in a common block, use the variable name. For example:

```
(ladebug) list 1,11
```

```
   1       PROGRAM EXAMPLE
   2
   3       INTEGER*4 INT4
   4       CHARACTER*1 CHR
   5       COMMON /COM_STRA/ INT4, CHR
   6
   7       CHR = 'L'
   8
   9       END
(ladebug) PRINT COM_STRA
COMMON
        INT4 = 0
        CHR = "L"
(ladebug)
(ladebug) PRINT CHR
"L"
```

If the name of a data item in a common block has the same name as the common block itself, the data item is accessed.

4.6.3 Compaq Fortran Derived-Type Variables

Variables in a Fortran 95/90 derived-type (TYPE statement) are represented in Ladebug commands such as `print` or `whatis` using Fortran 95/90 syntax form.

For derived-type structures, use the derived-type variable name, a percent sign (`%`), and the member name. For example:

```
(ladebug) list 3,11
    3  TYPE X
    4      INTEGER A(5)
    5    END TYPE X
    6
    7   TYPE (X) Z
    8
    9   Z%A = 1
   10
   11   PRINT *,Z%A
(ladebug) PRINT Z%A
(1) 1
(2) 1
(3) 1
(4) 1
(5) 1
(ladebug)
```

To display the entire object, use the `PRINT` command with the object name. For example:

```
(ladebug) PRINT Z
```

4.6.4 Compaq Fortran Record Variables

To display the value of a field in a record structure, enter the variable name as: the record name, a delimiter (either a period (.) or a percent sign (%)), and the field name.

To view all fields in a record structure, enter the name of the record structure, such as REC (instead of REC.CHR or REC%CHR) in the previous example.

4.6.5 Compaq Fortran Pointer Variables

Compaq Fortran supports two types of pointers:

- Fortran 95/90 pointers (standard-conforming)
- Compaq Fortran CRAY-style pointers (extension to the Fortran 95/90 standards)

4.6.5.1 Fortran 95/90 Pointers

Fortran 95/90 pointers display their corresponding target data with a `print` command. You must specify the `-ladebug` option to provide Ladebug with information about pointers to arrays.

```
% f90 -g -ladebug point.f90
% ladebug ./a.out
Welcome to the Ladebug Debugger Version x.x-xx
------------------
object file name: ./a.out
Reading symbolic information ...done
(ladebug) stop in ptr
[#1: stop in ptr ]
(ladebug) list 1:13
      1       program ptr
      2
      3       integer, target :: x(3)
      4       integer, pointer :: xp(:)
      5
      6       x = (/ 1, 2, 3/)
      7       xp => x
      8
      9       print *, "x = ", x
     10       print *, "xp = ", xp
     11
     12       end
(ladebug) run
[1] stopped at [ptr:6 0x120001838]
      6        x = (/ 1, 2, 3/)
(ladebug) whatis x
int x(1:3)
(ladebug) whatis xp  ❶
int xp(:)
```

```
(ladebug) s
stopped at [ptr:7 0x120001880]
      7       xp => x
(ladebug) s
stopped at [ptr:9 0x120001954]
      9       print *, "x = ", x
(ladebug) s
x =            1           2           3
stopped at [ptr:10 0x1200019c8]
(ladebug) s
 xp =            1           2           3
stopped at [point:12 0x120001ad8]
     12       end
(ladebug) S
 xp =            1           2           3
(ladebug) whatis xp ❷
int xp(1:3)
(ladebug) print xp
(1) 1
(2) 2
(3) 3
(ladebug) quit
%
```

❶ For the first `whatis xp` command, xp has not yet been assigned to point to variable x and is a generic pointer.

❷ Since xp has been assigned to point to variable x, for the second `whatis xp` command, xp takes the same size, shape, and values as x.

4.6.5.2 CRAY-Style Pointers

Like Fortran 95/90 pointers, Compaq Fortran (CRAY-style) pointers (POINTER statement) display the target data in their corresponding source form with a `print` command.

```
(ladebug) stop at 14
[#1: stop at "dfpoint.f90":14 ]
(ladebug) run
[1] stopped at [dfpoint:14 0x1200017e4]
(ladebug) list 1,14
      1       program dfpoint
      2
      3       real i(5)
      4       pointer (p,i)
      5
      6       n = 5
      7
      8       p = malloc(sizeof(i(1))*n)
      9
     10       do j = 1,5
     11          i(j) = 10*j
     12       end do
     13
>    14       end
(ladebug) whatis p
float (1:5) pointer p
(ladebug) print p
0x140003060 = (1) 10
(2) 20
(3) 30
(4) 40
(5) 50
(ladebug) quit
%
```

4.6.6 Compaq Fortran Array Variables

For array variables, put subscripts within parentheses, as with Fortran 95/90 source statements. For example:

```
(ladebug) assign arrayc(1)=1
```

You can print out all elements of an array using its name. For example:

```
(ladebug) print arrayc
(1) 1
(2) 0
(3) 0
(ladebug)
```

Avoid displaying all elements of a large array. Instead, display specific array elements or array sections. For example, to print array element arrayc(2):

```
(ladebug) print arrayc(2)
(2) 0
```

4.6.6.1 Array Sections

An array section is a portion of an array that is an array itself. An array section can use subscript triplet notation consisting of a three parts: a starting element, an ending element, and a stride.

Consider the following array declarations:

```
INTEGER, DIMENSION(0:99)    :: arr
INTEGER, DIMENSION(0:4,0:4)  :: FiveByFive
```

Assume that each array has been initialized to have the value of the index in each position, for example, FiveByFive(4,4) = 44, arr(43) = 43. The following examples are array expressions that will be accepted by the debugger:

```
(ladebug) print arr(2)

2
(ladebug) print arr(0:9:2)

(0) = 0
(2) = 2
(4) = 4
(6) = 6
(8) = 8
(ladebug) print FiveByFive(:,3)

(0,3) = 3
(1,3) = 13
(2,3) = 23
(3,3) = 33
(4,3) = 43
(ladebug)
```

The only operations permissible on array sections are `whatis` and `print`.

4.6.6.2 Assignment to Arrays

Assignment to array elements are supported by Ladebug.

For information about assigning values to whole arrays and array sections, see the Fortran chapter in the *Compaq Tru64 UNIX Ladebug Debugger Manual*.

4.6.7 Complex Variables

Ladebug supports COMPLEX or COMPLEX*8, COMPLEX*16, and COMPLEX*32 variables and constants in expressions.

Consider the following Fortran program:

```
PROGRAM complextest
  COMPLEX*8 C8 /(2.0,8.0)/
  COMPLEX*16 C16 /(1.23,-4.56)/
  REAL*4     R4 /2.0/
  REAL*8     R8 /2.0/
  REAL*16    R16 /2.0/

  TYPE *, "C8=", C8
  TYPE *, "C16=", C16
END PROGRAM
```

Ladebug supports the display and assignment of COMPLEX variables and constants as well as basic arithmetic operators. For example:

```
Welcome to the Ladebug Debugger Version x.x-xx
------------------
object file name: complex
Reading symbolic information ...done
(ladebug) stop in complextest
[#1: stop in complextest ]
(ladebug) run
[1] stopped at [complextest:15 0x1200017b4]
     15       TYPE *, "C8=", C8
(ladebug) whatis c8
complex c8
(ladebug) whatis c16
double complex c16
(ladebug) print c8
(2, 8)
(ladebug) print c16
(1.23, -4.56)
(ladebug) assign c16=(-2.3E+10,4.5e-2)
(ladebug) print c16
(-23000000512, 0.04500000178813934)
(ladebug)
```

4.6.8 Compaq Fortran Data Types

Table 4–3 shows the Compaq Fortran data types and their equivalent built-in debugger names:

Table 4–3 Fortran Data Types and Debugger Equivalents

Fortran 95/90 Data Type Declaration	Debugger Equivalent
CHARACTER	character
INTEGER, INTEGER(KIND=*n*)	integer, integer*n
LOGICAL, LOGICAL (KIND=*n*)	logical, logical*n
REAL, REAL(KIND=4)	real
DOUBLE PRECISION, REAL(KIND=8)	real*8
REAL(KIND=16)	real*16
COMPLEX, COMPLEX(KIND=4)	complex
DOUBLE COMPLEX, COMPLEX(KIND=8)	double complex
COMPLEX(KIND=16), COMPLEX*32	long double complex

4.7 Expressions in Debugger Commands

Expressions in debugger commands use Fortran 95/90 source language syntax for operators and expressions.

Enclose debugger command expressions between curly braces ({ }). For example, the expression `print k` in the following statement is enclosed between curly braces ({ }):

```
(ladebug) when at 12 {print k}
```

4.7.1 Fortran Operators

The Compaq Fortran operators include the following:

- Relational operators, such as less than (.LT. or <) and equal to (.EQ. or ==)
- Logical operators, such as logical conjunction (.AND.) and logical disjunction (.OR.)
- Arithmetic operators, including addition (+), subtraction (–), multiplication (*), and division (/).

For More Information:

- For a complete list of operators, see the *Compaq Fortran Language Reference Manual*.

4.7.2 Procedures

The Ladebug debugger supports invocation of user-defined specific procedures using Fortran 95/90 source language syntax. Ladebug also supports the invocation of some of the Fortran 95/90 generic and specific procedures, currently limited to scalar arguments.

For More Information:

- For a complete list of intrinsic procedures, see the *Compaq Fortran Language Reference Manual*.
- For a list of supported intrinsic procedures and known limitations of that support, see the *Compaq Tru64 UNIX Ladebug Debugger Manual*.

4.8 Debugging Mixed-Language Programs with Ladebug

The Ladebug debugger lets you debug mixed-language programs. Program flow of control across subprograms written in different languages is transparent.

The debugger automatically identifies the language of the current subprogram or code segment on the basis of information embedded in the executable file. For example, if program execution is suspended in a subprogram in Fortran, the current language is Fortran. If the debugger stops the program in a C function, the current language becomes C. The debugger uses the current language to determine the valid expression syntax and the semantics used to evaluate an expression.

The debugger sets the `$lang` variable to the language of the current subprogram or code segment. By manually setting the `$lang` debugger variable, you can force the debugger to interpret expressions used in commands by the rules and semantics of a particular language. For example, you can check the current setting of `$lang` and change it as follows:

```
(ladebug) print $lang
"C++"
(ladebug) set $lang = "Fortran"
```

When the `$lang` environment variable is set to “Fortran”, names are case insensitive. To make names case-sensitive when the program was compiled with the `-names as_is` option, specify another language for the `$lang` environment variable, such as C, view the variable, then set the `$lang` environment variable to “Fortran”.

4.9 Debugging a Program that Generates an Exception

If your program encounters an exception at run time, to make it easier to debug the program, you should recompile and relink with the following `f90` options before debugging the cause of the exception:

- Use the `-fpen` option to control the handling of exceptions. (See Section 3.44, -fpen — Control Arithmetic Exception Handling and Reporting.)
- If you use `-fpe0` (default), specify the `-synchronous_exceptions` option to ensure that the exception error is reported as close to the cause of the exception as possible. (See Section 3.86, -synchronous_exceptions — Report Exceptions More Precisely .)
- Like other debugging tasks, use the `-g` and `-ladebug` options to generate sufficient symbol table information and debug unoptimized code. (See Section 4.2, Compaq Fortran Options for Debugging.)

If requested, Ladebug will catch and handle exceptions before the Compaq Fortran run-time library does. You can use the Ladebug commands `catch` and `ignore` to control whether Ladebug catches exceptions or ignores them:

- When Ladebug *catches* an exception, a Ladebug message is displayed and execution stops at that statement line. The error-handling routines provided by the Compaq Fortran run-time library are not called. At this point, you can examine variables and determine where in the program the exception has occurred.
- When Ladebug *ignores* an exception, the exception is passed to the Compaq Fortran run-time library. This allows the handling and display of Compaq Fortran run-time exception messages in the manner requested during compilation.

To obtain the appropriate Compaq Fortran run-time error message when debugging a program that generates an exception (especially one that allows program continuation), you might need to use the `ignore` command before running the program. For instance, use the following command to tell the debugger to ignore floating-point exceptions and pass them through to the Compaq Fortran run-time library:

```
(ladebug) ignore fpe
```

In cases where you need to locate the part of the program causing an exception, consider using the `where` command.

For More Information:

- On run-time errors, see Chapter 8, Run-Time Errors and Signals.

4.10 Locating Unaligned Data

As discussed in Chapter 5, unaligned data can slow program execution. You should determine the cause of the unaligned data, fix the source code (if necessary), and recompile and relink the program.

If your program encounters unaligned data at run-time, to make it easier to debug the program, you should recompile and relink with the following command-line options *before* debugging the cause of the exception:

- Use the `-fpe`*n* option to control the handling of exceptions (see Section 3.44).
- If you use `-fpe0` (the default), specify the `-synchronous_exceptions` option to ensure that the exception error is reported as close to the cause of the exception as possible (see Section 3.86).

4.10.1 Locating Unaligned Data With Ladebug

To determine the cause of the unaligned data when using Ladebug, follow these steps:

1. Run the Ladebug debugger, specifying the program with the unaligned data (shown as `testprog` in the following example):

   ```
   % ladebug testprog
   ```

2. Before you run the program, enter the `catch unaligned` command:

   ```
   (ladebug) catch unaligned
   ```

3. Run the program:

   ```
   (ladebug) run
   Unaligned access pid=28413 <a.out> va=140000154 pc=3ff80805d60
       ra=1200017e8 type=stl
   Thread received signal BUS
   stopped at [oops:13 0x120001834]
        13       end
   ```

4. Enter a `list` command to display the source code at line 12:

   ```
   (ladebug) list 12
     12        i4 = 1
   > 13          end
   ```

5. Enter the `where` command to find the location of the unaligned access:

   ```
   (ladebug) where
   ```

6. Use any other appropriate debugger commands needed to isolate the cause of the unaligned data, such as `up`, `list`, and `down`.

7. Repeat these steps for other areas where unaligned data is reported. Use the `rerun` command to run the program again instead of exiting the debugger and running it from the shell prompt.

   After fixing the causes of the unaligned data (see Section 5.4), compile and link the program again.

4.10.2 Locating Unaligned Data With dbx

To determine the cause of the unaligned data when using `dbx` *(TU*X only)*, follow these steps:

1. Write down the addresses reported in the run-time message (see Section 5.4.2). For this example, assume the pc address in the "Unaligned access" message is 0x1200017f0.

2. Run the debugger, specifying the program with the unaligned data (shown as `testprog` in the following example):

   ```
   % dbx testprog
   ```

3. Set a breakpoint at the address reported in the warning message by using the `stopi at` command (shown as 0x1200017f0 below):

   ```
   (dbx) stopi at 0x1200017f0
    stopi at 0x1200017f0]
   ```

4. Run the program. It will stop at the breakpoint:

   ```
   (dbx) run
   [2] stopi at 0x1200017f0]
   [2] stopped at >*[oops:12, 0x1200017f0]         stl     r3, 0(r1)
   ```

5. Enter the `where` command to find the location of the unaligned access:

   ```
   (dbx) where
   ```

6. Use any other appropriate debugger commands needed to isolate the cause of the unaligned data, such as `list`, `up`, and `down`.

7. Repeat these steps for other addresses that warning messages report.

8. After fixing the causes of the unaligned data (see Section 5.4), compile and link the program again.

4.11 Using Alternate Entry Points

If a subprogram uses alternate entry points (ENTRY statement within the subprogram), Ladebug handles alternate entry points as separate subprograms, including:

- Use of the ENTRY statement name as a breakpoint (`stop in` command)
- Use of the `where` command at an alternate entry point breakpoint location

4.12 Debugging Optimized Programs

The Compaq Fortran compiler performs code optimizations (`-O4`) by default, unless you specify `-g` (or `-g2`).

Debugging optimized code is recommended only under special circumstances, for example, if a problem disappears when you specify the `-O0` option.

One aid to debugging optimized code is to use one of the following command-line options:

- Use the `-show code` and `-V` options to generate a listing file that shows the compiled code produced for your program.
- Use the `-S` option to generate an Assembler file (`.s`).

By referring to a listing of the generated code, you can see how the compiler optimizations affected your code. This lets you determine the debugging commands you need in order to isolate the problem.

When you try to perform a debugger operation on a variable or language construct that has been optimized, the variable or line may not exist in the debugging environment. For example:

- If the Compaq Fortran compiler can determine that a memory location for a variable is not needed for the correct operation of a program, no memory is allocated to it.
- If the Compaq Fortran compiler can determine that an entire Fortran 95/90 statement is not needed for correct operation of the program (for example, an unnecessary CONTINUE statement), that statement is not represented in the object code. As a result, the debugger will use the next available line.

For More Information:

- See Section 5.8, Optimization Levels: the -On Option.
- See Section 5.9, Other Options Related to Optimization.

5

Performance: Making Programs Run Faster

This chapter contains the following topics:

- Section 5.1, Efficient Compilation and the Software Environment
- Section 5.2, Using the time Command to Measure Performance
- Section 5.3, Using Profiling Tools
- Section 5.4, Data Alignment Considerations
- Section 5.5, Using Arrays Efficiently
- Section 5.6, Improving Overall I/O Performance
- Section 5.7, Additional Source Code Guidelines for Run-Time Efficiency
- Section 5.8, Optimization Levels: the -On Option
- Section 5.9, Other Options Related to Optimization

Note

To invoke the Compaq Fortran compiler, use:

- `f90` on Tru64 UNIX Alpha systems
- `fort` command on Linux Alpha systems

This chapter uses `f90` to indicate invoking Compaq Fortran on both systems, so replace this command with `fort` if you are working on a Linux Alpha system.

To invoke the Compaq C compiler, use:

- `cc` on Tru64 UNIX Alpha systems

- ccc on Linux Alpha systems

This chapter uses cc to indicate invoking Compaq C on both systems, so replace this command with ccc if you are working on a Linux Alpha system.

5.1 Efficient Compilation and the Software Environment

Before you attempt to analyze and improve program performance, you should:

- Obtain and install the latest version of Compaq Fortran, along with performance products that can improve application performance, such as the Compaq Extended Mathematical Library (CXML).
- Use the f90 command (or, on Linux systems, the fort command) and its options in a manner that lets the Compaq Fortran compiler perform as many optimizations as possible to improve run-time performance.
- Use certain performance capabilities provided by the Compaq Tru64 UNIX operating system.
- Make sure that you correct any errors you might have encountered during the early stages of program development.

5.1.1 Install the Latest Version of Compaq Fortran and Performance Products

To ensure that your software development environment can significantly improve the run-time performance of your applications, obtain and install the following optional software products:

- The latest version of Compaq Fortran

  New releases of the Compaq Fortran compiler and its associated run-time libraries may provide new features that improve run-time performance.

  The Compaq Fortran run-time libraries shipped with Compaq Fortran are also shipped with the Compaq Tru64 UNIX operating system. Always install the Compaq Fortran subset with the highest subset number. This number is always available, for both Tru64 and Linux operating systems, at the following Web page:

  http://www.compaq.com/fortran

  If your application will be run on a Compaq Tru64 UNIX system other than your program development system, be sure to install the same (or later) version of the Compaq Fortran run-time environment on those systems.

You can obtain the appropriate Compaq Services software product maintenance contract to automatically receive new versions of Compaq Fortran. For information on more recent Compaq Fortran releases, contact the Compaq Customer Support Center (CSC) if you have the appropriate support contract, or contact your local Compaq sales representative.

When using a shared memory, multiprocessor system, you can choose either directed parallel processing or Compaq KAP Fortran/OpenMP for Tru64 UNIX.

- Compaq KAP Fortran/OpenMP for Tru64 UNIX *(TU*X only)*

  Allows preprocessing of Compaq Fortran source files to improve their run-time performance. You can purchase this product from Compaq. See the KAP Optimizers Web site at:

  `http://www.compaq.com/hpc/software/kap.html`

  The KAP performance preprocessor also supports parallel processing using automatic and directed decomposition for a shared memory multiprocessor Alpha system.

  You can do one of the following:

  - Use the preprocessor-only `kapf90` command to produce improved Fortran 95/90 source files before compiling them with the `f90` command.
  - Use the `kf90` command to invoke the preprocessor, compiler, and linker to create an executable program.

  For example, the following `kf90` command:

  - Specifies the KAP preprocessor be run for the free-form file `for_cal.f90`
  - Recognizes the BLAS level 2 and 3 routines
  - Searches the CXML library for unresolved references
  - Compiles and links the resulting preprocessed source file:

    ```
    % kf90 -fkapargs='-lc=blas' for_cal.f90 -lcxml
    ```

  For More Information:

  - See *Compaq KAP Fortran/OpenMP for Tru64 UNIX User Guide*.

- Compaq Extended Mathematical Library (CXML) for Compaq Tru64 UNIX Systems

  See Chapter 13, Using the Compaq Extended Math Library (CXML).

- Performance profiling and feedback tools provided with Compaq Tru64 UNIX

  The standard set of U*X profiling and performance tools include `prof`, `gprof`, `pixie` *(TU*X only)*, `cord`, and the use of feedback files.

  Compaq Tru64 UNIX Version 4.0 or later also includes:

  – The Atom tool, which consists of a set of routines for creating custom-designed program-analysis tools.

  – Prepackaged Atom-based program-analysis tools, which include the profiling tools `pixie` *(TU*X only)* and `hiprof`.

  For More Information:

  - See `atom`(1).
  - See the *Compaq Tru64 UNIX Programmer's Guide*.

- System-wide performance products

  Other products are not specific to a particular programming language or application, but can improve system-wide performance, such as minimizing disk device I/O and handling capacity planning. Such Tru64 UNIX products include DECRaid (shadowing and striping) and such POLYCENTER products as the Capacity Planner, Performance Solution, and Performance Advisor.

  Adequate process limits and virtual memory space as well as proper system tuning are especially important when running large programs, such as those accessing large arrays.

For More Information:

- About system-wide tuning and suggestions for other performance enhancements on Compaq Tru64 UNIX systems, see *Compaq Tru64 UNIX System Tuning and Performance*.

5.1.2 Compile Using Multiple Source Files and Appropriate f90 Options

During the earlier stages of program development, you can use incremental compilation with minimal optimization. For example:

```
% f90 -c -O1 sub2.f90
% f90 -c -O1 sub3.f90
% f90 -o main.out -g -O0 main.f90  sub2.o  sub3.o
```

During the later stages of program development, you should specify multiple source files together and use an optimization level of at least `-O4` on the `f90` command line to allow more interprocedure optimizations to occur. For instance, the following command compiles all three source files together using the default level of optimization (`-O4`):

```
% f90 -o main.out main.f90  sub2.f90  sub3.f90
```

Compiling multiple source files lets the compiler examine more code for possible optimizations, which results in:

- Inlining more procedures
- More complete data flow analysis
- Reducing the number of external references to be resolved during linking

For very large programs, compiling all source files together may not be practical. In such instances, consider compiling source files containing related routines together using multiple `f90` commands, rather than compiling source files individually.

Table 5–1 shows `f90` options that can improve performance. Most of these options do not affect the accuracy of the results, while others improve run-time performance but can change some numeric results.

Compaq Fortran performs certain optimizations unless you specify the appropriate `f90` command options. Additional optimizations can be enabled or disabled using `f90` command options.

Table 5–1 lists the `f90` options that can directly improve run-time performance.

Table 5–1 Options That Affect Run-Time Performance

Option Names	Description	For More Information
`-align` *keyword*	Controls whether padding bytes are added between data items within common blocks, derived-type data, and Compaq Fortran record structures to make the data items naturally aligned.	Section 5.4
`-architecture` *keyword*	Determines the type of Alpha architecture code instructions to be generated for the program unit being compiled. All Alpha processors implement a core set of instructions; certain processor versions include additional instruction extensions.	Section 3.5
`-cord` and `-feedback` *file*	Uses a feedback file created during a previous compilation by specifying the `-gen_feedback` option. These options use the feedback file to improve run-time performance, optionally using `cord` to rearrange procedures.	Section 5.3.5

`-fast`	Sets the following performance-related options:
	`-align dcommons`
	`-align sequence`
	`-arch host`
	`-assume bigarrays` *(TU*X only)*
	`-assume nozsize` *(TU*X only)*
	`-assume noaccuracy_sensitive` (same as `-fp_reorder`)
	`-math_library fast`
	`-tune host`
		See description of each option

`-fp_reorder`	Allows the compiler to reorder code based on algebraic identities to improve performance, enabling certain optimizations. The numeric results can be slightly different from the default (`-no_fp_reorder`) because of the way intermediate results are rounded. This slight difference in numeric results is acceptable to most programs.	Section 5.9.7

(continued on next page)

Table 5–1 (Cont.) Options That Affect Run-Time Performance

Option Names	Description	For More Information
`-gen_feedback`	Requests generated code that allows accurate feedback information for subsequent use of the `-feedback` *file* option (optionally with `cord`). Using `-gen_feedback` changes the default optimization level from `-O4` to `-O0`.	Section 5.3.5
`-hpf` *num* and related options *(TU*X only)*	Specifies that the code generated for this program will allow parallel execution on multiple processors	Section 3.50
`-inline all`	Inlines every call that can possibly be inlined while generating correct code. Certain recursive routines are not inlined to prevent infinite loops.	Section 5.9.3
`-inline speed`	Inlines procedures that will improve run-time performance with a likely significant increase in program size.	Section 5.9.3
`-inline size`	Inlines procedures that will improve run-time performance without a significant increase in program size. This type of inlining occurs at optimization level `-O4` and `-O5`.	Section 5.9.3
`-math_library fast`	Requests the use of certain math library routines (used by intrinsic functions) that provide faster speed. Using this option causes a slight loss of accuracy and provides less reliable arithmetic exception checking to get significant performance improvements in those functions.	Section 3.61
`-mp` *(TU*X only)*	Enables parallel processing using directed decomposition (directives inserted in source code). This can improve the performance of certain programs running on shared memory multiprocessor systems	Section 3.64
`-O`*n* (`-O0` to `-O5`)	Controls the optimization level and thus the types of optimization performed. The default optimization level is `-O4`, unless you specify `-g2`, `-g`, or `-gen_feedback`, which changes the default to `-O0` (no optimizations). Use `-O5` to activate loop transformation optimizations.	Section 5.8

(continued on next page)

Table 5–1 (Cont.) Options That Affect Run-Time Performance

Option Names	Description	For More Information
`-om` *(TU*X only)*	Used with the `-non_shared` option to request certain code optimizations after linking, including nop (No Operation) removal, .lita removal, and reallocation of common symbols. This option also positions the global pointer register so the maximum addresses fall in the global-pointer window.	Section 3.73
`-omp` *(TU*X only)*	Enables parallel processing using directed decomposition (directives inserted in source code). This can improve the performance of certain programs running on shared memory multiprocessor systems	Section 3.74
`-p`, `-p1`	Requests profiling information, which you can use to identify those parts of your program where improving source code efficiency would most likely improve run-time performance. After you modify the appropriate source code, recompile the program and test the run-time performance.	Section 5.3
`-pg`	Requests profiling information for the `gprof` tool, which you can use to identify those parts of your program where improving source code efficiency would most likely improve run-time performance. After you modify the appropriate source code, recompile the program and test the run-time performance.	Section 5.3
`-pipeline`	Activates the software pipelining optimization (a subset of `-O4`).	Section 3.76
`-speculate` *keyword* *(TU*X only)*	Enables the speculative execution optimization, a form of instruction scheduling for conditional expressions.	Section 3.84
`-transform_loops`	Activates a group of loop transformation optimizations (a subset of `-O5`).	Section 3.89

(continued on next page)

Table 5–1 (Cont.) Options That Affect Run-Time Performance

Option Names	Description	For More Information
`-tune` *keyword*	Specifies the target processor generation (chip) architecture on which the program will be run, allowing the optimizer to make decisions about instruction tuning optimizations needed to create the most efficient code. Keywords allow specifying one particular Alpha processor generation type, multiple processor generation types, or the processor generation type currently in use during compilation. Regardless of the setting of `-tune` *keyword*, the generated code will run correctly on all implementations of the Alpha architecture.	Section 5.9.4
`-unroll` *num*	Specifies the number of times a loop is unrolled (*num*) when specified with optimization level `-O3` or higher. If you omit `-unroll` *num*, the optimizer determines how many times loops are unrolled.	Section 5.8.4.1

Table 5–2 lists options that can slow program performance. Some applications that require floating-point exception handling or rounding might need to use the `-fpe`*n* and `-fprm dynamic` options. Other applications might need to use the `-assume dummy_aliases` or `-vms` options for compatibility reasons. Other options listed in Table 5–2 are primarily for troubleshooting or debugging purposes.

Table 5–2 Options that Slow Run-Time Performance

Option Names	Description	For More Information
`-assume dummy_aliases`	Forces the compiler to assume that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use. These program semantics slow performance, so you should specify `-assume dummy_aliases` only for the called subprograms that depend on such aliases. The use of dummy aliases violates the FORTRAN-77 and Fortran 95/90 standards but occurs in some older programs.	Section 5.9.8
`-c`	If you use `-c` when compiling multiple source files, also specify `-o` *output* to compile many source files together into one object file. Separate compilations prevent certain interprocedure optimizations, such as when using multiple `f90` commands or using `-c` without the `-o` *output* option.	Section 2.1.6
`-check bounds`	Generates extra code for array bounds checking at run time.	Section 3.23
`-check omp_bindings` *(TU*X only)*	Provides run-time checking to enforce the binding rules for OpenMP Fortran API (parallel processing) compiler directives inserted in source code.	Section 3.26
`-check overflow`	Generates extra code to check integer calculations for arithmetic overflow at run time. Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.	Section 3.28
`-fpen` values greater than `-fpe0`	Using `-fpe1` *(TU*X only)*, `-fpe2` *(TU*X only)*, `-fpe3`, or `-fpe4` *(TU*X only)* (or using the `for_set_fpe` routine to set equivalent exception handling) slows program execution. For programs that specify `-fpe3` or `-fpe4` *(TU*X only)*, the impact on run-time performance can be significant.	Section 3.44
`-fprm dynamic` *(TU*X only)*	Certain rounding modes and changing the rounding mode can slow program execution slightly.	Section 3.46

(continued on next page)

Table 5–2 (Cont.) Options that Slow Run-Time Performance

Option Names	Description	For More Information
`-g`, `-g2`, `-g3`	Generates extra symbol table information in the object file. Specifying `-g` or `-g2` also reduces the default level of optimization to `-O0`.	Section 3.48
`-inline none` `-inline manual`	Prevents the inlining of all procedures (except statement functions).	Section 5.9.3
`-O0`, `-O1`, `-O2`, or `-O3`	Minimizes the optimization level (and types of optimizations). Use during the early stages of program development or when you will use the debugger.	Section 3.72 and Section 5.8
`-synchronous_exceptions`	Generates extra code to associate an arithmetic exception with the instruction that causes it, slowing efficient instruction execution. Use this option only when troubleshooting, such as when identifying the source of an exception.	Section 3.86
`-vms`	Controls certain VMS-related run-time defaults, including alignment. If you specify the `-vms` option, you may need to also specify the `-align records` option to obtain optimal run-time performance.	Section 3.98

For More Information:

- On compiling multiple files, see Section 2.1.6.
- On minimizing external references, see Section 11.1.1.

5.1.3 Process Shell Environment and Related Influences on Performance

Certain shell commands and system tuning can improve run-time performance:

- Specify adequate process limits and do system tuning.

  Especially when compiling or running large programs, check to make sure that process limits are adequate.

  With the C shell (`csh`), use the `limits` command to display the limits of your process and increase specified limits. For more information, see `csh`(1).

  With the Bourne, Korn, and bash *(L*X only)* shells, use the `ulimit` command to display the limits of your process and increase specified limits. For more information, see `sh`(1) (Bourne shell), `ksh`(1) (Korn shell), or `bash`(1) (bash shell) *(L*X only)*.

Your system manager can tune the system for efficient use. For example, to monitor system use during program execution or compilation, a system manager can use `vmstat`.

For more information on system tuning, see your operating system documentation.

- Redirect scrolled text.

  For programs that display a lot of text, consider redirecting text that is usually displayed on `stdout` to a file. Displaying a lot of text will slow down execution; scrolling text in a terminal window on a workstation can cause an I/O bottleneck (increased elapsed time) and use some CPU time.

  The following commands show how to run the program more efficiently by redirecting output to a file and then displaying the program output:

```
# myprog > results.lis
# more results.lis
```

- When compiling a program that contains a substantial amount of C language code, be aware that you can specify most `cc` options on the `f90` command line, including several that can improve performance. You can also compile C code using the `cc -c` option, and then use the `f90` command to compile and link the Compaq Fortran source files with the C language object files.

  Recall from Chapter 2 and Chapter 3 that the `f90` and `cc` commands invoke the Compaq Fortran compiler and Compaq C compiler, respectively, on Tru64 UNIX Alpha systems. The corresponding commands on Linux Alpha systems are `fort` and `ccc`.

For More Information:

- On system tuning and `cc` options related to performance, see your operating system documentation and the appropriate reference pages.

5.2 Using the time Command to Measure Performance

Use the `time` command to provide information about program performance.

Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.

Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same CPU system (model, amount of memory, version of the operating system, and so on) if possible.

If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.

For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

Using the form of the `time` command that specifies the name of the executable program provides the following:

- The elapsed, real, or "wall clock" time, which will be greater than the total charged actual CPU time.
- Charged actual CPU time, shown for both system and user execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

In the following example timings, the sample program being timed displays the following line:

```
Average of all the numbers is:    4368488960.000000
```

Using the Bourne shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
$ time a.out
Average of all the numbers is:    4368488960.000000
real    0m2.46s
user    0m0.61s
sys     0m0.58s
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

```
% time a.out
Average of all the numbers is:    4368488960.000000
0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Using the bash shell *(L*X only)*, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
[user@system user]$ time ./a.out
Average of all the numbers is:    4368488960.000000
elapsed  0m2.46s
user     0m0.61s
sys      0m0.58s
```

Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the `time` command line. (See Section 5.1.3.) Redirecting output from the program will change the times reported because of reduced screen I/O.

For more information, see `time`(1).

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example:

- Compaq Fortran intrinsic procedures, such as SYSTEM_CLOCK, DATE_AND_TIME, and TIME (see the *Compaq Fortran Language Reference Manual*)
- Library routines, such as `etime` or `time` (see Section 12.2, 3f Routines or `intro`(3f)).

5.3 Using Profiling Tools

To generate profiling information, use the `f90` compiler and the `prof`, `gprof`, and `pixie` *(TU*X only)* tools.

Profiling identifies areas of code where significant program execution time is spent. Along with the `f90` command, use the `prof` and `pixie` *(TU*X only)* tools to generate the following profile information:

- The CPU time spent in the different routines of the program, or **program counter sampling**. This type of profiling uses `prof`.
- The manner in which routines are called by other routines, or **call graph** information. This type of profiling uses `gprof`.
- The execution of basic blocks, called **basic block counting**. A **basic block** is a sequence of instructions entered only at the beginning and exited only at the end (no branches). This provides statistics on individual lines of code and is influenced by such optimizations as loop unrolling. This type of profiling uses `prof` and `pixie` *(TU*X only)*.

- The estimated number of CPU cycles spent for each source line in one or more procedures, or **source line CPU cycle use**. This type of profiling uses `prof` and `pixie` *(TU*X only)*.

Once you have determined those sections of code where most of the program execution time is spent, examine these sections for coding efficiency. Suggested guidelines for improving source code efficiency are provided in Section 5.7.

Along with profiling, you can consider generating a listing file with annotations of optimizations, by specifying the `-V` and `-annotations` options.

5.3.1 Program Counter Sampling (prof)

To obtain program counter sampling data, perform the following steps:

1. Use the `f90` command option `-p` to compile and link the program:

   ```
   % f90 -p -O3 -o profsample profsample.f90
   ```

   If you specify the `-c` option to prevent linking, you must specify the `-p` option when you link the program:

   ```
   % f90 -c -O3 profsample.f90
   % f90 -p -O3 -o profsample profsample.o
   ```

   Consider specifying optimization level `-O3` or `-inline manual` to minimize the inlining of procedures. Once inlined, procedures are not listed as separate routines but as part of the routine into which they have been inlined. Allowing full inlining would result in program counter sampling for a small number of (usually) large routines. This might not help you locate areas of the program where significant program execution time is spent.

2. Execute the profiled program:

   ```
   % profsample
   ```

   During program execution, profiling data is written to a profile data file, whose default name is `mon.out`. You can execute the program multiple times to generate multiple profile data files, which can be averaged. Use the `PROFDIR` environment variable to request a different profile data file name.

3. Run the `prof` command, which formats the profiling data and displays it in a readable format:

   ```
   % prof  profsample mon.out
   ```

You can limit the report created by `prof` by using `prof` command options, such as `-only`, `-exclude`, or `-quit`.

For example, if you only want reports on procedures calc_max and calc_min, you could use the following command line to read the profile data file named mon.out:

```
% prof -only calc_max -only calc_min profsample
```

The time spent in particular areas of code is reported by prof in the form of a percentage of the total CPU time spent by the program. To reduce the size of the report, you can either:

- Request that only certain procedures be included (by using the -only option).
- Exclude certain procedures (by using the -exclude option).

When you use the -only or -exclude options, the percentages are still based on all procedures of the application. To obtain percentages calculated by prof that are based on only those procedures included in the report, use the -Only and -Exclude options (use an uppercase initial letter in the option name).

You can use the -quit option to reduce the amount of information reported. For example, the following command prints information on only the five most time-consuming procedures:

```
% prof -quit 5 profsample
```

The following command limits information only to those procedures using 10% or more of the total execution time:

```
% prof  -quit 10% profsample
```

For More Information:

- On prof, see prof(1) and the *Compaq Tru64 UNIX Programmer's Guide*.

5.3.2 Call Graph Sampling (gprof)

To obtain call graph information, use the gprof tool. Perform the following steps:

1. Use the command-line option -pg when you compile and link the program:

   ```
   % f90 -pg -O3 -o profsample profsample.for
   ```

   If you specify the -c option to prevent linking, you must then specify the -pg option both when you compile and link the program:

   ```
   % f90 -pg -c -O3 profsample.f90
   % f90 -pg -O3 -o profsample profsample.f90
   ```

2. Execute the profiled program:

   ```
   % profsample
   ```

   During execution, profiling data is saved to the file `gmon.out`, unless the environment variable PROFDIR is set.

3. Run the formatting program `gprof`:

   ```
   % gprof profsample gmon.out
   ```

The output produced by `gprof` includes:

- Call graph profile
- Timing profile (similar to that produced by `prof`)
- Index

For More Information:

- On using `gprof` and its output, see the *Compaq Tru64 UNIX Programmer's Guide*.

5.3.3 Basic Block Counting (pixie and prof)

To obtain basic block counting information, perform the following steps:

1. Compile and link the program without the `-p` option:

   ```
   % f90 -O3 -o profsample profsample.f90
   ```

   Consider specifying optimization level `-O3` or `-inline manual` to minimize the inlining of procedures (once inlined, procedures are not listed as separate routines but as part of the routine into which they are inlined).

2. Run the profiling command `pixie`: *(TU*X only)*

   ```
   % atom -tools pixie profsample
   ```

   The `pixie` command creates: *(TU*X only)*

   - A program named `profsample.pixie` that is equivalent to `profsample` but contains additional code for counting the execution of each basic block.
   - A file named `profsample.Addrs`, which contains the address of each basic block.

3. Execute the profiled program `profsample.pixie` generated by `pixie`:

   ```
   % profsample.pixie
   ```

This program creates the file profsample.Counts, which contains the basic block counts.

4. Run prof with the -pixie option, to extract and display information from the profsample.Addrs and profsample.Counts files:

```
% prof -pixie profsample
```

When you specify the -pixie option *(TU*X only)*, the prof command searches for files with a suffix of .Addrs and .Counts (in this case profsample.Addrs and profsample.Counts).

You can reduce the amount of information in the report created by prof by using the -only, -exclude, -quit, and related options.

To create multiple profile data files, run the program multiple times.

For More Information:

- On prof, gprof, and pixie *(TU*X only)*, see prof(1), gprof(1), pixie(1), and the *Compaq Tru64 UNIX Programmer's Guide*.

5.3.4 Source Line CPU Cycle Use (prof and pixie)

You use the same files created by the pixie command (see Section 5.3.3) for basic block counting to estimate the number of CPU cycles used to execute each source file line.

To view a report of the number of CPU cycles estimated for each source file line, use the following options with the prof command:

- The -pixie *(TU*X only)* option is required to obtain source line information.
- The -heavy option prints an entry for each source code line, including the number of CPU cycles used by that line. Entries are sorted in descending order of CPU cycles and should be limited by using the prof command options that limit the report size, such as -quit, -only, or -exclude.
- The -lines option requests source line information, but in the order in which the lines occur in the program (not sorted in descending order of CPU cycles).

Depending on the level of optimization chosen, certain source lines might be optimized away.

The CPU cycle use estimates are based primarily on the instruction type and its operands and do not include memory effects such as cache misses or translation buffer fills.

For example, the following command sequence uses:

- The f90 and pixie *(TU*X only)* commands to create the necessary files.
- The prof command to request source line CPU cycle use information for the procedure named calc_max (-only option), sorted in descending order of CPU cycles (-heavy option):

```
% f90 -o profsample profsample.f90
% atom -tools pixie profsample
% profsample.pixie
% prof -pixie -heavy -only calc_max profsample
```

5.3.5 Creating and Using Feedback Files and Optionally cord

You can create a **feedback file** by using a series of commands. Once created, you can specify a feedback file in a subsequent compilation with the f90 command option -feedback. You can also request that cord use the feedback file to rearrange procedures, by specifying the -cord option on the f90 command line.

To create the feedback file, complete these steps:

1. Compile and link the program. Omit the -p option, but specify the -gen_feedback option:

   ```
   % f90 -o profsample -gen_feedback profsample.f90
   ```

   The -gen_feedback option changes the default optimization level to -O0.

   To include libraries in the profiling output, specify -non_shared.

2. Execute the profiling command pixie *(TU*X only)*:

   ```
   % pixie profsample
   ```

   The pixie command creates:

   - A program named profsample.pixie that is equivalent to profsample but contains additional code for counting the execution of each basic block.
   - A file named profsample.Addrs, which contains the address of each basic block.

3. Execute the profiled program profsample.pixie generated by pixie:

   ```
   % profsample.pixie
   ```

   This program creates the file profsample.Counts, which contains the basic block counts.

4. Run prof with the -pixie and -feedback options:

   ```
   % prof -pixie -feedback profsample.feedback profsample
   ```

   This prof command creates the feedback file profsample.feedback.

You can use the feedback file as input to the f90 compiler:

```
% f90 -feedback profsample.feedback -o profsample profsample.f90
```

The feedback file provides the compiler with actual execution information, which the compiler can use to improve such optimizations as inlining function calls.

Specify the desired optimization level (-O*n* option) for the f90 command with the -feedback *name* option (in this example the default is -O4).

You can use the feedback file as input to the f90 compiler and cord, as follows:

```
% f90 -cord -feedback profsample.feedback -o profsample profsample.f90
```

The -cord option invokes cord, which reorders the procedures in an executable program to improve program execution, using the information in the specified feedback file. Specify the desired optimization level (-O*n* option) for the f90 command with the -feedback *name* option (in this example -O4).

5.3.6 Atom Toolkit

*(TU*X only)* The Atom toolkit includes a programmable instrumentation tool and several prepackaged tools. The prepackaged tools include:

- hiprof

  Produces a flat profile of an application that shows the execution time spent in a given procedure, and a hierarchical profile that shows the execution time spent in a given procedure and all of its descendents.

- pixie

  Produces a profile of an application, by procedure, source line, or instruction. It partitions the application into basic blocks and counts the number of times each basic block is executed.

- third

  Performs memory access checks and detects memory leaks in an application.

To invoke atom tools, use the following general command syntax:

```
% atom -tool tool-name ...)
```

Atom does not work on programs built with the -om option.

For More Information:

- See the *Compaq Tru64 UNIX Programmers Guide*.
- See `atom`(1), `hiprof`(5), `pixie`(5), and `third`(5).

5.4 Data Alignment Considerations

For optimal performance on Alpha systems, make sure your data is aligned naturally.

A natural boundary is a memory address that is a multiple of the data item's size (data type sizes are described in Table 9–1). For example, a REAL (KIND=8) data item aligned on natural boundaries has an address that is a multiple of 8. An array is aligned on natural boundaries if all of its elements are.

All data items whose starting address is on a natural boundary are **naturally aligned**. Data not aligned on a natural boundary is called **unaligned data**.

Although the Compaq Fortran compiler naturally aligns individual data items when it can, certain Compaq Fortran statements (such as EQUIVALENCE) can cause data items to become unaligned (see Section 5.4.1).

Although you can use the `f90` command `-align` *keyword* options to ensure naturally aligned data, you should check and consider reordering data declarations of data items within common blocks and structures. Within each common block, derived type, or record structure, carefully specify the order and sizes of data declarations to ensure naturally aligned data. Start with the largest size numeric items first, followed by smaller size numeric items, and then nonnumeric (character) data.

5.4.1 Causes of Unaligned Data and Ensuring Natural Alignment

Common blocks (COMMON statement), derived-type data, and Compaq Fortran 77 record structures (RECORD statement) usually contain multiple items within the context of the larger structure.

The following declaration statements can force data to be unaligned:

- Common blocks (COMMON statement)

  The order of variables in the COMMON statement determines their storage order.

  Unless you are sure that the data items in the common block will be naturally aligned, specify either the `-align commons` or `-align dcommons` option, depending on the largest data size used.

  See Section 5.4.3.1, Arranging Data Items in Common Blocks.

- Derived-type (user-defined) data

  Derived-type data members are declared after a TYPE statement.

  If your data includes derived-type data structures, you should use the `-align records` option, unless you are sure that the data items in derived-type data structures will be naturally aligned.

  If you omit the SEQUENCE statement, the `-align records` option (default) ensures all data items are naturally aligned.

  If you specify the SEQUENCE statement, the `-align record` option is prevented from adding necessary padding to avoid unaligned data (data items are packed) unless you specify the `-align sequence` option. When you use SEQUENCE, you should specify data declaration order such that all data items are naturally aligned.

  See Section 5.4.3.2, Arranging Data Items in Derived-Type Data.

- Compaq Fortran record structures (RECORD and STRUCTURE statements)

  Compaq Fortran record structures usually contain multiple data items. The order of variables in the STRUCTURE statement determines their storage order. The RECORD statement names the record structure.

  If your data includes Compaq Fortran record structures, you should use the `-align records` option, unless you are sure that the data items in derived-type data and Compaq Fortran record structures will be naturally aligned.

  See Section 5.4.3.3, Arranging Data Items in Compaq Fortran Record Structures.

- EQUIVALENCE statements

  EQUIVALENCE statements can force unaligned data or cause data to span natural boundaries. For more information, see the *Compaq Fortran Language Reference Manual*.

To avoid unaligned data in a common block, derived-type data, or record structure (extension), use one or both of the following:

- For new programs or for programs where the source code declarations can be modified easily, plan the order of data declarations with care. For example, you should order variables in a COMMON statement such that numeric data is arranged from largest to smallest, followed by any character data (see the data declaration rules in Section 5.4.3).

- For existing programs where source code changes are not easily done or for array elements containing derived-type or record structures, you can use command line options to request that the compiler align numeric data by adding padding spaces where needed.

Other possible causes of unaligned data include unaligned actual arguments and arrays that contain a derived-type structure or Compaq Fortran record structure.

When actual arguments from outside the program unit are not naturally aligned, unaligned data access will occur. Compaq Fortran assumes all passed arguments are naturally aligned and has no information at compile time about data that will be introduced by actual arguments during program execution.

For arrays where each array element contains a derived-type structure or Compaq Fortran record structure, the size of the array elements may cause some elements (but not the first) to start on an unaligned boundary.

Even if the data items are naturally aligned within a derived-type structure without the SEQUENCE statement or a record structure, the size of an array element might require use of `f90 -align` options to supply needed padding to avoid some array elements being unaligned.

If you specify `-align norecords` or specify `-vms` without `-align records`, no padding bytes are added between array elements. If array elements each contain a derived-type structure with the SEQUENCE statement, array elements are packed without padding bytes regardless of the `f90` command options specified. In this case, some elements will be unaligned.

When `-align records` option is in effect, the number of padding bytes added by the compiler for each array element is dependent on the size of the largest data item within the structure. The compiler determines the size of the array elements as an exact multiple of the largest data item in the derived-type structure without the SEQUENCE statement or a record structure. The compiler then adds the appropriate number of padding bytes.

For instance, if a structure contains an 8-byte floating-point number followed by a 3-byte character variable, each element contains five bytes of padding (16 is an exact multiple of 8). However, if the structure contains one 4-byte floating-point number, one 4-byte integer, followed by a 3-byte character variable, each element would contain one byte of padding (12 is an exact multiple of 4).

For More Information:

- On the `-align` *keyword* options, see Section 5.4.4.

5.4.2 Checking for Inefficient Unaligned Data

During compilation, the Compaq Fortran compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described in Section 5.4.1.

Because unaligned data can slow run-time performance, it is worthwhile to:

- Double-check data declarations within common block, derived-type data, or record structures to ensure all data items are naturally aligned (see the data declaration rules in Section 5.4.3). Using modules to contain data declarations can ensure consistent alignment and use of such data.
- Avoid the EQUIVALENCE statement or use it in a manner that cannot cause unaligned data or data spanning natural boundaries.
- Ensure that passed arguments from outside the program unit are naturally aligned.
- Check that the size of array elements containing at least one derived-type data or record structure (extension) cause array elements to start on aligned boundaries (see Section 5.4.1).

There are two ways unaligned data might be reported:

- During compilation, warning messages are issued for any data items that are known to be unaligned (unless you specify the `-warn noalignments` option).
- During program execution, warning messages are issued for any data that is detected as unaligned. The message includes the address of the unaligned access. You can use the ladebug debugger to locate unaligned data.

  The following run-time message shows that:

  - The statement accessing the unaligned data (program counter) is located at 3ff80805d60
  - The unaligned data is located at address 140000154

```
Unaligned access pid=24821 <a.out> va=140000154, pc=3ff80805d60, ra=1200017bc
```

  To check where the address is located, use the debugger as described in Section 4.10.

  To suppress unaligned access run-time messages, use the `uac` command (see `uac`(1)).

5.4.3 Ordering Data Declarations to Avoid Unaligned Data

For new programs or when the source declarations of an existing program can be easily modified, plan the order of your data declarations carefully to ensure the data items in a common block, derived-type data, record structure, or data items made equivalent by an EQUIVALENCE statement will be naturally aligned.

Use the following rules to prevent unaligned data:

- Always define the largest size numeric data items first.
- If your data includes a mixture of character and numeric data, place the numeric data first.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.

When declaring data, consider using explicit length declarations, such as specifying a KIND parameter. For example, specify INTEGER(KIND=4) (or INTEGER(4)) rather than INTEGER. If you do use a default length (such as INTEGER, LOGICAL, COMPLEX, and REAL), be aware that the compiler options `-integer_size` and `-real_size` can change the size of an individual field's data declaration size and thus can alter the data alignment of a carefully planned order of data declarations.

Using the suggested data declaration guidelines minimizes the need to use the `-align` *keyword* options to add padding bytes to ensure naturally aligned data. In cases where the `-align` *keyword* options are still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

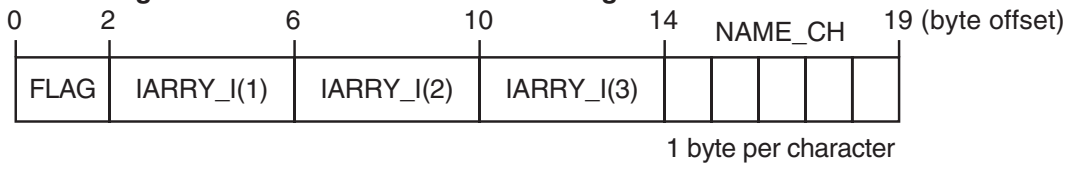
5.4.3.1 Arranging Data Items in Common Blocks

The order of data items in a COMMON statement determine the order in which the data items are stored. Consider the following declaration of a common block named X:

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ FLAG, IARRY_I(3), NAME_CH
```

As shown in Figure 5–1, if you omit the appropriate `f90` command options, the common block will contain unaligned data items beginning at the first array element of IARRY_I.

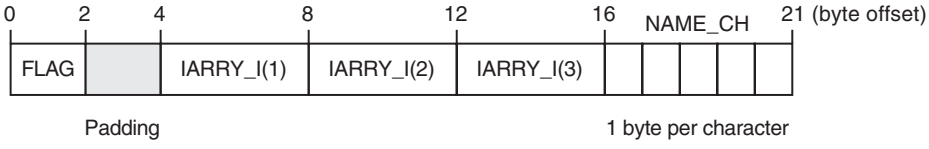
Figure 5–1 Common Block with Unaligned Data



ZK–6659A–GE

As shown in Figure 5–2, if you compile the program units that use the common block with the `-align commons` options, data items will be naturally aligned.

Figure 5–2 Common Block with Naturally Aligned Data



ZK–6660A–GE

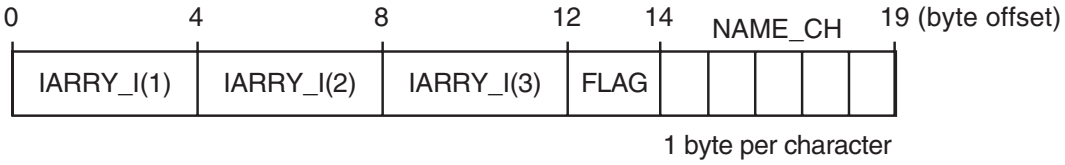
Because the common block X contains data items whose size is 32 bits or smaller, specify `-align commons`. If the common block contains data items whose size might be larger than 32 bits (such as REAL (KIND=8) data), use `-align dcommons`.

If you can easily modify the source files that use the common block data, define the numeric variables in the COMMON statement in descending order of size and place the character variable last. This provides more portability, ensures natural alignment without padding, and does not require the `f90` command options `-align commons` or `-align dcommons`:

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ IARRY_I(3), FLAG, NAME_CH
```

As shown in Figure 5–3, if you arrange the order of variables from largest to smallest size and place character data last, the data items will be naturally aligned.

Figure 5–3 Common Block with Naturally Aligned Reordered Data



ZK–7915A–GE

When modifying or creating all source files that use common block data, consider placing the common block data declarations in a module so the declarations are consistent. If the common block is not needed for compatibility (such as file storage or Compaq Fortran 77 use), you can place the data declarations in a module without using a common block.

5.4.3.2 Arranging Data Items in Derived-Type Data

Like common blocks, derived-type data may contain multiple data items (members).

Data item components within derived-type data will be naturally aligned on up to 64-bit boundaries, with certain exceptions related to the use of the SEQUENCE statement and `f90` options. See Section 5.4.4 for information about these exceptions.

Compaq Fortran stores a derived data type as a linear sequence of values, as follows:

- If you specify the SEQUENCE statement, the first data item is in the first storage location and the last data item is in the last storage location. The data items appear in the order in which they are declared. The `f90` options have no effect on unaligned data, so data declarations must be carefully specified to naturally align data.

  The `-align sequence` option specifically aligns data items in a SEQUENCE derived-type on natural boundaries.

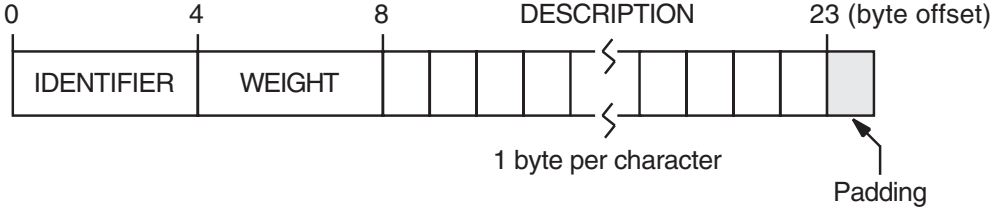
- If you omit the SEQUENCE statement, Compaq Fortran adds the padding bytes needed to naturally align data item components, unless you specify the `-align norecords` option.

Consider the following declaration of array CATALOG_SPRING of derived-type PART_DT:

```
MODULE DATA_DEFS
  TYPE PART_DT
    INTEGER           IDENTIFIER
    REAL              WEIGHT
    CHARACTER(LEN=15) DESCRIPTION
  END TYPE PART_DT
  TYPE (PART_DT) CATALOG_SPRING(30)
  .
  .
  .
END MODULE DATA_DEFS
```

As shown in Figure 5–4, the largest numeric data items are defined first and the character data type is defined last. There are no padding characters between data items and all items are naturally aligned. The trailing padding byte is needed because CATALOG_SPRING is an array; it is inserted by the compiler when the `-align records` option is in effect.

Figure 5–4 Derived-Type Naturally Aligned Data (in CATALOG_SPRING())



ZK–6658A–GE

5.4.3.3 Arranging Data Items in Compaq Fortran Record Structures

Compaq Fortran supports record structures provided by Compaq Fortran. Compaq Fortran record structures use the RECORD statement and optionally the STRUCTURE statement, which are extensions to the FORTRAN-77 and Fortran 95/90 standards. The order of data items in a STRUCTURE statement determine the order in which the data items are stored.

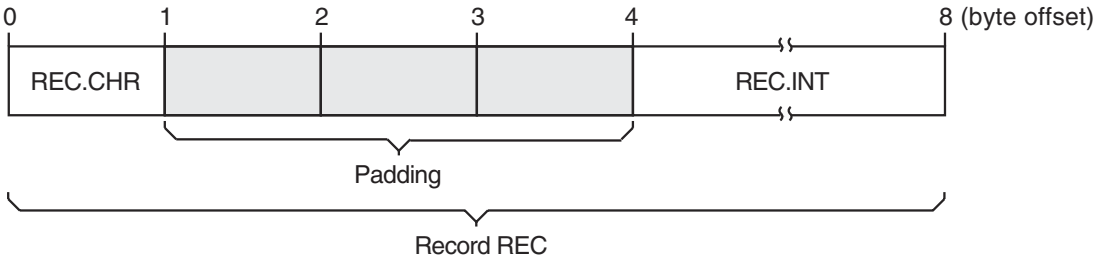
Compaq Fortran stores a record in memory as a linear sequence of values, with the record's first element in the first storage location and its last element in the last storage location. Unless you specify `-align norecords`, padding bytes are added if needed to ensure data fields are naturally aligned.

The following example contains a structure declaration, a RECORD statement, and diagrams of the resulting records as they are stored in memory:

```
STRUCTURE /STRA/
  CHARACTER*1 CHR
  INTEGER*4 INT
END STRUCTURE
   .
   .
   .
RECORD /STRA/ REC
```

Figure 5–5 shows the memory diagram of record REC for naturally aligned records.

Figure 5–5 Memory Diagram of REC for Naturally Aligned Records



ZK–2244A–GE

5.4.4 Options Controlling Alignment

The following options control whether the Compaq Fortran compiler adds padding (when needed) to naturally align multiple data items in common blocks, derived-type data, and Compaq Fortran record structures:

- The `-align commons` option requests that data in common blocks be aligned on up to 4-byte boundaries, by adding padding bytes as needed.

  Unless you specify `-fast`, the default is `-align nocommons` or arbitrary byte alignment of common block data. In this case, unaligned data can occur unless the order of data items specified in the COMMON statement places the largest numeric data item first, followed by the next largest numeric data (and so on), followed by any character data.

- The `-align dcommons` option requests that data in common blocks be aligned on up to 8-byte boundaries, by adding padding bytes as needed.

  Unless you specify `-fast`, the default is `-align nodcommons` or arbitrary byte alignment of data items in a common data.

Specify the -align dcommons option for applications that use common blocks, unless your application has no unaligned data or, if the application might have unaligned data, all data items are four bytes or smaller. For applications that use common blocks where all data items are four bytes or smaller, you can specify -align commons instead of -align dcommons.

- The -align norecords option requests that multiple data items in derived-type data and record structures (a Compaq Fortran extension) be aligned arbitrarily on byte boundaries instead of being naturally aligned. The default is -align records.
- The -align records option requests that multiple data items in record structures (extension) and derived-type data without the SEQUENCE statement be naturally aligned, by adding padding bytes as needed.
- The -align rec*N*byte option requests that fields of records and components of derived types be aligned on either the size byte boundary specified or the boundary that will naturally align them, whichever is smaller. This option does not affect whether common blocks are naturally aligned or packed.
- The -align sequence option controls alignment of derived types with the SEQUENCE attribute.

  The default -align nosequence option means that derived types with the SEQUENCE attribute are packed regardless of any other alignment rules. Note that -align none implies -align nosequence.

  The -align sequence option means that derived types with the SEQUENCE attribute obey whatever alignment rules are currently in use. Consequently, since -align records is a default value, then -align sequence alone on the command line will cause the fields in these derived types to be naturally aligned. Note that -fast and -align all imply -align sequence.

The default behavior is that multiple data items in derived-type data and record structures *will* be naturally aligned; data items in common blocks will *not* (-align records with -align nocommons). In derived-type data, using the SEQUENCE statement prevents -align records from adding needed padding bytes to naturally align data items.

If your command line includes the -std, -std90, or -std95 options, then the compiler ignores -align dcommons and -align sequence. See Section 3.85.

5.5 Using Arrays Efficiently

The following sections discuss:

- Section 5.5.1, Accessing Arrays Efficiently
- Section 5.5.2, Passing Array Arguments Efficiently

5.5.1 Accessing Arrays Efficiently

On Alpha systems, many of the array access efficiency techniques described in this section are applied automatically by the Compaq Fortran loop transformation optimizations (see Section 5.8.7) or by the Compaq KAP Fortran/OpenMP for Tru64 UNIX Systems performance preprocessor (described in Section 5.1.1).

Several aspects of array use can improve run-time performance:

- The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array instead of numerous operations on scattered array elements.

  Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable A:

  ```
  A = A + 1.
  ```

  When reading or writing an array, use the array name and not a DO loop or an implied DO-loop that specifies each element number. Fortran 95/90 array syntax allows you to reference a whole array by using its name in an expression. For example:

  ```
  REAL ::  A(100,100)
  A = 0.0
  A = A + 1.                        ! Increment all elements of A by 1
  .
  .
  .
  WRITE (8) A                       ! Fast whole array use
  ```

  Similarly, you can use derived-type array structure components, such as:

```
TYPE X
  INTEGER A(5)
END TYPE X
.
.
.
TYPE (X) Z
WRITE (8) Z%A                       ! Fast array structure component use
```

- Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the **natural ascending storage order**, which is **column-major order** for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Whole array access uses column-major order.

  Avoid **row-major order**, as is done by C, where the rightmost subscript varies most rapidly.

  For example, consider the nested DO loops that access a two-dimension array with the J loop as the innermost loop:

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3                    ! I outer loop varies slowest
  DO J=1,5                  ! J inner loop varies fastest
    X (I,J) = Y(I,J) + 1    ! Inefficient row-major storage order
  END DO                    ! (rightmost subscript varies fastest)
END DO
.
.
.
END PROGRAM
```

  Since J varies the fastest and is the second array subscript in the expression X (I,J), the array is accessed in row-major order.

  To make the array accessed in natural column-major order, examine the array algorithm and data being modified.

  Using arrays X and Y, the array can be accessed in natural column-major order by changing the nesting order of the DO loops so the innermost loop variable corresponds to the leftmost array dimension:

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
```

```
DO J=1,5                    ! J outer loop varies slowest
  DO I=1,3                  ! I inner loop varies fastest
    X (I,J) = Y(I,J) + 1    ! Efficient column-major storage order
  END DO                    ! (leftmost subscript varies fastest)
END DO
.
.
.
END PROGRAM
```

The Compaq Fortran whole array access (`X = Y + 1`) uses efficient column major order. However, if the application requires that J vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application program to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays X(5,3) and Y(5,3)
- The assignment of X(J,I) and Y(J,I) within the DO loops
- All other references to arrays X and Y

In this case, the original DO loop nesting is used where J is the innermost loop:

```
INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3                    ! I outer loop varies slowest
  DO J=1,5                  ! J inner loop varies fastest
    X (J,I) = Y(J,I) + 1    ! Efficient column-major storage order
  END DO                    ! (leftmost subscript varies fastest)
END DO
.
.
.
END PROGRAM
```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make inefficient use of the CPU memory cache. For more information on using natural storage order during record I/O operations, see Section 5.6.3.

- Use the available Fortran 95/90 array intrinsic procedures rather than create your own.

  Whenever possible, use Fortran 95/90 array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran 95/90 array intrinsic procedures are designed for efficient use with the various Compaq Fortran run-time components.

  Using the standard-conforming array intrinsics can also make your program more portable.

- With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of two (such as 256, 512).

  Since the **cache sizes** are a power of 2, **array dimensions** that are also a power of 2 may make inefficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make little use of the cache. This does not apply to contiguous sequential access or whole array access.

  One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

```
REAL A (512,100)
DO I = 2,511
  DO J = 2,99
    A(I,J)=(A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

  In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of A to 520 (REAL A (520,100)) allows the loop to provide better performance, but at the expense of some unused elements.

  Because loop index variables I and J are used in the calculation, changing the nesting order of the DO loops changes the results.

  For More Information:

  - On arrays and their data declaration statements, see the *Compaq Fortran Language Reference Manual*.

5.5.2 Passing Array Arguments Efficiently

In Fortran 95/90, there are two general types of array arguments:

- Explicit-shape arrays used with FORTRAN 77.

  These arrays have a fixed rank and extent that is known at compile time. Other dummy argument (receiving) arrays that are not deferred-shape (such as assumed-size arrays) can be grouped with explicit-shape array arguments.

- Deferred-shape arrays introduced with Fortran 95/90.

Types of deferred-shape arrays include array pointers and allocatable arrays. Assumed-shape array arguments generally follow the rules about passing deferred-shape array arguments.

When passing arrays as arguments, either the starting (base) address of the array or the address of an array descriptor is passed:

- When using explicit-shape (or assumed-size) arrays to receive an array, the starting address of the array is passed.
- When using deferred-shape or assumed-shape arrays to receive an array, the address of the array descriptor is passed (the compiler creates the array descriptor).

Passing an assumed-shape array or array pointer to an explicit-shape array can slow run-time performance. This is because the compiler needs to create an array temporary for the entire array. The array temporary is created because the passed array may not be contiguous and the receiving (explicit-shape) array requires a contiguous array. When an array temporary is created, the size of the passed array determines whether the impact on slowing run-time performance is slight or severe.

Table 5–3 summarizes what happens with the various combinations of array types. The amount of run-time performance inefficiency depends on the size of the array.

Table 5–3 Output Argument Array Types

Input Arguments Array Types	Explicit-Shape Arrays	Deferred-Shape and Assumed-Shape Arrays
Explicit-shape arrays	Very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional.	Efficient. Only allowed for assumed-shape arrays (not deferred-shape arrays). Does not use an array temporary. Passes an array descriptor. Requires an interface block.
Deferred-shape and assumed-shape arrays	When passing an allocatable array, very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional. When not passing an allocatable array, not efficient. Instead use allocatable arrays whenever possible. Uses an array temporary. Does not pass an array descriptor. Interface block optional.	Efficient. Requires an assumed-shape or array pointer as dummy argument. Does not use an array temporary. Passes an array descriptor. Requires an interface block.

5.6 Improving Overall I/O Performance

Improving overall I/O performance can minimize both device I/O and actual CPU time. The techniques listed in this section can greatly improve performance in many applications.

A **bottleneck** limits the maximum speed of execution by being the slowest process in an executing program. In some programs, I/O is the bottleneck that prevents an improvement in run-time performance. The key to relieving I/O bottlenecks is to reduce the actual amount of CPU and I/O device time involved in I/O.

Bottlenecks can be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement in I/O time
- Such coding practices as:
  - Unnecessary formatting of data and other CPU-intensive processing

- Unnecessary transfers of intermediate results
- Inefficient transfers of small amounts of data
- Application requirements

Improved coding practices can minimize actual device I/O, as well as the actual CPU time.

Compaq offers software solutions to system-wide problems like minimizing device I/O delays (see Section 5.1.1).

5.6.1 Use Unformatted Files Instead of Formatted Files

Use unformatted files whenever possible. Unformatted I/O of numeric data is more efficient and more precise than formatted I/O. Native unformatted data does not need to be modified when transferred and will take up less space on an external file.

Conversely, when writing data to formatted files, formatted data must be converted to character strings for output, less data can transfer in a single operation, and formatted data may lose precision if read back into binary form.

To write the array `A(25,25)` in the following statements, S_1 is more efficient than S_2:

```
S1          WRITE (7) A

S2          WRITE (7,100) A
     100    FORMAT (25(' ',25F5.21))
```

Although formatted data files are more easily ported to other systems, Compaq Fortran can convert unformatted data in several formats (see Chapter 10).

5.6.2 Write Whole Arrays or Strings

The general guidelines about array use discussed in Section 5.5 also apply to reading or writing an array with an I/O statement.

To eliminate unnecessary overhead, write whole arrays or strings at one time rather than individual elements at multiple times. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in implied-DO loops. When accessing whole arrays, use the array name (Fortran 95/90 array syntax) instead of using implied-DO loops.

5.6.3 Write Array Data in the Natural Storage Order

Use the natural ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1. (See Section 5.5.1, Accessing Arrays Efficiently.) If a program must read or write data in any other order, efficient block moves are inhibited.

If the whole array is not being written, natural storage order is the best order possible.

If you must use an **unnatural storage order**, in certain cases it might be more efficient to transfer the data to memory and reorder the data before performing the I/O operation.

5.6.4 Use Memory for Intermediate Results

Performance can improve by storing intermediate results in memory rather than storing them in a file on a peripheral device. One situation that may not benefit from using intermediate storage is when there is a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

If you are primarily concerned with the CPU performance of the system, consider using a memory file system (mfs) virtual disk to hold any files your code reads or writes (see `mfs(1)`).

5.6.5 Enable Implied-DO Loop Collapsing

DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Compaq Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

Compaq Fortran reduces the number of calls in implied-DO loops by replacing up to seven nested implied-DO loops with a single call to an optimized run-time library I/O routine. The routine can transmit many I/O elements at once.

Loop collapsing can occur in formatted and unformatted I/O, but only if certain conditions are met:

- The control variable must be an integer. The control variable cannot be a dummy argument or contained in an EQUIVALENCE or VOLATILE statement. Compaq Fortran must be able to determine that the control variable does not change unexpectedly at run time.
- The format must not contain a variable format expression.

For More Information:

- On VOLATILE attribute and statement, see the *Compaq Fortran Language Reference Manual*.
- On loop optimizations, see Section 5.8.

5.6.6 Use of Variable Format Expressions

Variable format expressions (a Compaq Fortran extension) are almost as flexible as run-time formatting, but they are more efficient because the compiler can eliminate run-time parsing of the I/O format. Only a small amount of processing and the actual data transfer are required during run time.

On the other hand, run-time formatting can impair performance significantly. For example, in the following statements, S_1 is more efficient than S_2 because the formatting is done once at compile time, not at run time:

```
S1        WRITE (6,400) (A(I), I=1,N)
     400   FORMAT (1X, <N> F5.2)
                        .
                        .
                        .
S2        WRITE (CHFMT,500) '(1X,',N,'F5.2)'
    500   FORMAT (A,I3,A)
          WRITE (6,FMT=CHFMT) (A(I), I=1,N)
```

5.6.7 Efficient Use of Record Buffers and Disk I/O

Records being read or written are transferred between the user's program buffers and one or more disk block I/O buffers, which are established when the file is opened by the Compaq Fortran RTL. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimizing physical disk I/O.

You can specify the size of the disk block physical I/O buffer by using the OPEN statement BLOCKSIZE specifier; the default size can be obtained from `fstat`(2). If you omit the BLOCKSIZE specifier in the OPEN statement, it is set for optimal I/O use with the type of device the file resides on (with the exception of network access).

The OPEN statement BUFFERCOUNT specifier specifies the number of I/O buffers. The default for BUFFERCOUNT is 1. Any experiments to improve I/O performance should increase the BUFFERCOUNT value and not the BLOCKSIZE value, to increase the amount of data read by each disk I/O.

If the OPEN statement has BLOCKSIZE and BUFFERCOUNT specifiers, then the internal buffer size in bytes is the product of these specifiers. If the OPEN statement does not have these specifiers, then the default internal buffer size is 8192 bytes. This internal buffer will grow to hold the largest single record, but will never shrink.

The default for the Fortran run-time system is to use unbuffered disk writes. That is, by default, records are written to disk immediately as each record is written instead of accumulating in the buffer to be written to disk later.

To enable buffered writes (that is, to allow the disk device to fill the internal buffer before the buffer is written to disk), use one of the following:

1. The OPEN statement BUFFERED specifier
2. The `-assume buffered_io` command-line option
3. The `FORT_BUFFERED` run-time environment variable

The OPEN statement BUFFERED specifier takes precedence over the `-assume buffered_io` option. If neither one is set (which is the default), the `FORT_BUFFERED` environment variable is tested at run time.

The OPEN statement BUFFERED specifier applies to a specific logical unit. In contrast, the `-assume [no]buffered_io` option and the `FORT_BUFFERED` environment variable apply to all Fortran units.

Using buffered writes usually makes disk I/O more efficient by writing larger blocks of data to the disk less often. However, a system failure when using buffered writes can cause records to be lost, since they might not yet have been written to disk. (Such records would have been written to disk with the default unbuffered writes.)

When performing I/O across a network, be aware that the size of the block of network data sent across the network can impact application efficiency. When reading network data, follow the same advice for efficient disk reads, by increasing the BUFFERCOUNT. When writing data through the network, several items should be considered:

- Unless the application requires that records be written using unbuffered writes, enable buffered writes by a method described above.
- Especially with large files, increasing the BLOCKSIZE value increases the size of the block sent on the network and how often network data blocks get sent.
- Time the application when using different BLOCKSIZE values under similar conditions to find the optimal network block size.

When writing records, be aware that I/O records are written to unified buffer cache (UBC) system buffers. To request that I/O records be written from program buffers to the UBC system buffers, use the `flush` library routine (see `flush`(3f) and Chapter 12). Be aware that calling `flush` also discards read-ahead data in user buffer.

To request that UBC system buffers be written to disk, use the `fsync` library routine (see `fsync`(3f) and Chapter 12).

When UBC buffers are written to disk depends on UBC characteristics on the system, such as the vm-ubcbuffers attribute (see the *Compaq Tru64 UNIX System Tuning and Performance* guide).

For More Information:

- See Section 7.5, Opening Files: OPEN Statement.
- See Section 3.6, -assume buffered_io — Buffered Output.

5.6.8 Specify RECL

The sum of the record length (RECL specifier in an OPEN statement) and its overhead is a multiple or divisor of the blocksize, which is device specific. For example, if the BLOCKSIZE is 8192 then RECL might be 24576 (a multiple of 3) or 1024 (a divisor of 8).

The RECL value should fill blocks as close to capacity as possible (but not over capacity). Such values allow efficient moves, with each operation moving as much data as possible; the least amount of space in the block is wasted. Avoid using values larger than the block capacity, because they create very inefficient moves for the excess data only slightly filling a block (allocating extra memory for the buffer and writing partial blocks are inefficient).

The RECL value unit for formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units (see Section 3.7).

When porting unformatted data files from non-Compaq systems, see Section 10.6.

5.6.9 Use the Optimal Record Type

Unless a certain record type is needed for portability reasons (see Section 7.4.3), choose the most efficient type, as follows:

- For sequential files of a consistent record size, the fixed-length record type gives the best performance.

- For sequential unformatted files when records are not fixed in size, the variable-length record type gives the best performance—particularly for BACKSPACE operations.
- For sequential formatted files when records are not fixed in size, the Stream_LF record type gives the best performance.

5.6.10 Reading from a Redirected Standard Input File

Due to certain precautions that the Fortran run-time system takes to ensure the integrity of standard input, reads can be very slow when standard input is redirected from a file. For example, when you use a command such as `myprogram.exe < myinput.data`, the data is read using the READ(*) or READ(5) statement, and performance is degraded. To avoid this problem, do one of the following:

- Explicitly open the file using the OPEN statement. For example:

  ```
  OPEN(5, STATUS='OLD', FILE='myinput.dat')
  ```

- Use an environment variable to specify the input file.

  For example, if read from unit 5:

  ```
  setenv FORT5=myinput.dat
  ```

  or if read from unit *:

  ```
  setenv FOR_READ=myinput.dat
  ```

To take advantage of these methods, be sure your program does not rely on sharing the standard input file.

For More Information:

- On Compaq Fortran data files and I/O, see Chapter 7.
- On OPEN statement specifiers and defaults, see Section 7.5 and the *Compaq Fortran Language Reference Manual*.

5.7 Additional Source Code Guidelines for Run-Time Efficiency

Other source coding guidelines can be implemented to improve run-time performance.

The amount of improvement in run-time performance is related to the number of times a statement is executed. For example, improving an arithmetic expression executed within a loop many times has the potential to improve performance, more than improving a similar expression executed once outside a loop.

5.7.1 Avoid Small Integer and Small Logical Data Items

Avoid using integer or logical data less than 32 bits, because the smallest unit of efficient access on Alpha systems is 32 bits.

Accessing a 16-bit (or 8-bit) data type can result in a sequence of machine instructions to access the data, rather than a single, efficient machine instruction for a 32-bit data item.

To minimize data storage and memory cache misses with arrays, use 32-bit data rather than 64-bit data, unless you require the greater numeric range of 8-byte integers or the greater range and precision of double precision floating-point numbers.

5.7.2 Avoid Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (REAL) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that I and J are both INTEGER variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data:

Original Code:
```
INTEGER I, J
I = J / 2.
```

Efficient Code:
```
INTEGER I, J
I = J / 2
```

For applications with numerous floating-point operations, consider using the `-fp_reorder` option (see Section 5.9.7) if a small difference in the result is acceptable.

You can use different sizes of the same general data type in an expression with minimal or no effect on run-time performance. For example, using REAL, DOUBLE PRECISION, and COMPLEX floating-point numbers in the same floating-point arithmetic expression has minimal or no effect on run-time performance.

5.7.3 Use Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer (also see Section 5.7.1)
- Single-precision real, expressed explicitly as REAL, REAL (KIND=4), or REAL*4
- Double-precision real, expressed explicitly as DOUBLE PRECISION, REAL (KIND=8), or REAL*8
- Extended-precision real, expressed explicitly as REAL (KIND=16) or REAL*16

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point (REAL) data (see Section 5.7.2).

5.7.4 Avoid Using Slow Arithmetic Operators

Before you modify source code to avoid slow arithmetic operators, be aware that optimizations convert many slow arithmetic operators to faster arithmetic operators. For example, the compiler optimizes the expression H=J**2 to be H=J*J.

Consider also whether replacing a slow arithmetic operator with a faster arithmetic operator will change the accuracy of the results or impact the maintainability (readability) of the source code.

Replacing slow arithmetic operators with faster ones should be reserved for critical code areas. The following hierarchy lists the Compaq Fortran arithmetic operators, from fastest to slowest:

- Addition (+), subtraction (-), and floating-point multiplication (*)
- Integer multiplication (*)
- Division (/)
- Exponentiation (**)

5.7.5 Avoid Using EQUIVALENCE Statements

Avoid using EQUIVALENCE statements. EQUIVALENCE statements can:

- Force unaligned data or cause data to span natural boundaries.
- Prevent certain optimizations, including:
  - Global data analysis under certain conditions (see Section 5.8.3)
  - Implied-DO loop collapsing when the control variable is contained in an EQUIVALENCE statement

5.7.6 Use Statement Functions and Internal Subprograms

Whenever the Compaq Fortran compiler has access to the use and definition of a subprogram during compilation, it may choose to inline the subprogram. Using statement functions and internal subprograms maximizes the number of subprogram references that will be inlined, especially when multiple source files are compiled together at optimization level `-O4` or higher.

For More Information:

- See Section 5.1.2.

5.7.7 Code DO Loops for Efficiency

Minimize the arithmetic operations and other operations in a DO loop whenever possible. Moving unnecessary operations outside the loop will improve performance (for example, when the intermediate nonvarying values within the loop are not needed).

For More Information:

- On loop optimizations, see Section 5.8.6 and Section 5.9.2.
- On coding Compaq Fortran statements, see the *Compaq Fortran Language Reference Manual*.

5.8 Optimization Levels: the -O*n* Option

Compaq Fortran performs many optimizations by default. You do not have to recode your program to use them. However, understanding how optimizations work helps you remove any inhibitors to their successful function.

Generally, Compaq Fortran increases compile time in favor of decreasing run time. If an operation can be performed, eliminated, or simplified at compile time, Compaq Fortran does so, rather than have it done at run time. The time required to compile the program usually increases as more optimizations occur.

The program will likely execute faster when compiled at -O4, but will require more compilation time than if you compile the program at a lower level of optimization.

The size of object file varies with the optimizations requested. Factors that can increase object file size include an increase of loop unrolling or procedure inlining.

Table 5–4 lists the levels of Compaq Fortran optimization with different -O options. For example: -O0 specifies no selectable optimizations (some optimizations always occur); -O5 specifies all levels of optimizations, including loop transformation.

Table 5–4 Levels of Optimization with Different -O*n* Options

Optimization Type	–O0	–O1	–O2	–O3	–O4	–O5
Loop transformation						X
Software pipelining					X	X
Automatic inlining					X	X
Additional global optimizations				X	X	X
Global optimizations			X	X	X	X
Local (minimal) optimizations		X	X	X	X	X

The default is -O4 (same as -O). However, if -g2, -g, or -gen_feedback is also specified, the default is -O0 (no optimizations).

In Table 5–4, the following terms are used to describe the levels of optimization:

- **Local (minimal) optimizations** (-O1 or higher) occur within the source program unit and include recognition of common subexpressions and the expansion of multiplication and division. See Section 5.8.2, Local (Minimal) Optimizations.
- **Global optimizations** (-O2 or higher) include such optimizations as data flow analysis, code motion, strength reduction, split-lifetime analysis, and instruction scheduling. See Section 5.8.3, Global Optimizations.
- Additional global optimizations (-O3 or higher) improve speed at the cost of extra code size. These optimizations include loop unrolling, prefetching of data, and code replication to eliminate branches. See Section 5.8.4, Additional Global Optimizations.

- **Automatic inlining** (`-O4` or higher) applies interprocedure analysis and inline expansion of small procedures, usually by using heuristics that limit extra code. See Section 5.8.5, Automatic Inlining.
- **Software pipelining** (`-O4` or higher) applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution. Software pipelining also enables the prefetching of data to reduce the impact of cache misses.
- **Loop transformation** (`-O5`) optimizations apply to array references within loops and can apply to multiple nested loops. These optimizations can improve the performance of the memory system. See Section 5.8.7, Loop Transformation.

5.8.1 Optimizations Performed at All Optimization Levels

The following optimizations occur at any optimization level (`-O0` through `-O5`):

- **Space optimizations**

  Space optimizations decrease the size of the object or executing program by eliminating unnecessary use of memory, thereby improving speed of execution and system throughput. Compaq Fortran space optimizations are:

  - **Constant pooling**

    Only one copy of a given constant value is ever allocated memory space. If that constant value is used in several places in the program, all references point to that value.

  - **Dead code elimination**

    If operations will never execute or if data items will never be used, Compaq Fortran eliminates them. Dead code includes unreachable code and code that becomes unused as a result of other optimizations, such as value propagation.

- Inlining arithmetic statement functions and intrinsic procedures

  Regardless of the optimization level, Compaq Fortran inserts arithmetic statement functions directly into a program instead of calling them as functions. This permits other optimizations of the inlined code and eliminates several operations, such as calls and returns or stores and fetches of the actual arguments. For example:

```
SUM(A,B) = A+B
   .
   .
   .
Y = 3.14
X = SUM(Y,3.0)         ! With value propagation, becomes: X = 6.14
```

Most intrinsic procedures are automatically inlined.

Inlining of other subprograms, such as contained subprograms, occurs at optimization level -O4.

- Implied-DO loop collapsing

  DO loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Compaq Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

  If Compaq Fortran can determine that the format will not change during program execution, it replaces the series of calls in up to seven nested implied-DO loops with a single call to an optimized RTL routine (see Section 5.6.5). The optimized RTL routine can transfer many elements in one operation.

  Compaq Fortran collapses implied-DO loops in formatted and unformatted I/O operations, but it is more important with unformatted I/O, where the cost of transmitting the elements is a higher fraction of the total cost.

- Array temporary elimination and FORALL statements

  Certain array store operations are optimized. For example, to minimize the creation of array temporaries, Compaq Fortran can detect when no overlap occurs between the two sides of an array expression. This type of optimization occurs for some assignment statements in FORALL constructs.

  Certain array operations are also candidates for loop unrolling optimizations (see Section 5.8.4.1).

5.8.2 Local (Minimal) Optimizations

To enable local optimizations, use -O1 or a higher optimization level (-O2, -O3, -O4, or -O5).

To prevent local optimizations, specify the -O0 option.

5.8.2.1 Common Subexpression Elimination

If the same subexpressions appear in more than one computation and the values do not change between computations, Compaq Fortran computes the result once and replaces the subexpressions with the result itself:

```
DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)
```

Without optimization, these statements can be compiled as follows:

```
t1 = ((J-1)*25+(I-1))*4
t2 = ((J-1)*25+(I-1))*4
A(t1) = B(t2)
```

Variables t1 and t2 represent equivalent expressions. Compaq Fortran eliminates this redundancy by producing the following:

```
t = ((J-1)*25+(I-1))*4
A(t) = B(t)
```

5.8.2.2 Integer Multiplication and Division Expansion

Expansion of multiplication and division refers to bit shifts that allow faster multiplication and division while producing the same result. For example, the integer expression (I*17) can be calculated as I with a 4-bit shift plus the original value of I. This can be expressed using the Compaq Fortran ISHFT intrinsic function:

```
J1 = I*17
J2 = ISHFT(I,4) + I     ! equivalent expression for I*17
```

The optimizer uses machine code that, like the ISHFT intrinsic function, shifts bits to expand multiplication and division by literals.

5.8.2.3 Compile-Time Operations

Compaq Fortran does as many operations as possible at compile time rather than at run time.

Constant Operations

Compaq Fortran can perform many operations on constants (including PARAMETER constants):

- Constants preceded by a unary minus sign are negated.
- Expressions involving +, –, *, or / operators are evaluated; for example:

```
PARAMETER (NN=27)
I = 2*NN+J              ! Becomes: I = 54 + J
```

Evaluation of some constant functions and operators is performed at compile time. This includes certain functions of constants, concatenation of string constants, and logical and relational operations involving constants.

- Lower-ranked constants are converted to the data type of the higher-ranked operand:

```
REAL X, Y
X = 10 * Y              ! Becomes: X = 10.0 * Y
```

- Array address calculations involving constant subscripts are simplified at compile time whenever possible:

```
INTEGER I(10,10)
I(1,2) = I(4,5)         ! Compiled as a direct load and store
```

Algebraic Reassociation Optimizations

Compaq Fortran delays operations to see whether they have no effect or can be transformed to have no effect. If they have no effect, these operations are removed. A typical example involves unary minus and .NOT. operations:

```
X = -Y * -Z             ! Becomes: Y * Z
```

5.8.2.4 Value Propagation

Compaq Fortran tracks the values assigned to variables and constants, including those from DATA statements, and traces them to every place they are used. Compaq Fortran uses the value itself when it is more efficient to do so.

When compiling subprograms, Compaq Fortran analyzes the program to ensure that propagation is safe if the subroutine is called more than once.

Value propagation frequently leads to more value propagation. Compaq Fortran can eliminate run-time operations, comparisons and branches, and whole statements.

In the following example, constants are propagated, eliminating multiple operations from run time:

Original Code	Optimized Code

`PI = 3.14`
`.`
`.`
`.`
`PIOVER2 = PI/2`
`.`
`.`
`.`
`I = 100`
`.`
`.`
`.`
`IF (I.GT.1) GOTO 10`
`10  A(I) = 3.0*Q`
	`.`
	`.`
	`.`
	`PIOVER2 = 1.57`
	`.`
	`.`
	`.`
	`I = 100`
	`.`
	`.`
	`.`
	`10  A(100) = 3.0*Q`

5.8.2.5 Dead Store Elimination

If a variable is assigned but never used, Compaq Fortran eliminates the entire assignment statement:

```
X = Y*Z
   .
   .
   .              ! If X is not used in between, X=Y*Z is eliminated.
X = A(I,J)* PI
```

Some programs used for performance analysis often contain such unnecessary operations. When you try to measure the performance of such programs compiled with Compaq Fortran, these programs may show unrealistically good performance results. Realistic results are possible only with program units using their results in output statements.

5.8.2.6 Register Usage

A large program usually has more data that would benefit from being held in registers than there are registers to hold the data. In such cases, Compaq Fortran typically tries to use the registers according to the following descending priority list:

1. For temporary operation results, including array indexes
2. For variables
3. For addresses of arrays (base address)
4. All other usages

Compaq Fortran uses heuristic algorithms and a modest amount of computation to attempt to determine an effective usage for the registers.

Holding Variables in Registers

Because operations using registers are much faster than using memory, Compaq Fortran generates code that uses the Alpha 64-bit integer and floating-point registers instead of memory locations. Knowing when Compaq Fortran uses registers may be helpful when doing certain forms of debugging.

Compaq Fortran uses registers to hold the values of variables whenever the Fortran language does not require them to be held in memory, such as holding the values of temporary results of subexpressions, even if `-O0` (no optimization) was specified.

Compaq Fortran may hold the same variable in different registers at different points in the program:

```
V = 3.0*Q
   .
   .
   .
X = SIN(Y)*V
   .
   .
   .
V = PI*X
   .
   .
   .
Y = COS(Y)*V
```

Compaq Fortran may choose one register to hold the first use of `V` and another register to hold the second. Both registers can be used for other purposes at points in between. There may be times when the value of the variable does not exist anywhere in the registers. If the value of `V` is never needed in memory, it is never assigned.

Compaq Fortran uses registers to hold the values of I, J, and K (so long as there are no other optimization effects, such as loops involving the variables):

```
A(I) = B(J) + C(K)
```

More typically, an expression uses the same index variable:

```
A(K) = B(K) + C(K)
```

In this case, K is loaded into only one register and is used to index all three arrays at the same time.

5.8.2.7 Mixed Real/Complex Operations

In mixed REAL/COMPLEX operations, Compaq Fortran avoids the conversion and performs a simplified operation on:

- Add (+), subtract (–), and multiply (*) operations if either operand is REAL
- Divide (/) operations if the right operand is REAL

For example, if variable R is REAL and A and B are COMPLEX, no conversion occurs with the following:

```
COMPLEX A, B
   .
   .
   .
B = A + R
```

5.8.3 Global Optimizations

To enable global optimizations, use `-O2` or a higher optimization level (`-O3`, `-O4`, or `-O5`). Using `-O2` or higher also enables local optimizations (`-O1`).

Global optimizations include:

- **Data flow analysis**
- **Split lifetime analysis**
- **Strength reduction** (replaces a CPU-intensive calculation with one that uses fewer CPU cycles)
- **Code motion** (also called code hoisting)
- **Instruction scheduling**

Data flow analysis and split lifetime analysis (global data analysis) traces the values of variables and whole arrays as they are created and used in different parts of a program unit. During this analysis, Compaq Fortran assumes that any pair of array references to a given array might access the same memory location, unless a constant subscript is used in both cases.

To eliminate unnecessary recomputations of invariant expressions in loops, Compaq Fortran hoists them out of the loops so they execute only once.

Global data analysis includes which data items are selected for analysis. Some data items are analyzed as a group and some are analyzed individually. Compaq Fortran limits or may disqualify data items that participate in the following constructs, generally because it cannot fully trace their values:

- VOLATILE declarations

VOLATILE declarations are needed to use certain run-time features of the operating system. Declare a variable as VOLATILE if the variable can be accessed using rules in addition to those provided by the Fortran 95/90 language. Examples include:

- COMMON data items or entire common blocks that can change value by means other than direct assignment or during a routine call. For such applications, you must declare the variable or the COMMON block to which it belongs as volatile.
- An address not saved by the %LOC built-in function.
- Variables read or written by a signal handler, including those in a common block or module.

As requested by the VOLATILE statement, Compaq Fortran disqualifies any volatile variables from global data analysis.

- Subroutine calls or external function references

  Compaq Fortran cannot trace data flow in a called routine that is not part of the program unit being compiled, unless the same `f90` command compiled multiple program units (see Section 5.1.2). Arguments passed to a called routine that are used again in a calling program are assumed to be modified, unless the proper INTENT is specified in an interface block (the compiler must assume they are referenced by the called routine).

- Common blocks

  Compaq Fortran limits optimizations on data items in common blocks. If common block data items are referenced inside called routines, their values might be altered. In the following example, variable I might be altered by FOO, so Compaq Fortran cannot predict its value in subsequent references.

```
COMMON /X/ I

DO J=1,N
   I = J
   CALL FOO
   A(I) = I
ENDDO
```

- Variables in Fortran 95/90 modules

  Compaq Fortran limits optimizations on variables in Fortran 95/90 modules. Like common blocks, if the variables in Fortran 95/90 modules are referenced inside called routines, their values might be altered.

- Variables referenced by a %LOC built-in function or variables with the TARGET attribute

Compaq Fortran limits optimizations on variables indirectly referenced by a %LOC function or on variables with the TARGET attribute, because the called routine may dereference a pointer to such a variable.

- Equivalence groups

  An **equivalence group** is formed explicitly with the EQUIVALENCE statement or implicitly by the COMMON statement. A program section is a particular common block or local data area for a particular routine. Compaq Fortran combines equivalence groups within the same program section and in the same program unit.

  The equivalence groups in separate program sections are analyzed separately, but the data items within each group are not, so some optimizations are limited to the data within each group.

5.8.4 Additional Global Optimizations

To enable additional global optimizations, use `-O3` or a higher optimization level (`-O4` or `-O5`). Using `-O3` or higher also enables local optimizations (`-O1`) and global optimizations (`-O2`).

Additional global optimizations improve speed at the cost of longer compile times and possibly extra code size.

5.8.4.1 Loop Unrolling

At optimization level `-O3` or above, Compaq Fortran attempts to unroll certain innermost loops, minimizing the number of branches and grouping more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow.

As more loops are unrolled, the average size of basic blocks increases. Loop unrolling generates multiple copies of the code for the loop body (loop code iterations) in a manner that allows efficient instruction pipelining.

The loop body is replicated a certain number of times, substituting index expressions. An initialization loop might be created to align the first reference with the main series of loops. A remainder loop might be created for leftover work.

The loop unroller also inserts data prefetches for arrays with affine subscripts. Prefetches (that is, prefetch instructions) can be inserted even if the unroller chooses not to unroll. On some architectures (21264 and later), write-hint instructions are also generated.

The number of times a loop is unrolled can be determined either by the optimizer or by using the `-unroll` *num* option, which can specify the limit for loop unrolling. Unless the user specifies a value, the optimizer will choose an unroll amount that minimizes the overhead of prefetching while also limiting code size expansion.

Array operations are often represented as a nested series of loops when expanded into instructions. The innermost loop for the array operation is the best candidate for loop unrolling (like DO loops). For example, the following array operation (once optimized) is represented by nested loops, where the innermost loop is a candidate for loop unrolling:

```
A(1:100,2:30) = B(1:100,1:29) * 2.0
```

For More Information:

- See Section 3.94, -unroll num — Specify Number for Loop Unroll Optimization.

5.8.4.2 Code Replication to Eliminate Branches

In addition to loop unrolling and other optimizations, the number of branches are reduced by replicating code that will eliminate branches. Code replication decreases the number of basic blocks and increases instruction-scheduling opportunities.

Code replication normally occurs when a branch is at the end of a flow of control, such as a routine with multiple, short exit sequences. The code at the exit sequence gets replicated at the various places where a branch to it might occur.

For example, consider the following unoptimized routine and its optimized equivalent that uses code replication (R0 is register 0):

Unoptimized Instructions	Optimized (Replicated) Instructions
.	.
.	.
.	.
branch to exit1	move 1 into R0
.	return
.	.
.	.
branch to exit1	.
.	move 1 into R0
.	return
.	.
exit1: move 1 into R0	.
return	.
	move 1 into R0
	return

Similarly, code replication can also occur within a loop that contains a small amount of shared code at the bottom of a loop and a case-type dispatch within the loop. The loop-end test-and-branch code might be replicated at the end of each case to create efficient instruction pipelining within the code for each case.

5.8.5 Automatic Inlining

To enable optimizations that perform automatic inlining, use `-O4` or a higher optimization level (`-O5`). Using `-O4` also enables local optimizations (`-O1`), global optimizations (`-O2`), and additional global optimizations (`-O3`).

The default is `-O4` (unless `-g2`, `-g`, or `-gen_feedback` is specified).

5.8.5.1 Interprocedure Analysis

Compiling multiple source files at optimization level `-O4` or higher lets the compiler examine more code for possible optimizations, including multiple program units. This results in:

- Inlining more procedures
- More complete global data analysis
- Reducing the number of external references to be resolved during linking

As more procedures are inlined, the size of the executable program and compile times may increase, but execution time should decrease.

5.8.5.2 Inlining Procedures

Inlining refers to replacing a subprogram reference (such as a CALL statement or function invocation) with the replicated code of the subprogram. As more procedures are inlined, global optimizations often become more effective.

The optimizer inlines small procedures, limiting inlining candidates based on such criteria as:

- Estimated size of code
- Number of call sites
- Use of constant arguments

You can specify:

- One of the `-O`*n* options to control the optimization level. For example, specifying `-O4` or higher enables interprocedure optimizations.

  Different `-O`*n* options set `-inline` *xxxx* options. For example, `-O4` sets `-inline speed`.

- One of the `-inline` *xxxx* options to directly control the inlining of procedures (see Section 5.9.3). For example, `-inline speed` inlines more procedures than `-inline size`.

5.8.6 Software Pipelining

Software pipelining and additional software dependence analysis are enabled by using the `-pipeline` option, the `-O4` option, or the `-O5` option. Software pipelining in certain cases improves run-time performance.

Software pipelining applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution.

Software pipelining also includes associated additional software dependence analysis and enables the prefetching of data to reduce the impact of cache misses.

Loop unrolling (enabled at `-O3` or above) *cannot* schedule across iterations of a loop. Because software pipelining *can* schedule across loop iterations, it can perform more efficient scheduling to eliminate instruction stalls within loops.

For instance, if software dependence analysis of data flow reveals that certain calculations can be done before or after that iteration of the loop, software pipelining reschedules those instructions ahead of or behind that loop iteration, at places where their execution can prevent instruction stalls or otherwise improve performance.

Software pipelining can be more effective when you combine `-pipeline` (or `-O4` or `-O5`) with the appropriate `-tune` *keyword* for the target Alpha processor generation (see Section 5.9.4).

To specify software pipelining without loop transformation optimizations, do one of the following:

- Specify `-O5` with `-notransform_loops` (preferred method)
- Specify `-O4`
- Specify `-pipeline` with `-O3` or `-O2`

This optimization is not performed at optimization levels below `-O2`.

Loops chosen for software pipelining:

- Are always innermost loops (those executed the most).
- Do not contain branches or procedure calls.
- Do not use COMPLEX floating-point data.

By modifying the unrolled loop and inserting instructions as needed before and/or after the unrolled loop, software pipelining generally improves run-time performance, except where the loops contain a large number of instructions with many existing overlapped operations. In this case, software pipelining may not have enough registers available to effectively improve execution performance. Run-time performance using `-O4` or `-O5` (or `-pipeline`) may not improve performance, as compared to using `-O3`.

This option might increase compilation time and/or program size. For programs that contain loops that exhaust available registers, longer execution times may occur. In this case, specify options `-unroll 1` or `-unroll 2` with the `-pipeline` option.

To determine whether using `-pipeline` benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without software pipelining (such as with `-pipeline` and `-nopipeline`).

For programs that contain loops that exhaust available registers, longer execution times may result with `-O4` or `-O5`, requiring use of `-unroll` *n* to limit loop unrolling (see Section 3.94).

For More Information:

- On the interaction of command-line options and timing programs compiled with software pipelining, see Section 3.76.

5.8.7 Loop Transformation

The loop transformation optimizations are enabled by using the `-transform_loops` option or the `-O5` option. Loop transformation attempts to improve performance by rewriting loops to make better use of the memory system. By rewriting loops, the loop transformation optimizations can increase the number of instructions executed, which can degrade the run-time performance of some programs.

To request loop transformation optimizations without software pipelining, do one of the following:

- Specify `-O5` with `-nopipeline` (preferred method)
- Specify `-transform_loops` with `-O4`, `-O3`, or `-O2`

This optimization is not performed at optimization levels below `-O2`.

You must specify `-notransform_loops` if you want this type of optimization disabled and you are also specifying `-O5`.

The loop transformation optimizations apply to array references within loops. These optimizations can improve the performance of the memory system and usually apply to multiple nested loops.

The loops chosen for loop transformation optimizations are always **counted loops**. Counted loops use a variable to count iterations, thereby determining the number of iterations before entering the loop. For example, DO and IF loops are normally counted loops, but uncounted DO WHILE loops are not.

Conditions that typically prevent the loop transformation optimizations from occurring include subprogram references that are not inlined (such as an external function call), complicated exit conditions, and uncounted loops.

The types of optimizations associated with `-transform_loops` include the following:

- **Loop blocking**—Can minimize memory system use with multidimensional array elements by completing as many operations as possible on array elements currently in the cache. Also known as loop tiling.
- **Loop distribution**—Moves instructions from one loop into separate, new loops. This can reduce the amount of memory used during one loop so that the remaining memory may fit in the cache. It can also create improved opportunities for loop blocking.

- **Loop fusion**—Combines instructions from two or more adjacent loops that use some of the same memory locations into a single loop. This can avoid the need to load those memory locations into the cache multiple times and improves opportunities for instruction scheduling.
- **Loop interchange**—Changes the nesting order of some or all loops. This can minimize the stride of array element access during loop execution and reduce the number of memory accesses needed. Also known as loop permutation.
- **Scalar replacement**—Replaces the use of an array element with a scalar variable under certain conditions.
- **Outer loop unrolling**—Unrolls the outer loop inside the inner loop under certain conditions to minimize the number of instructions and memory accesses needed. This also improves opportunities for instruction scheduling and scalar replacement.

To determine whether using `-transform_loops` benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without loop transformation optimizations (such as with `-transform_loops` and `-notransform_loops`).

For More Information:

- See Section 3.89, -transform_loops — Activate Loop Transformation Optimizations.

5.9 Other Options Related to Optimization

In addition to the `-O`*n* options (discussed in Section 5.8), several other `f90` command options can prevent or facilitate improved optimizations.

5.9.1 Setting Multiple Options with the -fast Option

Specifying the `-fast` option sets many performance options. For details, see Section 3.40, -fast — Set Options to Improve Run-Time Performance.

5.9.2 Controlling the Number of Times a Loop Is Unrolled

You can specify the number of times a loop is unrolled by using the `-unroll` *num* option (see Section 3.94).

The `-unroll` *num* option can also influence the run-time results of software pipelining optimizations performed when you specify one of the following:

- `-O5`
- `-O4`

- `-pipeline` with `-O3` or `-O2`

Although unrolling loops usually improves run-time performance, the size of the executable program may increase.

For More Information:

- See Section 5.8.4.1, Loop Unrolling.

5.9.3 Controlling the Inlining of Procedures

To specify the types of procedures to be inlined, use the `-inline` *keyword* option. Also, compile multiple source files together and specify an adequate optimization level, such as `-O4`.

If you omit `-noinline` and the `-inline` *keyword* options, the optimization level `-O`*n* option used determines the types of procedures that are inlined.

Maximizing the types of procedures that are inlined usually improves run-time performance, but compile-time memory usage and the size of the executable program may increase.

To determine whether using `-inline all` benefits your particular program, time program execution for the same program compiled with and without `-inline all`.

For More Information:

- See Section 3.54, -inline keyword, -noinline — Control Procedure Inlining.
- See Section 5.8.5.2, Inlining Procedures.

5.9.4 Requesting Optimized Code for a Specific Processor Generation

You can specify the types of optimized code to be generated by using the `-tune` *keyword* and `-arch` *keyword* options. Regardless of the specified keyword, the generated code will run correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can improve run-time performance; it is also possible that code tuned for a specific target may run slower on another target.

Specifying the correct keyword for `-tune` *keyword* for the target processor generation type usually slightly improves run-time performance. Unless you request software pipelining, the run-time performance difference for using the wrong keyword for `-tune` *keyword* (such as using `-tune ev4` for an ev5 processor) is usually less than 5%. When using software pipelining (using `-O4` or `-O5`) with `-tune` *keyword*, the difference can be more than 5%.

The combination of the specified keyword for -tune *keyword* and the type of processor generation used has no effect on producing the expected correct program results.

For More Information:

- See Section 3.90, -tune keyword — Specify Alpha Processor Implementation.

5.9.5 Requesting the Speculative Execution Optimization

*(TU*X only)* Speculative execution reduces instruction latency stalls to improve run-time performance for certain programs or routines. Speculative execution evaluates conditional code (including exceptions) and moves instructions that would otherwise be executed conditionally to a position before the test, so they are executed unconditionally.

The default, -speculate none, means that the speculative execution code scheduling optimization is not used and exceptions are reported as expected. You can specify -speculate all or -speculate by_routine to request the speculative execution optimization.

Performance improvements may be reduced because the run-time system must dismiss exceptions caused by speculative instructions. For certain programs, longer execution times may result when using the speculative execution optimization. To determine whether using -speculate all or -speculate by_routine benefits your particular program, you should time the program execution with one of these options for the same program compiled with -speculate none (default).

Speculative execution does not support some run-time error checking, since exception and signal processing (including SIGSEGV, SIGBUS, and SIGFPE) is conditional. When the program needs to be debugged or while you are testing for errors, only use -speculate none.

For More Information:

- On -speculate all or -speculate by_routine and the interaction with other command-line options, see Section 3.84.

5.9.6 Request Nonshared Object Optimizations

When you specify -non_shared to request a nonshared object file, you can specify the -om option to request code optimizations after linking, including nop (No Operation) removal, .lita removal, and reallocation of common symbols. This option also positions the global pointer register so the maximum addresses fall in the global-pointer window.

For More Information:

- On the -WL,*arg* command-line options that enable nonshared object file code optimizations, see Section 3.73.

5.9.7 Arithmetic Reordering Optimizations

If you use the -fp_reorder option (or -assume noaccuracy_sensitive, which are equivalent), Compaq Fortran may reorder code (based on algebraic identities) to improve performance.

For example, the following expressions are mathematically equivalent but may not compute the same value using finite precision arithmetic:

```
X = (A + B) + C
X = A + (B + C)
```

The results can be slightly different from the default -no_fp_reorder because of the way intermediate results are rounded. However, the -no_fp_reorder results are not categorically less accurate than those gained by the default. In fact, dot product summations using -fp_reorder can produce more accurate results than those using -no_fp_reorder.

The effect of -fp_reorder is important when Compaq Fortran hoists divide operations out of a loop. If -fp_reorder is in effect, the unoptimized loop becomes the optimized loop:

Unoptimized Code	Optimized Code
	T = 1/V
DO I=1,N	DO I=1,N
.	.
.	.
.	.
B(I) = A(I)/V	B(I) = A(I)*T
END DO	END DO

The transformation in the optimized loop increases performance significantly, and loses little or no accuracy. However, it does have the potential for raising overflow or underflow arithmetic exceptions.

The compiler can also reorder code based on algebraic identities to improve performance if you specify -fast.

5.9.8 Dummy Aliasing Assumption

Some programs compiled with Compaq Fortran (or Compaq Fortran 77) may have results that differ from the results of other Fortran compilers. Such programs may be aliasing dummy arguments to each other or to a variable in a common block or shared through use association, and at least one variable access is a store.

This program behavior is prohibited in programs conforming to the Fortran 95/90 standards, but not by Compaq Fortran. Other versions of Fortran allow dummy aliases and check for them to ensure correct results. However, Compaq Fortran assumes that no dummy aliasing will occur, and it can ignore potential data dependencies from this source in favor of faster execution.

The Compaq Fortran default is safe for programs conforming to the Fortran 95/90 standards. It will improve performance of these programs, because the standard prohibits such programs from passing overlapped variables or arrays as actual arguments if either is assigned in the execution of the program unit.

The `-assume dummy_aliases` option allows dummy aliasing. It ensures correct results by assuming the exact order of the references to dummy and common variables is required. Program units taking advantage of this behavior can produce inaccurate results if compiled with `-assume nodummy_aliases`.

Example 5–1 is taken from the DAXPY routine in the Fortran-77 version of the Basic Linear Algebra Subroutines (BLAS).

Example 5–1 Using the -assume dummy_aliases Option

```
      SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)

C     Constant times a vector plus a vector.
C     uses unrolled loops for increments equal to 1.

      DOUBLE PRECISION DX(1), DY(1), DA
      INTEGER I,INCX,INCY,IX,IY,M,MP1,N
C
      IF (N.LE.0) RETURN
      IF (DA.EQ.0.0) RETURN
      IF (INCX.EQ.1.AND.INCY.EQ.1) GOTO 20

C     Code for unequal increments or equal increments
C     not equal to 1.
      .
      .
      .
      RETURN
```

(continued on next page)

Example 5–1 (Cont.) Using the -assume dummy_aliases Option

```
C     Code for both increments equal to 1.
C     Clean-up loop

20    M = MOD(N,4)
      IF (M.EQ.0) GOTO 40
      DO I=1,M
          DY(I) = DY(I) + DA*DX(I)
      END DO

      IF (N.LT.4) RETURN
40    MP1 = M + 1
      DO I = MP1, N, 4
          DY(I) = DY(I) + DA*DX(I)
          DY(I + 1) = DY(I + 1) + DA*DX(I + 1)
          DY(I + 2) = DY(I + 2) + DA*DX(I + 2)
          DY(I + 3) = DY(I + 3) + DA*DX(I + 3)
      END DO

      RETURN
      END SUBROUTINE
```

The second DO loop contains assignments to DY. If DY is overlapped with DA, any of the assignments to DY might give DA a new value, and this overlap would affect the results. If this overlap is desired, then DA must be fetched from memory each time it is referenced. The repetitious fetching of DA degrades performance.

Linking Routines with Opposite Settings

You can link routines compiled with the `-assume dummy_aliases` option to routines compiled with `-assume nodummy_aliases`. For example, if only one routine is called with dummy aliases, you can use `-assume dummy_aliases` when compiling that routine, and compile all the other routines with `-assume nodummy_aliases` to gain the performance value of that option.

Programs calling DAXPY with DA overlapping DY do not conform to the FORTRAN-77 and Fortran 95/90 standards. However, they are supported if `-assume dummy_aliases` was used to compile the DAXPY routine.

6

Parallel Compiler Directives and Their Programming Environment

Note

The information in this chapter pertains only to Compaq Fortran on Tru64 UNIX systems.

This chapter describes two sets of parallel compiler directives:

- Section 6.1, OpenMP Fortran API Compiler Directives
- Section 6.2, Compaq Fortran Parallel Compiler Directives

The following topics apply to *both* the OpenMP Fortran API and the Compaq Fortran parallel compiler directives:

- Section 6.3, Decomposing Loops for Parallel Processing
- Section 6.4, Environment Variables for Adjusting the Run-Time Environment
- Section 6.5, Calls to Programs Written in Other Languages
- Section 6.6, Compiling, Linking, and Running Parallelized Programs on SMP Systems
- Section 6.7, Debugging Parallelized Programs

Note

The compiler can recognize either OpenMP directives or Compaq Fortran directives in a program, but not members of both sets of directives in a program.

For reference material on both sets of parallel compiler directives, see the *Compaq Fortran Language Reference Manual*.

6.1 OpenMP Fortran API Compiler Directives

Note

These directives comply with OpenMP Fortran 1.1 Application Program Interface, as described in the specification at:

`http://www.openmp.org/specs/`

These topics are described:

- Command-line option and directives format (see Section 6.1.1)
- Directive summary descriptions (see Section 6.1.2)
- Parallel processing thread model (see Section 6.1.3)
- Privatizing named common blocks (see Section 6.1.4)
- Controlling data scope attributes (see Section 6.1.5)
- Parallel region (see Section 6.1.6)
- Worksharing constructs (see Section 6.1.7)
- Combined parallel/worksharing constructs (see Section 6.1.8)
- Synchronization constructs (see Section 6.1.9)
- Specifying schedule type and chunk size (see Section 6.1.10)

6.1.1 Command-Line Option and Directives Format

To use OpenMP Fortran API compiler directives in your program, you must include the `-omp` compiler option on your `f90` command:

```
% f90 -omp prog.f -o prog
```

Directives are structured so that they appear to be Compaq Fortran comments. The format of an OpenMP Fortran API compiler directive is:

```
prefix directive_name [clause[[,] clause]...]
```

All OpenMP Fortran API compiler directives must begin with a directive prefix. Directives are not case-sensitive. Clauses can appear in any order after the directive name and can be repeated as needed, subject to the restrictions of individual clauses.

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

6.1.1.1 Directive Prefixes

The directive prefix you use depends on the source form you use in your program:

- Use the !$OMP prefix when compiling either fixed source form or free source form programs.
- Use the C$OMP and the *$OMP prefixes only when compiling fixed source form programs.

Fixed Source Form

For fixed source form programs, the prefix is one of the following:

```
!$OMP
C$OMP
*$OMP
```

Prefixes must start in column 1 and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.

Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6. For example, the following formats for specifying directives are equivalent:

```
c23456789
!$OMP PARALLEL DO SHARED(A,B,C)
!Is the same as...
c$OMP PARALLEL DO
c$OMP+SHARED(A,B,C)
!Which is the same as...
c$OMP PARALLEL DO SHARED(A,B,C)
```

Free Source Form

For free source form programs, the prefix is !$OMP.

The prefix can appear in any column as long as it is preceded only by white space. It must appear as a single string with no intervening white space. Free-form source rules apply to the directive line.

Initial directive lines must have a space after the prefix. Continued directive lines must have an ampersand as the last nonblank character on the line. Continuation directive lines can have an ampersand after the directive prefix with optional white space before and after the ampersand. For example, the following formats for specifying directives are equivalent:

```
!$OMP PARALLEL DO &
!$OMP SHARED(A,B,C)
!The same as...
!$OMP PARALLEL &
!$OMP&DO SHARED(A,B,C)
!Which is the same as...
!$OMP PARALLEL DO SHARED(A,B,C)
```

6.1.1.2 Directive Prefixes for Conditional Compilation

OpenMP Fortran API allows you to conditionally compile Compaq Fortran statements. The directive prefix you use for conditional compilation statements depends on the source form you use in your program:

- Use the !$ prefix when compiling either fixed source form or free source form programs.
- Use the C$ (or c$) and the *$ prefixes only when compiling fixed source form programs.

The prefix must be followed by a legal Compaq Fortran statement on the same line. When you use the `-omp` compiler option, the prefix is replaced by two spaces and the rest of the line is treated as a normal Compaq Fortran statement during compilations. You can also use the C preprocessor macro `_OPENMP` for conditional compilation.

Fixed Source Form

For fixed source form programs, the conditional compilation prefix is one of the following: !$, C$ (or c$), or *$.

The prefix must start in column 1 and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.

Initial lines must have a space or zero in column 6, and continuation lines must have a character other than a space or zero in column 6. For example, the following forms for specifying conditional compilation are equivalent:

```
c23456789
!$    IAM = OMP_GET_THREAD_NUM() +
!$   * INDEX

#IFDEF _OPENMP
      IAM = OMP_GET_THREAD_NUM() +
     * INDEX
#ENDIF
```

Free Source Form

The free source form conditional compilation prefix is !$. This prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Free-form source rules apply to the directive line.

Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix with optional white space before and after the ampersand.

6.1.2 Summary Descriptions of OpenMP Fortran API Compiler Directives

Table 6–1 provides summary descriptions of the OpenMP Fortran API compiler directives. For complete information about the OpenMP Fortran API compiler directives, see the *Compaq Fortran Language Reference Manual*.

Table 6–1 OpenMP Fortran API Compiler Directives

Directive Format	Description
prefix ATOMIC	This directive defines a synchronization construct that ensures that a specific memory location is updated atomically. See Section 6.1.9.1, ATOMIC Directive.
prefix BARRIER	This directive defines a synchronization construct that synchronizes all the threads in a team. See Section 6.1.9.2, BARRIER Directive.
prefix CRITICAL [(name)] *block* **prefix END CRITICAL [(name)]**	

(continued on next page)

Table 6–1 (Cont.) OpenMP Fortran API Compiler Directives

Directive Format	Description
	These directives define a synchronization construct that restricts access to the contained code to only one thread at a time. See Section 6.1.9.3, CRITICAL and END CRITICAL Directives.
prefix DO [clause[[,] clause] . . .] *do_loop* **[prefix END DO [NOWAIT]]**	These directives define a worksharing construct that specifies that the iterations of the DO loop are executed in parallel. See Section 6.1.7.1, DO and END DO directives.
prefix FLUSH [(var[,var] . . .)]	This directive defines a synchronization construct that identifies the precise point at which a consistent view of memory is provided. See Section 6.1.9.4, FLUSH Directive.
prefix MASTER *block* **prefix END MASTER**	These directives define a synchronization construct that specifies that the contained block of code is to be executed only by the master thread of the team. See Section 6.1.9.5, MASTER and END MASTER Directives.
prefix ORDERED *block* **prefix END ORDERED**	

(continued on next page)

Table 6–1 (Cont.) OpenMP Fortran API Compiler Directives

Directive Format	Description
	These directives define a synchronization construct that specifies that the contained block of code is executed in the order in which iterations would be executed during a sequential execution of the loop. See Section 6.1.9.6, ORDERED and END ORDERED Directives.
prefix PARALLEL [clause[[,] clause] . . .] *block* **prefix END PARALLEL**	These directives define a parallel construct that is a region of a program that must be executed by a team of threads until the END PARALLEL directive is encountered. See Section 6.1.6, Parallel Region: PARALLEL and END PARALLEL Directives.
prefix PARALLEL DO [clause[[,] clause] . . .] *do_loop* **prefix END PARALLEL DO**	These directives define a combined parallel/worksharing construct that is an abbreviated form of specifying a parallel region that contains a single DO directive. See Section 6.1.8.1, PARALLEL DO and END PARALLEL DO Directives.
prefix PARALLEL SECTIONS [clause[[,] clause] . . .] *block* **prefix END PARALLEL SECTIONS**	These directives define a combined parallel/worksharing construct that is an abbreviated form of specifying a parallel region that contains a single SECTIONS directive. See Section 6.1.8.2, PARALLEL SECTIONS and END PARALLEL SECTIONS Directives.

(continued on next page)

Table 6–1 (Cont.) OpenMP Fortran API Compiler Directives

Directive Format	Description
prefix SECTIONS [clause[[,] clause] . . .] **[prefix SECTION]** *block* **[prefix SECTION** *block***]** . . . **prefix END SECTIONS [NOWAIT]**	These directives define a worksharing construct that specifies that the enclosed sections of code are to be divided among threads in the team. Each section is executed once by some thread in the team. See Section 6.1.7.2, SECTIONS, SECTION, and END SECTIONS Directives.
prefix SINGLE [clause[[,] clause] . . .] *block* **prefix END SINGLE [NOWAIT]**	These directives define a worksharing construct that specifies that the enclosed code is to be executed by only one thread in the team. See Section 6.1.7.3, SINGLE and END SINGLE Directives.
prefix THREADPRIVATE(/cb/[,/cb/] . . .)	This data environment directive makes named common blocks private to a thread, but global within the thread. See Section 6.1.4, Privatizing Named Common Blocks: THREADPRIVATE Directive.

6.1.3 Parallel Processing Thread Model

A program containing OpenMP Fortran API compiler directives begins execution as a single process, called the **master thread** of execution. The master thread executes sequentially until the first parallel construct is encountered.

In OpenMP Fortran API, the PARALLEL and END PARALLEL directives define the **parallel construct**. When the master thread encounters a parallel construct, it creates a **team** of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the **static extent** of the construct. The **dynamic extent** includes the static extent as well as the routines called from within the construct. When the END PARALLEL directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.

You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called **orphaned directives**. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines.

For example:

```
subroutine F
...
!$OMP parallel...
...
    call G
...
subroutine G
...
!$OMP DO...
...
```

The !$OMP DO is an orphaned directive because the parallel region it will execute in is not lexically present in G.

A **parallel region** is a block of code that must be executed by a team of threads in parallel.

A **worksharing construct** is the heart of parallel processing. A worksharing construct divides the execution of the enclosed code region among the members of the team created on entering the enclosing parallel region.

A **combined parallel/worksharing construct** denotes a parallel region that contains only one worksharing construct.

Synchronization is the interthread communication that ensures the consistency of shared data and coordinates parallel execution among threads. Shared data is consistent within a team of threads when all threads obtain the identical value when the data is accessed. A **synchronization construct** is used to assure this consistency of shared data.

A **data environment directive** controls the data environment during the execution of parallel constructs.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize named common blocks (see Section 6.1.4)
- Control data scope attributes (see Section 6.1.5)

6.1.4 Privatizing Named Common Blocks: THREADPRIVATE Directive

You can make named common blocks private to a thread, but global within the thread, by using the THREADPRIVATE directive.

Each thread gets its own copy of the common block with the result that data written to the common block by one thread is not directly visible to other threads. During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block.

You cannot use a thread private common block or its constituent variables in any clause other than the COPYIN clause.

In the following example, common blocks BLK1 and FIELDS are specified as thread private:

```
      COMMON /BLK1/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/,/FIELDS/)
```

6.1.5 Controlling Data Scope Attributes

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is SHARED for those variables affected by the directive.

Each of the data scope attribute clauses accepts a *list*, which is a comma-separated list of named variables or named common blocks that are accessible in the scoping unit. When you specify named common blocks, they must appear between slashes (/*name*/).

Not all of the clauses are allowed on all directives, but the directives to which each clause applies are listed in the clause descriptions.

The data scope attribute clauses are:

- COPYIN
- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

COPYIN Clause

Use the COPYIN clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to copy the data in the master thread common block to the thread private copies of the common block. The copy occurs at the beginning of the parallel region. The COPYIN clause applies only to common blocks that have been declared THREADPRIVATE (see Section 6.1.4).

You do not have to specify a whole common block to be copied in; you can specify named variables that appear in the THREADPRIVATE common block. In the following example, the common blocks BLK1 and FIELDS are specified as thread private, but only one of the variables in common block FIELDS is specified to be copied in:

```
      COMMON /BLK1/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(/BLK1/,ZFIELD)
```

DEFAULT Clause

Use the DEFAULT clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to specify a default data scope attribute for all variables within the lexical extent of a parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause. You can specify only one DEFAULT clause on a directive. The default data scope attribute can be one of the following:

- PRIVATE

  Makes all named objects in the lexical extent of the parallel region private to a thread. The objects include common block variables, but exclude THREADPRIVATE variables.

- SHARED

  Makes all named objects in the lexical extent of the parallel region shared among all the threads in the team.

- NONE

  Declares that there is no implicit default as to whether variables are PRIVATE or SHARED. You must explicitly specify the scope attribute for each variable in the lexical extent of the parallel region.

If you do not specify the DEFAULT clause, the default is DEFAULT(SHARED). However, loop control variables are always PRIVATE by default.

You can exempt variables from the default data scope attribute by using other scope attribute clauses on the parallel region as shown in the following example:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED(R) LASTPRIVATE(I)
```

PRIVATE Clause

Use the PRIVATE clause on the PARALLEL, DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to declare variables to be private to each thread in the team.

The behavior of variables declared PRIVATE is as follows:

- A new object of the same type and size is declared once for each thread in the team, and the new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.

- Variables defined as PRIVATE are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent, but inside the dynamic extent, of the construct unless they are passed as actual arguments to called routines.

In the following example, the values of I and J are undefined on exit from the parallel region:

```
      INTEGER I,J
      I =1
      J =2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
      I =3
      J =J+ 2
!$OMP END PARALLEL
      PRINT *, I, J
```

FIRSTPRIVATE Clause

Use the FIRSTPRIVATE clause on the PARALLEL, DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

In addition to the PRIVATE clause functionality, private copies of the variables are initialized from the original object existing before the parallel construct.

LASTPRIVATE Clause

Use the LASTPRIVATE clause on the DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

When the LASTPRIVATE clause appears on a DO or PARALLEL DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct.

When the LASTPRIVATE clause appears on a SECTIONS or PARALLEL SECTIONS directive, the thread that executes the lexically last section updates the version of the object it had before the construct.

Subobjects that are not assigned a value by the last iteration of the DO loop or the lexically last SECTION directive are undefined after the construct.

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. You must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially. As shown in the following example, the value of I at the end of the parallel region is equal to N+1, as it would be with sequential execution.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N
         A(I) = B(I) + C(I)
      END DO
!$OMP END PARALLEL
      CALL REVERSE(I)
```

REDUCTION Clause

Use the REDUCTION clause on the PARALLEL, DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to perform a reduction on the specified variables by using an operator or intrinsic as shown:

REDUCTION ({ *operator* / *intrinsic* } :list)

Operator can be one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV..

Intrinsic can be one of the following: MAX, MIN, IAND, IOR, or IEOR.

The specified variables must be named scalar variables of intrinsic type and must be SHARED in the enclosing context. A private copy of each specified variable is created for each thread as if you had used the PRIVATE clause. The private copy is initialized to a value that depends on the operator or intrinsic as shown in Table 6–2. The actual initialization value will be consistent with the data type of the reduction variable.

Table 6–2 Operators/Intrinsics and Initialization Values for Reduction Variables

Operator/Intrinsic	Initialization Value
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

At the end of the construct to which the reduction applies, the shared variable is updated to reflect the result of combining the original value of the SHARED reduction variable with the final value of each of the private copies using the specified operator.

Except for subtraction, all of the reduction operators are associative and the compiler can freely reassociate the computation of the final value. The partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct. However, if you use the REDUCTION clause on a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all of the threads have completed the REDUCTION clause.

The REDUCTION clause is intended to be used on a region or worksharing construct in which the reduction variable is used only in reduction statements having one of the following forms:

```
x = x operator expr

x = expr operator x (except for subtraction)

x = intrinsic (x,expr)

x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a MAX reduction might be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator specified in the REDUCTION clause matches the reduction operation.

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a REDUCTION clause for that directive as shown in the following example:

```
!$OMP DO REDUCTION(+: A, Y),REDUCTION(.OR.: AM)
```

The following example shows how to use the REDUCTION clause:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(A,B),REDUCTION(+: A,B)
      DO I=1,N
         CALL WORK(ALOCAL,BLOCAL)
         A = A + ALOCAL
         B = B + BLOCAL
      END DO
!$OMP END PARALLEL DO
```

SHARED Clause

Use the SHARED clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to make variables shared among all the threads in a team.

In the following example, the variables *X* and *NPOINTS* are shared among all the threads in the team:

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

6.1.6 Parallel Region: PARALLEL and END PARALLEL Directives

Note

For overview information, see Section 6.1.3, Parallel Processing Thread Model.

The PARALLEL and END PARALLEL directives define a parallel region as follows:

```
!$OMP PARALLEL
  !parallel region
!$OMP END PARALLEL
```

When a thread encounters a parallel region, it creates a team of threads and becomes the master of the team. You can control the number of threads in a team by the use of an environment variable or a run-time library call, or both.

For More Information:

- See Section 6.4, Environment Variables for Adjusting the Run-Time Environment.
- See Appendix D, Parallel Library Routines.

The PARALLEL directive takes an optional comma-separated list of clauses that specifies:

- Whether the statements in the parallel region are executed in parallel by a team of threads or serially by a single thread (IF clause)
- Whether variables are PRIVATE, FIRSTPRIVATE, SHARED, or REDUCTION
- Whether variables have a DEFAULT data scope attribute
- Whether master thread common block values are copied to THREADPRIVATE copies of the common block (COPYIN clause)

Once created, the number of threads in the team remains constant for the duration of that parallel region. However, you can explicitly change the number of threads used in the next parallel region by calling the `OMP_SET_NUM_THREADS` run-time library routine from a serial portion of the program. This routine overrides any value you may have set using the `OMP_NUM_THREADS` environment variable.

Assuming you have used the OMP_NUM_THREADS environment variable to set the number of threads to 6, you can change the number of threads between parallel regions as follows:

```
      CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
      .
      .
      .
!$OMP END PARALLEL
      CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
      .
      .
      .
!$OMP END PARALLEL DO
```

Use the worksharing directives such as DO, SECTIONS, and SINGLE to divide the statements in the parallel region into units of work and to distribute those units so that each unit is executed by one thread.

In the following example, the !$OMP DO and !$OMP END DO directives and all the statements enclosed by them comprise the static extent of the parallel region:

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

In the following example, the !$OMP DO and !$OMP END DO directives and all the statements enclosed by them, including all statements contained in the WORK subroutine, comprise the dynamic extent of the parallel region:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
         CALL WORK(I,N)
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

When an IF clause is present on the PARALLEL directive, the enclosed code region is executed in parallel only if the scalar logical expression evaluates to TRUE. Otherwise, the parallel region is serialized. When there is no IF clause, the region is executed in parallel by default.

In the following example, the statements enclosed within the !$OMP DO and !$OMP END DO directives are executed in parallel only if there are more than three processors available. Otherwise the statements are executed serially:

```
!$OMP PARALLEL IF (OMP_GET_NUM_PROCS() .GT. 3)
!$OMP DO
    DO I=1,N
       Y(I) = SQRT(Z(I))
    END DO
!$OMP END DO
!$OMP END PARALLEL
```

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, **nested parallel regions** are always executed by a team of one thread.

To achieve better performance than sequential execution, a parallel region must contain one or more worksharing constructs so that the team of threads can execute work in parallel. It is the contained worksharing constructs that lead to the performance enhancements offered by parallel processing.

6.1.7 Worksharing Constructs

A worksharing construct must be enclosed dynamically within a parallel region if the worksharing directive is to execute in parallel. No new threads are launched and there is no implied barrier on entry to a worksharing construct.

The worksharing constructs are:

- DO and END DO directives (see Section 6.1.7.1)
- SECTIONS, SECTION, and END SECTIONS directives (see Section 6.1.7.2)
- SINGLE and END SINGLE directives (see Section 6.1.7.3)

6.1.7.1 DO and END DO directives

The DO directive specifies that the iterations of the immediately following DO loop must be dispatched across the team of threads so that each iteration is executed by a single thread. The loop that follows a DO directive cannot be a DO WHILE or a DO loop that does not have loop control. The iterations of the DO loop are dispatched among the existing team of threads.

You cannot use a GOTO statement, or any other statement, to transfer control into or out of the DO construct.

If you specify the optional END DO directive, it must appear immediately after the end of the DO loop. If you do not specify the END DO directive, an END DO directive is assumed at the end of the DO loop, and threads synchronize at that point.

The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

The clauses for the DO directive specify:

- Whether variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION
- How loop iterations are SCHEDULEd onto threads

In addition, the ORDERED clause must be specified if the ORDERED directive appears in the dynamic extent of the DO directive.

If you do not specify the optional NOWAIT clause on the END DO directive, threads synchronize at the END DO directive. If you specify NOWAIT, threads do not synchronize, and threads that finish early proceed directly to the instructions following the END DO directive.

The DO directive optionally lets you:

- Control data scope attributes (see Section 6.1.5, Controlling Data Scope Attributes).
- Use the SCHEDULE clause to specify schedule type and chunk size (see Section 6.1.10, Specifying Schedule Type and Chunk Size).

6.1.7.2 SECTIONS, SECTION, and END SECTIONS Directives

Use the noniterative worksharing SECTIONS directive to divide the enclosed sections of code among the team. Each section is executed just one time by one thread.

Each section should be preceded with a SECTION directive, except for the first section, in which the SECTION directive is optional. The SECTION directive must appear within the lexical extent of the SECTIONS and END SECTIONS directives.

The last section ends at the END SECTIONS directive. When a thread completes its section and there are no undispatched sections, it waits at the END SECTION directive unless you specify NOWAIT.

The SECTIONS directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

The following example shows how to use the SECTIONS and SECTION directives to execute subroutines XAXIS, YAXIS, and ZAXIS in parallel. The first SECTION directive is optional:

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

For More Information:

- See Section 6.1.5, Controlling Data Scope Attributes.

6.1.7.3 SINGLE and END SINGLE Directives

Use the SINGLE directive when you want just one thread of the team to execute the enclosed block of code.

Threads that are not executing the SINGLE directive wait at the END SINGLE directive unless you specify NOWAIT.

The SINGLE directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE or FIRSTPRIVATE. that specifies which variables are PRIVATE or FIRSTPRIVATE.

When the END SINGLE directive is encountered, an implicit barrier is erected and threads wait until all threads have finished. This can be overridden by using the NOWAIT option.

In the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT:

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

For More Information:

- See Section 6.1.5, Controlling Data Scope Attributes.

6.1.8 Combined Parallel/Worksharing Constructs

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- PARALLEL DO (see Section 6.1.8.1)
- PARALLEL SECTIONS (see Section 6.1.8.2)

6.1.8.1 PARALLEL DO and END PARALLEL DO Directives

Use the PARALLEL DO directive to specify a parallel region that implicitly contains a single DO directive.

You can specify one or more of the clauses for the PARALLEL and the DO directives.

The following example shows how to parallelize a simple loop. The loop iteration variable is private by default, so it is not necessary to declare it explicitly. The END PARALLEL DO directive is optional:

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

For More Information:

- See Section 6.1.6, Parallel Region: PARALLEL and END PARALLEL Directives.
- See Section 6.1.7.1, DO and END DO directives.

6.1.8.2 PARALLEL SECTIONS and END PARALLEL SECTIONS Directives

Use the PARALLEL SECTIONS directive to specify a parallel region that implicitly contains a single SECTIONS directive.

You can specify one or more of the clauses for the PARALLEL and the SECTIONS directives.

The last section ends at the END PARALLEL SECTIONS directive.

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently. The first SECTION directive is optional. Note that all SECTION directives must appear in the lexical extent of the PARALLEL SECTIONS/END PARALLEL SECTIONS construct:

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

For More Information:

- See Section 6.1.6, Parallel Region: PARALLEL and END PARALLEL Directives.
- See Section 6.1.7.2, SECTIONS, SECTION, and END SECTIONS Directives.

6.1.9 Synchronization Constructs

Synchronization constructs are used to assure the consistency of shared data and to coordinate parallel execution among threads.

The synchronization constructs are:

- ATOMIC directive (see Section 6.1.9.1)
- BARRIER directive (see Section 6.1.9.2)
- CRITICAL directive (see Section 6.1.9.3)
- FLUSH directive (see Section 6.1.9.4)
- MASTER directive (see Section 6.1.9.5)
- ORDERED directive (see Section 6.1.9.6)

6.1.9.1 ATOMIC Directive

Use the ATOMIC directive to ensure that a specific memory location is updated atomically instead of exposing the location to the possibility of multiple, simultaneously writing threads.

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

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type
- *expr* is a scalar expression that does not reference *x*
- *intrinsic* is either MAX, MIN, IAND, IOR, or IEOR
- *operator* is either +, *, -, /, .AND., .OR., .EQV., or .NEQV.

This directive permits optimization beyond that of a critical section around the assignment. An implementation can replace all ATOMIC directives by enclosing the statement in a critical section. All of these critical sections must use the same unique name.

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

This restriction applies to the ATOMIC directive: All references to storage location *x* must have the same type and type parameters.

In the following example, the collection of Y locations is updated atomically:

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

6.1.9.2 BARRIER Directive

To synchronize all threads within a parallel region, use the BARRIER directive. You can use this directive only within a parallel region defined by using the PARALLEL directive. You cannot use the BARRIER directive within the DO, PARALLEL DO, SECTIONS, PARALLEL SECTIONS, and SINGLE directives.

When encountered, each thread waits at the BARRIER directive until all threads have reached the directive.

In the following example, the BARRIER directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

```
c$OMP PARALLEL
c$OMP DO PRIVATE(i)
        DO i = 1, 100
           b(i) = i
        END DO
c$OMP BARRIER
c$OMP DO PRIVATE(i)
        DO i = 1, 100
           a(i) = b(101-i)
        END DO
c$OMP END PARALLEL
```

6.1.9.3 CRITICAL and END CRITICAL Directives

Use the CRITICAL and END CRITICAL directives to restrict access to a block of code, referred to as a **critical section**, to one thread at a time.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name.

When a thread enters the critical section, a **latch variable** is set to closed and all other threads are locked out. When the thread exits the critical section at the END CRITICAL directive, the latch variable is set to open, allowing another thread access to the critical section.

If you specify a critical section name in the CRITICAL directive, you must specify the same name in the END CRITICAL directive. If you do not specify a name for the CRITICAL directive, you cannot specify a name for the END CRITICAL directive.

All unnamed CRITICAL directives map to the same name. Critical section names are global to the program.

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

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

Unnamed critical sections use the global lock from the Pthread package. This allows you to synchronize with other code by using the same lock. Named locks are created and maintained by the compiler and can be significantly more efficient.

6.1.9.4 FLUSH Directive

Use the FLUSH directive to identify a synchronization point at which a consistent view of memory is provided. Thread-visible variables are written back to memory at this point.

To avoid flushing all thread-visible variables at this point, include a list of comma-separated named variables to be flushed.

The following example uses the FLUSH directive for point-to-point synchronization between thread 0 and thread 1 for the variable *ISYNC*:

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK()
! I Am Done With My Work, Synchronize With My Neighbor
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
! Wait Till Neighbor Is Done
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
      END DO
!$OMP END PARALLEL
```

6.1.9.5 MASTER and END MASTER Directives

Use the MASTER and END MASTER directives to identify a block of code that is executed only by the master thread.

The other threads of the team skip the code and continue execution. There is no implied barrier at the END MASTER directive.

In the following example, only the master thread executes the routines OUTPUT and INPUT:

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

6.1.9.6 ORDERED and END ORDERED Directives

Use the ORDERED and END ORDERED directives within a DO construct to allow work within an ordered section to execute sequentially while allowing work outside the section to execute in parallel.

When you use the ORDERED directive, you must also specify the ORDERED clause on the DO directive.

Only one thread at a time is allowed to enter the ordered section, and then only in the order of loop iterations.

In the following example, the code prints out the indexes in sequential order:

```
!$OMP DO ORDERED,SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
         CALL WORK(I)
      END DO
      SUBROUTINE WORK(K)
!$OMP ORDERED
      WRITE(*,*) K
!$OMP END ORDERED
```

6.1.10 Specifying Schedule Type and Chunk Size

The SCHEDULE clause of the DO or PARALLEL DO directive specifies a scheduling algorithm that determines how iterations of the DO loop are divided among and dispatched to the threads of the team. The SCHEDULE clause applies only to the current DO or PARALLEL DO directive.

Within the SCHEDULE clause, you must specify a **schedule type** and, optionally, a **chunk size**. A **chunk** is a contiguous group of iterations dispatched to a thread. Chunk size must be a scalar integer expression.

The following list describes the schedule types and how the chunk size affects scheduling:

- STATIC

  The iterations are divided into pieces having a size specified by chunk. The pieces are statically dispatched to threads in the team in a round-robin manner in the order of thread number.

  When chunk is not specified, the iterations are first divided into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins.

- DYNAMIC

The iterations are divided into pieces having a size specified by chunk. As each thread finishes its currently dispatched piece of the iteration space, the next piece is dynamically dispatched to the thread.

When no chunk is specified, the default is 1.

- GUIDED

  The chunk size is decreased exponentially with each succeeding dispatch. Chunk specifies the minimum number of iterations to dispatch each time. If there are less than chunk number of iterations remaining, the rest are dispatched.

  When no chunk is specified, the default is 1.

- RUNTIME

  The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by using the `OMP_SCHEDULE` environment variable (see Table 6–4).

  When you specify RUNTIME, you cannot specify a chunk size.

The following list shows which schedule type is used, in priority order:

1. The schedule type specified in the SCHEDULE clause of the current DO or PARALLEL DO directive
2. If the schedule type for the current DO or PARALLEL DO directive is RUNTIME, the default value specified in the `OMP_SCHEDULE` environment variable
3. The compiler default schedule type of STATIC

The following list shows which chunk size is used, in priority order:

1. The chunk size specified in the SCHEDULE clause of the current DO or PARALLEL DO directive
2. For RUNTIME schedule type, the value specified in the `OMP_SCHEDULE` environment variable
3. For DYNAMIC and GUIDED schedule types, the default value 1
4. If the schedule type for the current DO or PARALLEL DO directive is STATIC, the loop iteration space divided by the number of threads in the team

6.2 Compaq Fortran Parallel Compiler Directives

These directives are provided for compatibility with older programs that were written for parallel execution.

These topics are described:

- Command-line option and directives format (see Section 6.2.1)
- Directive summary descriptions (see Section 6.2.2)
- Parallel processing thread model (see Section 6.2.3)
- Privatizing named common blocks (see Section 6.2.4)
- Controlling data scope attributes (see Section 6.2.5)
- Parallel region (see Section 6.2.6)
- Worksharing constructs (see Section 6.2.7)
- Combined parallel/worksharing constructs (see Section 6.2.8)
- Synchronization constructs (see Section 6.2.9)
- Specifying a default chunk size (see Section 6.2.10)
- Specifying a default schedule type (see Section 6.2.11)
- Terminating loop execution early (see Section 6.2.12)

6.2.1 Command-Line Option and Directives Format

To use Compaq Fortran parallel compiler directives in your program, you must include the `-mp` compiler option on your `f90` command:

```
% f90 -mp prog.f -o prog
```

The format of a Compaq Fortran parallel compiler directive is:

```
prefix directive_name [option[[,] option]...]
```

All Compaq Fortran parallel compiler directives must begin with a directive prefix. Directives are not case-sensitive. Options can appear in any order after the directive name and can be repeated as needed, subject to the restrictions of individual options.

Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Trailing comments are allowed.

6.2.1.1 Directive Prefixes

The directive prefix you use depends on the source form you use in your program:

- Use the !$PAR prefix when compiling either fixed source form or free source form programs.
- Use the C$PAR (or c$PAR) and the *$PAR prefixes only when compiling fixed source form programs.

Fixed Source Form

For fixed source form programs, the prefix is one of the following:

```
!$PAR
C$PAR (or c$PAR)
*$PAR
```

For four directives, there is another form for fixed source form programs. This nonpreferred form is accepted by the compiler for compatibility reasons. The four directives are: CHUNK, COPYIN, DOACROSS, and MP_SCHEDULE. The prefix is c$. Thus, these four directives are acceptable:

Preferred Directive Name	Acceptable Directive Name
!$PAR CHUNK	c$CHUNK
!$PAR COPYIN	c$COPYIN
!$PAR DOACROSS	c$DOACROSS
!$PAR MP_SCHEDULE	c$MP_SCHEDULE

For More Information:

- See Section 6.1.1.1, Directive Prefixes.

Free Source Form

For free source form programs, the prefix is !$PAR.

For More Information:

- See Section 6.1.1.1, Directive Prefixes.

6.2.2 Summary Descriptions of Compaq Fortran Parallel Compiler Directives

Table 6–3 provides summary descriptions of the Compaq Fortran parallel compiler directives. For complete information about the Compaq Fortran parallel compiler directives, see the *Compaq Fortran Language Reference Manual*.

Table 6–3 Compaq Fortran Parallel Compiler Directives

Directive Format	Description
prefix BARRIER	This directive defines a synchronization construct that synchronizes all the threads in a team. See Section 6.2.9.1, BARRIER Directive.
prefix CHUNK = chunksize	This directive sets a default chunk size used to divide iterations among the threads of the team. See Section 6.2.10, Specifying a Default Chunk Size.
prefix COPYIN object[, object] . . .	This data environment directive specifies that the listed variables, single array elements, and common blocks be copied from the master thread to the PRIVATE data objects having the same name. Single array elements can be copied, but array sections cannot be copied. Shared variables cannot be copied. When an allocatable array is to be copied, it must be allocated when the COPYIN directive is encountered. This directive is allowed only within PARALLEL and PARALLEL DO directives.
prefix CRITICAL SECTION [(latch-var)] *code* **prefix END CRITICAL SECTION**	

(continued on next page)

Table 6–3 (Cont.) Compaq Fortran Parallel Compiler Directives

Directive Format	Description
	These directives define a synchronization construct that specifies a block of code that is executed by one thread at a time. See Section 6.2.9.2, CRITICAL SECTION and END CRITICAL SECTION Directives.
prefix INSTANCE { SINGLE | PARALLEL } /com-blk-name/[[,]/com-blk-name/] . . .	This data environment directive makes named common blocks available to threads. See Section 6.2.4, Privatizing Named Common Blocks: TASKCOMMON or INSTANCE Directives.
prefix MP_SCHEDTYPE = mode	This directive sets a default run-time schedule type. See Section 6.2.11, Specifying a Default Schedule Type.
prefix PARALLEL [region-option[[,]region-option] . . .] *code* **prefix END PARALLEL**	These directives define a parallel construct that is a region of a program that must be executed by a team of threads in parallel until the END PARALLEL directive is encountered. See Section 6.2.6, Parallel Region: PARALLEL and END PARALLEL Directives .
prefix { PARALLEL DO | DOACROSS } [par-do-option[[,]par-do-option] . . .] *do_loop* **[prefix END PARALLEL DO]**	

(continued on next page)

Table 6–3 (Cont.) Compaq Fortran Parallel Compiler Directives

Directive Format	Description
	These directives define a combined parallel/worksharing construct that specifies an abbreviated form of specifying a parallel region that contains a single PDO directive. See Section 6.2.8.1, PARALLEL DO and END PARALLEL DO Directives.
prefix PARALLEL SECTIONS [par-sect-option[[,]par-sect-option] . . .] *code* **prefix END PARALLEL SECTIONS**	These directives define a combined parallel/worksharing construct that specifies an abbreviated form of specifying a parallel region that contains a single SECTION directive. The semantics are identical to explicitly specifying the PARALLEL directive immediately followed by a PSECTIONS directive. See Section 6.2.8.2, PARALLEL SECTIONS and END PARALLEL SECTIONS Directives.
prefix PDO [pdo-option[[,]pdo-option] . . .] *do_loop* **[prefix END PDO [NOWAIT]]**	These directives define a worksharing construct that specifies that each set of iterations of the contained DO LOOP is a unit of work that can be scheduled on a single thread. See Section 6.2.7.1, PDO and END PDO Directives.
prefix PDONE	This directive specifies that the DO loop in which the PDONE directive is contained should be terminated early. See Section 6.2.12, Terminating Loop Execution Early: PDONE Directive.

(continued on next page)

Table 6–3 (Cont.) Compaq Fortran Parallel Compiler Directives

Directive Format	Description
prefix PSECTION[S] [sect-option[[,]sect-option] . . .] **[prefix SECTION]** *code* **[prefix SECTION** *code*] **prefix END PSECTION[S] [NOWAIT]**	These directives define a worksharing construct that specifies that the enclosed sections of code are to be divided among threads in the team. See Section 6.2.7.2, PSECTIONS, SECTION, and END PSECTIONS Directives.
prefix SINGLE PROCESS [proc-option[[,]proc-option] . . .] *code* **prefix END SINGLE PROCESS [NOWAIT]**	These directives define a worksharing construct that specifies a block of code that is executed by only one thread. See Section 6.2.7.3, SINGLE PROCESS and END SINGLE PROCESS Directives.
prefix TASKCOMMON com-blk-name[,com-blk-name] . . .	This data environment directive makes named common blocks private to a thread, but global within the thread. See Section 6.2.4, Privatizing Named Common Blocks: TASKCOMMON or INSTANCE Directives.

6.2.3 Parallel Processing Thread Model

The concepts of the parallel processing thread model are the same as those for OpenMP Fortran API with one exception: orphaned directives are not possible with Compaq Fortran parallel compiler directives.

For More Information:

- See Section 6.1.3, Parallel Processing Thread Model.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment options on directives, you can:

- Privatize named common blocks (see Section 6.2.4)
- Control data scope attributes (see Section 6.2.5)

6.2.4 Privatizing Named Common Blocks: TASKCOMMON or INSTANCE Directives

You can make named common blocks private to a thread, but global within the thread by using the TASKCOMMON or the INSTANCE PARALLEL directive:

- For TASKCOMMON, specify a comma-separated list of common block names
- For INSTANCE PARALLEL, specify a comma-separated list of common block names, each enclosed by slashes (/*name*/)

The TASKCOMMON and INSTANCE PARALLEL directives are semantically equivalent and differ only in form.

Only named common blocks can be made thread private.

Each thread gets its own copy of the common block, with the result that data written to the common block by one thread is not directly visible to other threads. During serial portions of the program, accesses are to the master thread copy of the common block.

You should assume that the data in thread private common blocks is undefined on entry into the first parallel region unless you specified the COPYIN option in the PARALLEL directive (see COPYIN Option in Section 6.2.5).

When you make thread private a common block that is initialized using DATA statements, the copy of the common block for each thread has that initial value. If no initial value is provided, the variables in the common block are assigned the value of zero.

You can also specify INSTANCE SINGLE, which is the default in the absence of any attribute for the directive. In this case, all threads share the same copy of the common block in the master thread. Assignments made by one thread affect the copy in all other threads.

When you specify INSTANCE PARALLEL, the named common blocks are made private to a thread, but global within the thread.

The TASKCOMMON directive is the same as the OpenMP Fortran API THREADPRIVATE directive except that slashes (//) do not have to be used to delimit named common blocks.

For More Information:

- See Section 6.1.4, Privatizing Named Common Blocks: THREADPRIVATE Directive.

6.2.5 Controlling Data Scope Attributes

You can use several options to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute option on a directive, the default is SHARED for those variables affected by the directive.

Each of the data scope attribute options accepts a *list*, which is a comma-separated list of named variables or named common blocks that are accessible in the scoping unit. When you specify named common blocks, they must appear between slashes (/*name*/).

Not all of the options are allowed on all directives, but the directives to which each option applies are listed in the clause descriptions.

The data scope attribute options are:

- COPYIN
- DEFAULT
- FIRSTPRIVATE
- LASTLOCAL or LAST LOCAL
- PRIVATE or LOCAL
- REDUCTION
- SHARED or SHARE

COPYIN Option

Use the COPYIN option on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to copy named common block values from the master thread copy to threads at the beginning of a parallel region, use the COPYIN option on the PARALLEL directive. The COPYIN option applies only to named common blocks that have been previously declared thread private using the TASKCOMMON or the INSTANCE PARALLEL directive (see Section 6.2.4).

Use a comma-separated list to name the common blocks and variables in common blocks you want to copy.

DEFAULT Option

This option is the same as the OpenMP Fortran API DEFAULT clause (see Section 6.1.5).

FIRSTPRIVATE Option

This option is the same as the OpenMP Fortran API FIRSTPRIVATE clause (see Section 6.1.5).

LASTLOCAL or LAST LOCAL Option

Except for differences in directive name spelling, the LASTLOCAL or LAST LOCAL option is the same as the OpenMP Fortran API LASTPRIVATE clause (see Section 6.1.5).

PRIVATE or LOCAL Option

Except for the alternate directive spelling of LOCAL, the PRIVATE or LOCAL option is the same as the OpenMP Fortran API PRIVATE clause (see Section 6.1.5).

REDUCTION Option

Use the REDUCTION option on the PDO directive to declare variables that are to be the object of a reduction operation. Use a comma-separated list to name the variables you want to declare as objects of a reduction.

The REDUCTION option in the Compaq Fortran parallel compiler directive set is different from the REDUCTION clause in the OpenMP Fortran API directive set. In the OpenMP Fortran API directive set, both a variable and an operator type are given. In the Compaq Fortran parallel compiler directive set, the operator is not given. The compiler must be able to determine the reduction operation from the source code. The REDUCTION option can be applied to a variable in a DO loop only if the variable meets the following criteria:

- It must be scalar.
- It must be assigned to exactly once in the DO loop.
- It must be read from exactly once in the DO loop and only in the right side of the assignment.
- The assignment must be one of the following forms:

  `x = x` *`operator`* `expr`

  `x = expr` *`operator`* `x` (except for subtraction)

  `operator``(x, expr)`

  `operator``(expr, x)`

where *operator* is one of the following supported reduction operations: +, -, *, .AND., .OR., .EQV., .NEQV., MAX, MIN, IAND, or IOR.

The compiler rewrites the reduction operation by computing partial results into local variables and then combining the results into the reduction variable. The reduction variable must be SHARED in the enclosing context.

SHARED or SHARE Option

Except for the alternate directive spelling of SHARE, the SHARED or SHARE option is the same as the OpenMP Fortran API SHARED clause (see Section 6.1.5).

6.2.6 Parallel Region: PARALLEL and END PARALLEL Directives

The concepts of using a parallel region are the same as those for OpenMP Fortran API (see Section 6.1.6), with these differences:

- Use the worksharing directives such as DO, SECTIONS, and SINGLE to divide the statements in the parallel region into units of work and to distribute those units so that each unit is executed by one thread.
- The environment variable you use to set the default number of threads is `MP_THREAD_COUNT` and the run-time library routine is `OtsSetNumThreads`.

For More Information:

- See Section 6.4, Environment Variables for Adjusting the Run-Time Environment.
- See Appendix D, Parallel Library Routines.

6.2.7 Worksharing Constructs

A worksharing construct must be enclosed lexically (not dynamically, as with OpenMP Fortran API directives) within a parallel region if the worksharing directive is to execute in parallel. No new threads are launched and there is no implied barrier on entry to a worksharing construct.

The worksharing constructs are:

- PDO and END PDO directives (see Section 6.2.7.1)
- PSECTIONS, SECTION, and END PSECTIONS directives (see Section 6.2.7.2)
- SINGLE PROCESS and END SINGLE PROCESS directives (see Section 6.2.7.3)

6.2.7.1 PDO and END PDO Directives

The PDO directive specifies that the iterations of the immediately following DO loop must be dispatched across the team of threads so that each iteration is executed in parallel by a single thread.

The loop that follows a PDO directive cannot be a DO WHILE or a DO loop that does not have loop control. The iterations of the DO loop are divided among and dispatched to the existing threads in the team.

You cannot use a GOTO statement, or any other statement, to transfer control into or out of the PDO construct.

This directive must be nested within the lexical extent of a PARALLEL directive. The PARALLEL directive takes an optional comma-separated list of options that specifies:

- Whether variables are PRIVATE, FIRSTPRIVATE, LASTLOCAL, or REDUCTION
- How iterations are scheduled onto threads and whether this is deferred until run time (MP_SCHEDTYPE option)
- How many iterations each thread is assigned (CHUNK or BLOCKED option)
- Whether iterations are in an ordered sequence (ORDERED option)

If you specify the optional END PDO directive, it must appear immediately after the end of the DO loop. If you do not specify the END PDO directive, an END PDO directive is assumed at the end of the DO loop.

If you do not specify the optional NOWAIT clause on the END PDO directive, threads synchronize at the END PDO directive. If you specify NOWAIT, threads do not synchronize at the END PDO directive. Threads that finish early proceed directly to the instructions following the END PDO directive.

You can use the ORDERED option to affect the way threads are dispatched. When you specify this option, iterations are dispatched to threads in the same order they would be for sequential execution.

The PDO directive optionally lets you:

- Control data scope attributes (see Section 6.1.5)
- Specify chunk size (see Section 6.2.10)
- Specify schedule type (see Section 6.2.11)
- Terminate loop execution early (see Section 6.2.12)

- Override implicit synchronization

Overriding Implicit Synchronization

Whether or not you include the END PDO directive at the end of the DO loop, by default an implicit synchronization point exists immediately after the last statement in the loop. Threads reaching this point wait until all threads complete their work and reach this synchronization point.

If there are no data dependences between the variables inside the loop and those outside the loop, there may be no reason to make threads wait. In this case, use the NOWAIT clause on the END PDO directive to override synchronization and allow threads to continue.

6.2.7.2 PSECTIONS, SECTION, and END PSECTIONS Directives

This directive is the same as the OpenMP Fortran API SECTIONS directive with the following exceptions:

- The names are PSECTIONS and END PSECTIONS.
- No REDUCTION clause or LASTPRIVATE clause is permitted.
- LOCAL is permitted as an alternative spelling for the PRIVATE clause.

For More Information:

- See Section 6.1.7.2, SECTIONS, SECTION, and END SECTIONS Directives.

6.2.7.3 SINGLE PROCESS and END SINGLE PROCESS Directives

This directive is the same as the OpenMP Fortran API SINGLE directive with the following exceptions:

- The names are SINGLE PROCESS and END SINGLE PROCESS.
- LOCAL is permitted as an alternative spelling for the PRIVATE clause.

For More Information:

- See Section 6.1.7.3, SINGLE and END SINGLE Directives.

6.2.8 Combined Parallel/Worksharing Constructs

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- PARALLEL DO (see Section 6.2.8.1)
- PARALLEL SECTIONS (see Section 6.2.8.2)

6.2.8.1 PARALLEL DO and END PARALLEL DO Directives

This directive is the same as the OpenMP Fortran API PARALLEL DO directive with the following exceptions:

- You can use the alternate DOACROSS directive name instead of PARALLEL DO. For compatibility, the following form is also allowed (for fixed source form only): c$DOACROSS.
- The options can be one or more of the options for the PARALLEL and PDO directives.

The PARALLEL DO directive optionally lets you:

- Control data scope attributes (see Section 6.1.5)
- Specify chunk size (see Section 6.2.10)
- Specify schedule type (see Section 6.2.11)
- Terminate loop execution early (see Section 6.2.12)

For More Information:

- See Section 6.1.8.1, PARALLEL DO and END PARALLEL DO Directives.

6.2.8.2 PARALLEL SECTIONS and END PARALLEL SECTIONS Directives

This directive is the same as the OpenMP Fortran API PARALLEL SECTIONS directive with the following exception: The options can be one or more of the options for the PARALLEL and PSECTIONS directives, instead of the PARALLEL and SECTIONS directives.

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

For More Information:

- See Section 6.1.8.2, PARALLEL SECTIONS and END PARALLEL SECTIONS Directives.

6.2.9 Synchronization Constructs

Synchronization constructs are used to assure the consistency of shared data and to coordinate parallel execution among threads.

The synchronization constructs are:

- BARRIER directive (see Section 6.2.9.1)
- CRITICAL SECTION directive (see Section 6.2.9.2)

6.2.9.1 BARRIER Directive

The BARRIER directive is the same as the OpenMP Fortran API BARRIER directive (see Section 6.1.9.2).

6.2.9.2 CRITICAL SECTION and END CRITICAL SECTION Directives

The CRITICAL SECTION and END CRITICAL SECTION directives are the same as the OpenMP Fortran API CRITICAL and END CRITICAL directives with the following exceptions:

- The directive names are CRITICAL SECTION and END CRITICAL SECTION.
- You can specify an optional latch variable name.
- If you do not specify a latch variable name, the compiler assigns a unique name.
- The END CRITICAL SECTION directive does not take a latch variable name.
- You must explicitly initialize a latch variable to zero before any critical section using that latch variable is executed.
- You must not reuse that latch variable in anything other than a critical section until all uses as a latch variable are complete.

For More Information:

- See Section 6.1.9.3, CRITICAL and END CRITICAL Directives.

6.2.10 Specifying a Default Chunk Size

A chunk is a contiguous group of iterations dispatched to a thread. You can explicitly define a chunk size for a PDO or PARALLEL DO directive by using the CHUNK or BLOCKED option. Chunk size must be a scalar integer expression. The specified chunk size applies only to the current PDO or PARALLEL DO directive.

The following list shows which chunk size is used, in priority order:

1. The chunk size specified in the CHUNK or BLOCKED option of the current PDO or PARALLEL DO directive.
2. The value specified in the most recent CHUNK directive. (The CHUNK directive is provided for compatibility reasons.)
3. If the schedule type for the current PDO or PARALLEL DO directive is either INTERLEAVED, DYNAMIC, GUIDED, or RUNTIME, the chunk size default value specified in the `MP_CHUNK_SIZE` environment variable.

4. The compiler default chunk size value of 1.

The interaction between the chunk size and the schedule type are:

- For the DYNAMIC and INTERLEAVED schedule types, iterations are always dispatched to threads in chunk size groups. If the total number of iterations is not evenly divisible by chunk size, the last group dispatched has fewer iterations.
- For the GUIDED schedule type, chunk size is the minimum number of iterations that can be dispatched to a thread. If less than chunk size iterations remain, the remaining iterations are dispatched to the next available thread.
- For the STATIC schedule type, chunk size is ignored.

6.2.11 Specifying a Default Schedule Type

The schedule type specifies a scheduling algorithm that determines how chunks of loop iterations are dispatched to the threads of a team. The schedule type does not affect the semantics of the program, but might affect performance. You can explicitly define a run-time schedule type for the current PDO or PARALLEL DO directive by using the MP_SCHEDTYPE option. The specified schedule type applies to the current PDO or PARALLEL DO directive only.

The following list shows which schedule type is used, in priority order:

1. The schedule type specified in the MP_SCHEDTYPE option of the current PDO or PARALLEL DO directive.
2. The schedule type specified in the most recent MP_SCHEDTYPE directive. (The MP_SCHEDTYPE directive is provided for compatibility reasons.)
3. If the schedule type for the current PDO or PARALLEL DO directive is RUNTIME, the default value specified in the `MP_SCHEDTYPE` environment variable.
4. The compiler default schedule type of STATIC.

The following list describes the schedule types and how the chunk size affects scheduling:

- For the STATIC or SIMPLE schedule types, one contiguous group of iterations is dispatched to each thread, with each group having approximately the same number of iterations.
- For the INTERLEAVED or INTERLEAVE schedule types, a chunk-sized group of iterations is dispatched to each thread in a round-robin manner.

- For the DYNAMIC schedule type, a chunk-sized group of the remaining iterations is dispatched to the next available thread. If less than one chunk size of iterations remain, all the remaining iterations are dispatched.
- For the GUIDED or GSS schedule types (similar to the DYNAMIC schedule type), the number of iterations dispatched is relatively large at the beginning of the loop and decreases exponentially. The number of iterations dispatched is not necessarily evenly divisible by chunk size.

  The specified chunk size is the minimum number of iterations that can be dispatched when a thread becomes available. When the number of remaining iterations is less than or equal to chunk size, all of the remaining iterations are dispatched to the next available thread.

  In some cases, setting a chunk size greater than 1 improves execution efficiency as the loop nears termination. This is because contention between threads for the small number of remaining iterations is reduced.
- For the RUNTIME schedule type, the schedule type and the chunk size are those specified in the `MP_SCHEDTYPE` environment variable.

The DYNAMIC and GUIDED schedule types introduce some amount of overhead required to manage the continuing dispatching of iterations to threads. However, this overhead is sometimes offset by better load balancing when the average execution time of iterations is not uniform throughout the loop.

The STATIC and INTERLEAVED schedule types dispatch all of the iterations to the threads in advance, with each thread receiving approximately equal numbers of iterations. One of these types is usually the most efficient schedule type when the average execution time of iterations is uniform throughout the loop.

6.2.12 Terminating Loop Execution Early: PDONE Directive

If you want to terminate loop execution early because a specified condition has been satisfied, use the PDONE directive. This is an executable directive and any undispatched iterations are not executed. However, all previously dispatched iterations are completed.

This directive must be nested within the lexical extent of a PDO or PARALLEL DO directive.

When the schedule type is STATIC or INTERLEAVED, this directive has no effect because all loop iterations are dispatched before the DO loop executes.

6.3 Decomposing Loops for Parallel Processing

Note

This section contains information that applies to both the OpenMP Fortran API and the Compaq Fortran parallel compiler directives. The code examples use the OpenMP API directive format.

To run in parallel, the source code in iterated DO loops must be decomposed by the user, and adequate system resources must be made available. **Decomposition** is the process of analyzing code for data dependences, dividing up the workload, and ensuring correct results when iterations run concurrently.

The term **loop decomposition** is used to specify the process of dividing the iterations of an iterated DO loop and running them on two or more threads of a shared-memory multi-processor computer system.

The only type of decomposition available with Compaq Fortran is **directed decomposition** using a set of parallel compiler directives.

The following sections describe how to decompose loops and how to use the OpenMP Fortran API and the Compaq Fortran parallel compiler directives to achieve parallel processing.

6.3.1 Steps in Using Directed Decomposition

When a program is compiled using the `-omp` or the `-mp` option, the compiler parses the parallel compiler directives. However, you must transform the source code to resolve any loop-carried dependences and improve run-time performance. (Another method of supporting parallel processing does not involve iterated DO loops. Instead, it allows large amounts of independent code to be run in parallel using the SECTIONS and SECTION directives.)

To use directed decomposition effectively, take the following steps:

1. Identify the loops that benefit most from parallel processing:
   - Consider whether another algorithm might achieve more parallelism in general.
   - Evaluate any caller or called loops and decompose the most CPU-intensive loops in the application (as long as there are no interfering dependences).

If a parallel DO loop invokes a subprogram containing another parallel DO loop, only the parallel DO loop of the calling program will be run in parallel. Each of the threads executing the outermost parallel DO loop will execute all of the iterations in the innermost parallel DO loop in a serial, nonparallel fashion.

- Make sure the loop contains enough CPU work to outweigh the parallel-processing startup overhead.

2. Analyze the loop and resolve dependences as needed. (See Section 6.3.2, Resolving Dependences Manually.) If you cannot resolve loop-carried dependences, you cannot safely decompose the loop.
3. Make sure the shared or private attributes inside the loop are consistent with corresponding use outside the loop. By default, common blocks and individual variables are shared, except for the loop control variable and variables referenced in a subprogram called from within a parallel loop (in which case they are private by default).
4. Precede the loop with the PARALLEL directive followed by the DO directive. You can combine the two directives by using the PARALLEL DO directive.
5. As needed, manually optimize the loop.
6. Make sure the loop complies with restrictions of the parallel-processing environment.
7. Without using the `-omp` option or the `-mp` option, compile, test, and debug the program.
8. Using `-omp` (or `-mp`), repeat the previous step.
9. Evaluate the parallel run:
   - If you reach an acceptable level of performance and if the results are correct, stop.
   - If the results are inaccurate, analyze the manually decomposed loops for dependences, apply a method to resolve them, and retest the parallel run.
   - If performance is inadequate, consider adjusting the run-time environment (see Section 6.4) or performing other manual optimizations, or consider other alternatives discussed in this manual. Then reenter the cycle by retesting the parallel program.

6.3.2 Resolving Dependences Manually

In directed decomposition, you must resolve loop-carried dependences and dependences involving temporary variables to ensure safe parallel execution. Only cycles of dependences are nearly impossible to resolve.

Do one of the following:

- Let the loop execute serially (possibly decompose an outer loop level)
- Use a lock (CRITICAL) to force the critical section to execute serially
- Recode or restructure the loop
- Find another algorithm that does not have cycles of dependences

There are several methods for resolving dependences manually:

- For dependences on variables used as temporaries, declare them PRIVATE; this effectively makes separate copies of temporary values for each thread.
- Recode the loop so that the loop-carried dependence becomes loop independent, with each thread having the involved store and fetch operation contained in a single iteration.
- Insert locks (CRITICAL) around the critical section containing the dependence.

  Use this technique only for very CPU-intensive loops, when no other method is possible, and for the smallest amount of code possible. The locks extend processing time by making individual threads wait while only one executes the critical region at a time.
- Recode loops with cycles of dependences (these are typically linear recurrences).

6.3.2.1 Resolving Dependences Involving Temporary Variables

Declare temporary variables PRIVATE to resolve dependences involving them. Temporary variables are used in intermediate calculations. If they are used in more than one iteration of a parallel loop, the program can produce incorrect results.

One thread might define a value and another thread use that value instead of the one it defined for a particular iteration. Loop control variables are prime examples of temporary variables that are declared PRIVATE by default within a parallel region. For example:

```
DO I = 1,100
   TVAR = A(I) + 2
   D(I) = TVAR + Y(I-1)
END DO
```

As long as certain criteria are met, you can resolve this kind of dependence by declaring the temporary variable (TVAR, in the example) PRIVATE. That way, each thread keeps its own copy of the variable.

For the criteria to be met, the values of the temporary variable must be all of the following:

- Defined in each iteration, inside the loop
- Meant to be used inside the same iteration that established it
- Used nowhere outside the loop unless it is redefined outside the loop before subsequent use

The default for variables in a parallel loop is SHARED, so you must explicitly declare these variables PRIVATE to resolve this kind of dependence.

6.3.2.2 Resolving Loop-Carried Dependences

You can often resolve loop-carried dependences using one or more of the following loop transformations:

- Loop alignment
- Code replication
- Loop distribution
- Restructure the loop into an inner and outer loop

These techniques also resolve dependences that inhibit autodecomposition.

6.3.2.3 Loop Alignment

Loop alignment offsets memory references in the loop so that the dependence is no longer loop carried. The following example shows a loop that is aligned to resolve the dependence in array A:

Loop with Dependence	Aligned Statements

`DO I = 2,N`
`  A(I) = B(I)`
`  C(I) = A(I+1)`
`END DO`
	`C(I-1) = A(I)`
	`A(I) = B(I)`

To compensate for the alignment and achieve the same calculations as the original loop, you probably have to perform one or more of the following:

- Change the loop control variable.
- Add IF constructs.
- Switch the order of the statements (this preserves the relative store-fetch order of the original loop).

Example 6–1 shows two possible forms of the final loop.

Example 6–1 Aligned Loop

```
!     First possible form:
!$OMP PARALLEL PRIVATE (I)
!$OMP DO
      DO I = 2,N+1
         IF (I .GT. 2) C(I-1) = A(I)
         IF (I .LE. N) A(I) = B(I)
      END DO
!$OMP END DO
!$OMP END PARALLEL
!
!     Second possible form; more efficient because the tests are
!     performed outside the loop:
!
!$OMP PARALLEL
!$OMP DO
      DO I = 3,N
         C(I-1) = A(I)
         A(I) = B(I)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      IF (N .GE. 2)
         A(2) = B(2)
         C(N) = A(N+1)
      END IF
```

6.3.2.4 Code Replication

When a loop contains a loop-independent dependence as well as a loop-carried dependence, loop alignment alone is usually not adequate. By resolving the loop-carried dependence, you often misalign another dependence. **Code replication** creates temporary variables that duplicate operations and keep the loop-independent dependences inside each iteration.

In S_2 of the following loop, aligning the A(I-1) reference without code replication would misalign the A(I) reference:

Loop with Multiple Dependences	Misaligned Dependence
DO I = 2,100	
S_1 A(I) = B(I) + C(I)	D(I-1) = A(I-1) + A(I)
S_2 D(I) = A(I) + A(I-1)	A(I) = B(I) + C(I)
END DO	

Example 6–2 uses code replication to keep the loop-independent dependence inside each iteration. The temporary variable, TA, must be declared PRIVATE.

Example 6–2 Transformed Loop Using Code Replication

```
!$OMP PARALLEL PRIVATE (I,TA)
      A(2) = B(2) + C(2)
      D(2) = A(2) + A(1)
!$OMP DO
      DO I = 3,100
         A(I) = B(I) + C(I)
         TA = B(I-1) + C(I-1)
         D(I) = A(I) + TA
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

6.3.2.5 Loop Distribution

Loop distribution allows more parallelism when neither loop alignment nor code replication can resolve the dependences. Loop distribution divides the contents of loops into multiple loops so that dependences cross between two separate loops. The loops run serially in relation to each other, even if they both run in parallel.

The following loop contains multiple dependences that cannot be resolved by either loop alignment or code replication:

```
    DO I = 1,100
S1    A(I) = A(I-1) + B(I)
S2    C(I) = B(I) - A(I)
    END DO
```

Example 6–3 resolves the dependences by distributing the loop. S_2 can run in parallel despite the data recurrence in S_1.

Example 6–3 Distributed Loop

```
      DO I 1,100
S1     A(I) = A(I-1) + B(I)
      END DO

      DO I 1,100
S2     C(I) = B(I) - A(I)
      END DO
```

6.3.2.6 Restructuring a Loop into an Inner and Outer Nest

Restructuring a loop into an inner and outer loop nest can resolve some recurrences that are used as rapid approximations of a function of the loop control variable. For example, the following loop uses sines and cosines:

```
THETA = 2.*PI/N
DO I=0,N-1
  S = SIN(I*THETA)
  C = COS(I*THETA)
    .
    .          ! use S and C
    .
END DO
```

Using a recurrence to approximate the sines and cosines can make the serial loop run faster (with some loss of accuracy), but it prevents the loop from running in parallel:

```
THETA = 2.*PI/N
STH = SIN(THETA)
CTH = COS(THETA)
S = 0.0
C = 1.0
DO I=0,N-1
  .
  .           ! use S and C
  .
  S = S*CTH + C*STH
  C = C*CTH - S*STH
END DO
```

To resolve the dependences, substitute the SIN and COS calls. (However, this loses the performance improvement gained from using the recurrence.) You can also restructure the loop into an outer parallel loop and an inner serial loop. Each iteration of the outer loop reinitializes the recurrence, and the inner loop uses the value:

```
!$OMP PARALLEL SHARED (THETA,STH,CTH,LCHUNK) PRIVATE (ISTART,I,S,C)
      THETA = 2.*PI/N
      STH = SIN(THETA)
      CTH = COS(THETA)
      LCHUNK = (N + NWORKERS()-1) / NWORKERS
!$OMP DO
      DO ISTART = 0,N-1,LCHUNK
        S = SIN(ISTART*THETA)
        C = COS(ISTART*THETA)
        DO I = ISTART, MIN(N,ISTART+LCHUNK-1)
          .
          .         ! use S and C
          .
          S = S*CTH + C*STH
          C = C*CTH - S*STH
        END DO
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

6.3.2.7 Dependences Requiring Locks

When no other method can resolve a dependence, you can put locks around the critical section that contains them. Locks force threads to execute the critical section serially, while allowing the rest of the loop to run in parallel.

However, locks degrade performance because they force the critical section to run serially and increase the overhead. They are best used only when no other technique resolves the dependence, and only in CPU-intensive loops.

To create locks in a loop, enclose the critical section between the CRITICAL and END CRITICAL directives. When a thread executes the CRITICAL directive and the latch variable is open, it takes possession of the latch variable, and other threads must wait to execute the section. The latch variable becomes open when the thread executing the section executes the END CRITICAL directive.

The latch variable is closed when a thread has possession of it and open when the latch variable is free.

In Example 6–4, the statement updating the sum is locked for safe parallel execution of the loop.

Example 6–4 Decomposed Loop Using Locks

```
      INTEGER(4) LCK
!$OMP PARALLEL PRIVATE (I,Y) SHARED (LCK,SUM)
      LCK = 0
      .
      .
      .
!$OMP DO
      DO I = 1,1000
         Y = some_calculation
!$OMP CRITICAL (LCK)
         SUM = SUM + Y
!$OMP END CRITICAL (LCK)
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

This particular example is better solved using a REDUCTION clause as shown in Example 6–5.

Example 6–5 Decomposed Loop Using a REDUCTION Clause

```
      INTEGER(4) LCK
!$OMP PARALLEL PRIVATE (I,Y) SHARED (LCK,SUM)
      LCK = 0
      .
      .
      .
!$OMP DO REDUCTION (SUM)
      DO I = 1,1000
         Y = some_calculation
         SUM = SUM + Y
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

6.3.3 Coding Restrictions

Because iterations in a parallel DO loop execute in an indeterminate order and in different threads, certain constructs in these loops can cause unpredictable run-time behavior.

The following restrictions are flagged:

- The loop control variable for a parallel loop must be declared an integer.
- Only comment lines and blank lines can exist between a DO directive and the DO loop statement.

- The loop body must not contain any RETURN statements.
- A loop with a branch (GOTO) into or out of its body, or having an EXIT statement cannot be run in parallel.

The following restrictions are not flagged:

- Loop-carried dependences involving shared variables must not exist between iterations of a parallel loop.
- Dependences involving private variables must not exist between code within a parallel loop and code executed before entry into or after the completion of the loop.
- System services or run-time library routines that change the context of a thread (such as a change in privileges, priority, access mode, or environment variables) must not be called from within a parallel loop.
- I/O statements and the control statements PAUSE and STOP must not be used in a routine called at any call level from within a parallel loop.
- Private symbols must not be referenced in a SAVE statement in a routine called at any call level from within a parallel loop.
- If a dummy argument is referenced within a parallel DO loop, the corresponding actual argument must reside in shared memory.
- Random number generators must be used carefully inside parallel loops, because parallel processing affects how numbers are generated.

6.3.4 Manual Optimization

To manually optimize structures containing parallel loops:

- Interchange loops so that the parallel loop has the most CPU work and the caches can perform efficiently.
- Balance the parallel work among threads when it is unusually unbalanced.

6.3.4.1 Interchanging Loops

The following example shows a case in which an inner loop can run in parallel and an outer loop cannot, because of a loop-carried dependence. The inner loop also has a more effective memory-referencing pattern for parallel processing than the outer loop. By interchanging the loops, more work executes in parallel and the cache can perform more efficiently.

Original Structure	Interchanged Structure
	`!$OMP PARALLEL PRIVATE (J,I) SHARED (A)`
	`!$OMP DO`
`DO I = 1,100`	`DO J = 1,300`
`DO J = 1,300`	`DO I = 1,100`
`A(I,J) = A(I+1,J) + 1`	`A(I,J) = A(I+1,J) + 1`
`END DO`	`END DO`
`END DO`	`END DO`
	`!$OMP END DO`
	`!$OMP END PARALLEL`

6.3.4.2 Balancing the Workload

On the DO directive, you can specify the SCHEDULE(GUIDED) clause to use guided self-scheduling in manually decomposed loops, which is effective for most loops. However, when the iterations contain a predictably unbalanced workload, you can obtain better performance by manually balancing the workload. To do this, specify the chunk size in the SCHEDULE clause of the DO directive.

In the following loop, it might be very inefficient to divide the iterations into chunks of 50. A chunk size of 25 would probably be much more efficient on a system with two processors, depending on the amount of work being done by the routine SUB.

```
DO I = 1,100
   .
   .
   .
   IF (I .LT. 50) THEN
       CALL SUB(I)
   END IF
   .
   .
   .
END DO
```

6.4 Environment Variables for Adjusting the Run-Time Environment

Note

This section contains information that applies to both the OpenMP Fortran API and the Compaq Fortran parallel compiler directives.

The OpenMP Fortran API and the Compaq Fortran parallel compiler directive sets also provide environment variables that adjust the run-time environment in unusual situations.

Regardless of whether you used the `-omp` or the `-mp` compiler option, when the compiler needs information supplied by an environment variable, the compiler first looks for an OpenMP Fortran API environment variable and then for a Compaq Fortran parallel compiler environment variable. If neither one is found, the compiler uses a default.

The compiler looks for environment variable information in the following situations:

- When entering a parallel region, it looks for the number of threads (`OMP_NUM_THREADS` or `MP_THREAD_COUNT`), the spin count (`MP_SPIN_COUNT`), the yield count (`MP_YIELD_COUNT`), and the stack size (`MP_STACK_SIZE`).
- When entering a DO or PARALLEL DO directive that has RUNTIME specified, it looks at schedule type (`OMP_SCHEDULE`).
- When entering a worksharing directive, it looks at chunk size (`MP_CHUNK_SIZE`).

The OpenMP Fortran API environment variables are listed in Table 6–4.

Table 6–4 OpenMP Fortran API Environment Variables

Environment Variable[1]	Interpretation
OMP_SCHEDULE	This variable applies only to DO and PARALLEL DO directives that have the schedule type of RUNTIME. You can set the schedule type and an optional chunk size for these loops at run time. The schedule types are STATIC, DYNAMIC, and GUIDED. For directives that have a schedule type other than RUNTIME, this variable is ignored. The compiler default schedule type is STATIC. If the optional chunk size is not set, a chunk size of one is assumed, except for the STATIC schedule type. For this schedule type, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.
OMP_NUM_THREADS	Use this environment variable to set the number of threads to use during execution. This number applies unless you explicitly change it by calling the OMP_SET_NUM_THREADS run-time library routine. When you have enabled dynamic thread adjustment, the value assigned to this environment variable represents the maximum number of threads that can be used. The default value is the number of processors in the current system.
OMP_DYNAMIC	Use this environment variable to enable or disable dynamic thread adjustment for the execution of parallel regions. When set to TRUE, the number of threads used can be adjusted by the run-time environment to best utilize system resources. When set to FALSE, dynamic adjustment is disabled. The default is FALSE.
OMP_NESTED	Use this environment variable to enable or disable nested parallelism. When set to TRUE, nested parallelism is enabled. When set to FALSE, it is disabled. The default is FALSE.

[1]Environment variable names must be in uppercase; the assigned values are *not* case-sensitive.

The Compaq Fortran parallel compiler environment variables are listed in Table 6–5.

Table 6–5 Compaq Fortran Parallel Environment Variables

Environment Variable[1]	Interpretation
MP_THREAD_COUNT	Specifies the number of threads the run-time system is to create. The default is the number of processors available to your process.
MP_CHUNK_SIZE	Specifies the chunk size the run-time system uses when dispatching loop iterations to threads if the program specified the RUNTIME schedule type or specified another schedule type requiring a chunk size, but omitted the chunk size. The default chunk size is 1.
MP_STACK_SIZE	Specifies how many bytes of stack space the run-time system allocates for each thread when creating it. If you specify zero, the run-time system uses the default, which is very small. Therefore, if a program declares any large arrays to be PRIVATE, specify a value large enough to allocate them. If you do not use this environment variable at all, the run-time system allocates 5 MB.
MP_SPIN_COUNT	Specifies how many times the run-time system spins while waiting for a condition to become true. The default is 16,000,000, which is approximately one second of CPU time.
MP_YIELD_COUNT	Specifies how many times the run-time system alternates between calling sched_yield and testing the condition before going to sleep by waiting for a thread condition variable. The default is 10.

[1]Environment variable names must be in uppercase; the assigned values are *not* case-sensitive.

6.5 Calls to Programs Written in Other Languages

Note

This section contains information that applies to both the OpenMP Fortran API and the Compaq Fortran parallel compiler directives.

Only programs written in Compaq Fortran support parallel directives. Any procedures or routines called from within a parallel region in a Compaq Fortran program must consider the following:

- Compile any Compaq Fortran programs containing parallel directives using the `-mp` or the `-omp` option.
- Called procedures or routines must be thread safe.

- It is the programmer's responsibility to ensure that all data objects in the called procedures or routines are shared or allocated on each thread's private stack.

6.6 Compiling, Linking, and Running Parallelized Programs on SMP Systems

Note

This section contains information that applies to both the OpenMP Fortran API and the Compaq Fortran parallel compiler directives.

Whether you compile and link your program in one step or in separate steps, you must include the name of the `f90` Compaq Fortran driver (and the `-omp` or `-mp` option if you want to use parallel compiler directives) on each command line. For example, to compile and link the program `prog.f` with its OpenMP Fortran API directives in one step, use the command:

```
% f90 -omp prog.f -o prog
```

To separately compile and link the program `prog.f`, use these commands:

```
% f90 -c -omp prog.f
% f90 -omp prog.o -o prog
```

To run your program, use the command:

```
% prog
```

When you use the `-omp` (or `-mp`) option, the driver sets the `-reentrancy threaded` and the `-automatic` options for the compiler if you did not specify them on the command line. The options are not set if you used the negated forms of the options on the command line. The driver also sets the `-pthread` and library options for the linker.

6.7 Debugging Parallelized Programs

Note

This section contains information that applies to both the OpenMP Fortran API and the Compaq Fortran parallel compiler directives.

When a Compaq Fortran program uses parallel decomposition directives, there are some special considerations concerning how that program can be debugged.

When a bug occurs in a Compaq Fortran program that uses parallel decomposition directives, the bug might be caused by incorrect Compaq Fortran statements, or it might be caused by incorrect parallel decomposition directives. In either case, the program to be debugged can be executed by multiple threads simultaneously.

6.7.1 Debugger Limitations for Parallelized Programs

Debuggers such as the Compaq Ladebug debugger provide features that support the debugging of programs that are executed by multiple threads. However, the currently available versions of Ladebug do not directly support the debugging of parallel decomposition directives, and therefore, there are limitations on the debugging features.

Other debuggers are available for use on UNIX. Before attempting to debug programs containing parallel decomposition directives, determine what level of support the debugger provides for these directives by reading the documentation or by contacting the supplier of the debugger.

Some of the new features used in OpenMP are not yet fully supported by the debuggers, so it is important to understand how these features work to understand how to debug them. The two problem areas are:

- Outlining of parallel regions (see Section 6.7.2)
- Shared variables (see Section 6.7.3)

6.7.2 Debugging Parallel Regions

The compiler implements a parallel region by taking the code in the region and putting it into a separate, compiler-created subroutine. This process is called outlining because it is the inverse of inlining a subroutine into its call site.

In place of the parallel region, the compiler inserts a call to a run-time library routine, which starts up threads and causes them to call the outlined routine. As threads return from the outlined routine, they return to the run-time library, which waits for all threads to finish before returning to the master thread in the original program.

Example 6–6 contains a section of the source listing with machine code (produced using `f90 -omp -V -machine_code`). Note that the original program unit was named `outline_example` and the parallel region was at line 2. The compiler created an outlined routine called `_2_outline_example_`. In general, the outlined routine is named *_line-number_original-routine-name*.

Example 6–6 Code Using Parallel Region

```
OUTLINE_EXAMPLE                 Source Listing

       1  program outline_example
       2 !$omp parallel
       3  print *, 'hello world'
       4 !$omp end parallel
       5  print *, 'done'
       6  end

OUTLINE_EXAMPLE                 Machine Code Listing

    .text
    .ent _2_outline_example_
    .eflag 16
             0000 _2_outline_example_:
27BB0001     0000  ldah gp, _2_outline_example_
23BD8180     0004  lda gp, _2_outline_example_
23DEFFA0     0008  lda sp, -96(sp)
B75E0000     000C  stq r26, (sp)
    .mask 0x04000000,-96
    .fmask 0x00000000,0
    .frame  $sp, 96, $26
    .prologue 1
A45D8040     0010  ldq r2, 48(gp)
A77D8020     0014  ldq r27, for_write_seq_lis
63FF0000     0018  trapb
47E17400     001C  mov 11, r0
265F0385     0020  ldah r18, 901(r31)
A67D8018     0024  ldq r19, 8(gp)
B3FE0008     0028  stl r31, var$0001
221E0008     002C  lda r16, var$0001
B41E0048     0030  stq r0, 72(sp)
47E0D411     0034  mov 6, r17
B45E0050     0038  stq r2, 80(sp)
2252FF00     003C  lda r18, -256(r18)
229E0048     0040  lda r20, 72(sp)
6B5B4000     0044  jsr r26, for_write_seq_lis
27BA0001     0048  ldah gp, _2_outline_example_
23BD8180     004C  lda gp, _2_outline_example_
A75E0000     0050  ldq
63FF0000     0054  trapb
23DE0060     0058  lda sp, 96(sp)
6BFA8001     005C  ret (r26)
    .end _2_outline_example_
Routine Size: 96 bytes,    Routine Base: $CODE$ + 0000
```

(continued on next page)

Example 6–6 (Cont.) Code Using Parallel Region

```
    .globl  outline_example_
    .ent outline_example_
    .eflag 16
      0060 outline_example_:
27BB0001     0060  ldah gp, outline_example_
23BD8180     0064  lda gp, outline_example_
A77D8038     0068  ldq r27, for_set_reentrancy
23DEFFA0     006C  lda sp, -96(sp)
A61D8010     0070  ldq r16, (gp)
B75E0000     0074  stq r26, (sp)
    .mask 0x04000000,-96
    .fmask 0x00000000,0
    .frame  $sp, 96, $26
    .prologue 1
6B5B4000     0078  jsr r26, for_set_reentrancy
27BA0001     007C  ldah gp, outline_example_
23BD8180     0080  lda gp, outline_example_
47FE0411     0084  mov sp, r17
A77D8028     0088  ldq r27, _OtsEnterParallelOpenMP
A61D8030     008C  ldq r16, _2_outline_example_
47FF0412     0090  clr r18
6B5B4000     0094  jsr r26, _OtsEnterParallelOpenMP
27BA0001     0098  ldah gp, outline_example_
47E09401     009C  mov 4, r1
23BD8180     00A0  lda gp, outline_example_
265F0385     00A4  ldah r18, 901(r31)
A47D8018     00A8  ldq r3, 8(gp)
A77D8020     00AC  ldq r27, for_write_seq_lis
A67D8018     00B0  ldq r19, 8(gp)
221E0008     00B4  lda r16, var$0001
20630008     00B8  lda r3, 8(r3)
B3FE0008     00BC  stl r31, var$0001
B43E0048     00C0  stq r1, 72(sp)
47E0D411     00C4  mov 6, r17
B47E0050     00C8  stq r3, 80(sp)
2252FF00     00CC  lda r18, -256(r18)
229E0048     00D0  lda r20, 72(sp)
6B5B4000     00D4  jsr r26, for_write_seq_lis
27BA0001     00D8  ldah gp, outline_example_
A75E0000     00DC  ldq r26, (sp)
23BD8180     00E0  lda gp, outline_example_
47E03400     00E4  mov 1, r0
23DE0060     00E8  lda sp, 96(sp)
6BFA8001     00EC  ret (r26)
    .end   outline_example_
```

(continued on next page)

Example 6–6 (Cont.) Code Using Parallel Region

In the preceding example, the run-time library routine _OtsEnterParallelOpenMP is responsible for creating threads (if they have not already been created) and causing them to call the outlined routine. The outlined routine is called once by each thread.

Debugging the program at this level is just like debugging a program that uses POSIX threads directly. Breakpoints can be set in the outlined routine just like any other routine (leave off the trailing underscore. However, all Compaq Fortran routines are appended with a trailing underscore, so the debugger automatically inserts it.

6.7.3 Debugging Shared Variables

When a variable appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause on some block, the variable is made private to the parallel region by redeclaring it in the block. SHARED data, however, is not declared in the outlined routine. Instead, it gets its declaration from the parent routine.

When in a debugger, you can switch from one thread to another. Each thread has its own program counter so each thread can be in a different place in the code. Example 6–7 shows a Ladebug session.

Example 6–7 Code Using Multiple Threads

```
% ladebug a.out
Welcome to the Ladebug Debugger Version 4.0-xx
------------------
object file name: a.out
Reading symbolic information ...done
(ladebug) stop in _2_outline_example
[#1: stop in subroutine _2_outline_example() ]
(ladebug) run
[1] stopped at [_2_outline_example:2 0x120002d14]
      2 !$omp parallel
```

(continued on next page)

Example 6–7 (Cont.) Code Using Multiple Threads

```
(ladebug) show thread

Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
>*   1 running                   throughput 11       default thread
    -1 blocked    kernel          fifo       32       manager thread
    -2 ready                      idle        0       null thread for VP 0x0
     2 ready      not started     throughput 11       <anonymous>
     3 ready      not started     throughput 11       <anonymous>
     4 ready      not started     throughput 11       <anonymous>
     5 ready      not started     throughput 11       <anonymous>
     6 ready      not started     throughput 11       <anonymous>
(ladebug)
```

Thread 1 is the master thread. Do not confuse debugger thread numbers with OpenMP thread numbers. The compiler numbers threads beginning at zero, but the debugger numbers threads beginning at 1. There are also two extra threads in the debugging process, numbered -1 and -2, for use by the kernel.

Thread 1 has started running and is currently stopped just inside the outlined routine. The other threads have not started running because the example session was run on a uniprocessor workstation. On a multiprocessor, the other threads can run on different processors, so switch processors and examine the stack as shown in Example 6–8.

Example 6–8 Code Using Multiple Processors

```
(ladebug) thread 2

Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
>    2 ready      not started     throughput 11       <anonymous>

(ladebug) where
>0  0x3ff805739e0 in thdBase(0x14005d7d0, 0x0, 0x0, 0x120003c20, 0x4, 0x0)
(ladebug) thread 1
Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
>*   1 running                   throughput 11       default thread
```

(continued on next page)

Example 6–8 (Cont.) Code Using Multiple Processors

```
(ladebug) where
>0  0x120002d14 in _2_outline_example() omp_hello.f:2
#1  0x12000495c in _OtsEnterParallelOpenMP()
#2  0x120002d98 in outline_example() omp_hello.f:1
#3  0x120002ccc in main() for_main.c:203
(ladebug)
```

Thread 2 has not yet started and is reported as being in thdBase, a POSIX run-time support routine that threads run when they are created. Thread 1 is the master thread and is currently executing the outlined routine, called from the run-time library, which was called from the original program.

Note that only the master thread (thread 1) has a full call tree. The other threads have thdBase( ), from which they call the outlined routine. If you want to look at variables higher on the call stack than the parallel region, you must first tell the debugger to switch to thread 1, and then use the up command to climb the call stack.

If SHARED data is in common blocks, the outlined routine accesses it the same way any other routine would. If the SHARED data is automatic storage associated with the routine where the parallel region appears, however, each thread has a pointer to the master thread stack when the parallel region is reached.

Variables on the master stack can be accessed through the pointer. The compiler handles this automatically and does describe the access in the symbol table, but Ladebug and TotalView currently do not support this uplevel access mechanism.

Example 6–9 makes this clearer.

Example 6–9 Code Using Shared Variables

```
UPLEVEL                          Source Listing

      1       program uplevel
      2       implicit none
      3       integer i
```

(continued on next page)

Example 6–9 (Cont.) Code Using Shared Variables

```
      4
      5 !$omp parallel
      6 !$omp atomic
      7       i = i + 1
      8 !$omp end parallel
      9
     10       print *, i
     11       end

UPLEVEL                          Machine Code Listing

    .text
    .ent _5_uplevel_
    .eflag 16
      0000 _5_uplevel_:
23DEFFC0     0000  lda  sp, -64(sp)
    .frame  $sp, 64, $26
    .prologue 0
47E10402     0004  mov  r1, __StaticLink.1 # r1, r2
63FF0000     0008  trapb
20620010     000C  lda  r3, 16(r2)
      0010  L$3:
A8230000     0010  ldl_l r1, (r3)
40203000     0014  addl r1, 1, r0
B8030000     0018  stl_c r0, (r3)
E4000003     001C  beq r0, L$4
63FF0000     0020  trapb
23DE0040     0024  lda sp, 64(sp)
6BFA8001     0028  ret (r26)
      002C  L$4:
C3FFFFF8     002C br L$3
     .end _5_uplevel_

Routine Size: 48 bytes,    Routine Base: $CODE$ + 0000

     .globl  uplevel_
     .ent uplevel_
     .eflag 16
       0030 uplevel_:
27BB0001     0030  ldah gp, uplevel_ # gp, (r27)
23BD8130     0034  lda gp, uplevel_ # gp, (gp)
23DEFFA0     0038  lda sp, -96(sp)
B75E0000     003C  stq r26, (sp)
     .mask 0x04000000,-96
     .fmask 0x00000000,0
     .frame  $sp, 96, $26
     .prologue 1
```

(continued on next page)

Example 6–9 (Cont.) Code Using Shared Variables

```
A61D8010     0040  ldq r16, (gp)
A77D8038     0044  ldq r27, for_set_reentrancy # r27, 40(gp)
6B5B4000     0048  jsr r26, for_set_reentrancy # r26, (r27)
27BA0001     004C  ldah gp, uplevel_ # gp, (r26)
23BD8130     0050  lda gp, uplevel_ # gp, (gp)
A61D8030     0054  ldq r16, _5_uplevel_ # r16, 32(gp)
47FE0411     0058  mov sp, r17
47FF0412     005C  clr r18
A77D8028     0060  ldq r27, _OtsEnterParallelOpenMP # r27, 24(gp)
6B5B4000     0064  jsr r26, _OtsEnterParallelOpenMP # r26, (r27)
27BA0001     0068  ldah gp, uplevel_ # gp, (r26)
23BD8130     006C  lda gp, uplevel_ # gp, (gp)
B3FE0018     0070  stl r31, var$0001 # r31, 24(sp)
A67D8018     0074  ldq r19, 8(gp)
203E0010     0078  lda r1, I # r1, 16(sp)
B43E0058     007C  stq r1, 88(sp)
221E0018     0080  lda r16, var$0001 # r16, 24(sp)
47E0D411     0084  mov 6, r17
265F0385     0088  ldah r18, 901(r31)
2252FF00     008C  lda r18, -256(r18)
229E0058     0090  lda r20, 88(sp)
A77D8020     0094  ldq r27, for_write_seq_lis # r27, 16(gp)
6B5B4000     0098  jsr r26, for_write_seq_lis # r26, (r27)
27BA0001     009C  ldah gp, uplevel_ # gp, (r26)
23BD8130     00A0  lda gp, uplevel_ # gp, (gp)
47E03400     00A4  mov 1, r0
A75E0000     00A8  ldq r26, (sp)
23DE0060     00AC  lda sp, 96(sp)
6BFA8001     00B0  ret (r26)
     .end uplevel_

Routine Size: 132 bytes,    Routine Base: $CODE$ + 0030
```

Note that in this example in the main routine, the variable *i* is kept at offset 16 from the stack pointer. The stack pointer is passed into `_OtsEnterParallelOpenMP`, which puts it into register `r1` before calling `_5_uplevel_`. Each thread uses indirect address through this address to get to the shared *i*.

Because the debuggers have not yet been adjusted to understand uplevel addressing, the variable *i* does not appear to be declared in the outlined region, only in the original routine. To look at the value of the shared variable, we have to switch threads to the master thread and then get into the appropriate context. This is shown in Example 6–10.

Example 6–10 Code Looking at a Shared Variable Value

```
% ladebug a.out
Welcome to the Ladebug Debugger Version 4.0-xx
------------------
object file name: a.out
Reading symbolic information ...done
(ladebug) stop in _5_uplevel
[#1: stop in subroutine _5_uplevel() ]
(ladebug) run
[1] stopped at [_5_uplevel:5 0x120002cd8]
      5 !$omp parallel
(ladebug) where
>0  0x120002cd8 in _5_uplevel() omp_uplevel.f:5
#1  0x1200048ec in _OtsEnterParallelOpenMP
#2  0x120002d34 in uplevel() omp_uplevel.f:1
#3  0x120002c9c in main() for_main.c:203
(ladebug) p i
0
(ladebug) c
[1] stopped at [_5_uplevel:5 0x120002cd8]
      5 !$omp parallel
(ladebug) show thread
Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
     1 ready                     throughput 11       default thread
    -1 blocked    kernel          fifo       32       manager thread
    -2 ready                     idle        0       null thread for VP 0x0
>*   2 running                   throughput 11       <anonymous>
     3 ready      not started     throughput 11       <anonymous>
     4 ready      not started     throughput 11       <anonymous>
     5 ready      not started     throughput 11       <anonymous>
     6 ready      not started     throughput 11       <anonymous>

(ladebug) p i
Error: no value for symbol I
Error: no value for i
```

(continued on next page)

Example 6–10 (Cont.) Code Looking at a Shared Variable Value

```
(ladebug) thread 1
Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
>    1 ready                     throughput 11       default thread

(ladebug) where
>0  0x12000493c in _OtsEnterParallelOpenMP
#1  0x120002d34 in uplevel() omp_uplevel.f:1
#2  0x120002c9c in main() for_main.c:203
(ladebug) p i
1
(ladebug) c
[1] stopped at [_5_uplevel:5 0x120002cd8]
      5 !$omp parallel
(ladebug) show thread

Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
     1 ready                     throughput 11       default thread
    -1 blocked    kernel          fifo       32       manager thread
    -2 ready                      idle        0       null thread for VP 0x0
     2 ready                     throughput 11       <anonymous>
>*   3 running                   throughput 11       <anonymous>
     4 ready      not started     throughput 11       <anonymous>
     5 ready      not started     throughput 11       <anonymous>
     6 ready      not started     throughput 11       <anonymous>

(ladebug) where
>0  0x120002cd8 in _5_uplevel() omp_uplevel.f:5
#1  0x120003d90 in slave_main(arg=2) ots_parallel.bli:859
#2  0x3ff80573ea4 in thdBase(0x0, 0x0, 0x0, 0x1, 0x45586732, 0x3)
    DebugInformationStrippedFromFile101
(ladebug) p i
Error: no value for symbol I
Error: no value for i
(ladebug) thread 1
Thread State      Substate        Policy     Priority Name
------ ---------- --------------- ---------- -------- -------------
>    1 ready                     throughput 11       default thread
```

(continued on next page)

Example 6–10 (Cont.) Code Looking at a Shared Variable Value

```
(ladebug) up
>1  0x120002d34 in uplevel() omp_uplevel.f:1
      1       program uplevel
(ladebug) p i
2
(ladebug) q
%
```

7

Compaq Fortran Input/Output (I/O)

This chapter describes input/output (I/O) as implemented for Compaq Fortran and discusses the following topics:

- Section 7.1, Logical I/O Units
- Section 7.2, Types of I/O Statements
- Section 7.3, Forms of I/O Statements
- Section 7.4, Types of Files and File Characteristics
- Section 7.5, Opening Files: OPEN Statement
- Section 7.6, Obtaining File Information: INQUIRE Statement
- Section 7.7, Closing a File: CLOSE Statement
- Section 7.8, Record Operations
- Section 7.9, User-Supplied OPEN Procedures: USEROPEN Specifier
- Section 7.10, Format of Compaq Fortran Record Types

For More Information:

- On native data types, see Chapter 9.
- On reading and writing nonnative unformatted numeric data (originally written to the file using unformatted I/O statements), see Chapter 10.
- On porting Compaq OpenVMS Fortran data, see Section A.4.
- On performing I/O to the same unit with Compaq Fortran and Compaq Fortran 77 object files, see Section A.5.3.

7.1 Logical I/O Units

In Compaq Fortran, a **logical unit** is a channel through which data transfer occurs between the program and a device or file. You identify each logical unit with a logical unit number, which can be any nonnegative integer from 0 to a maximum value of 2,147,483,647 (2**31–1). For example:

```
READ (2,100) I,X,Y
```

This READ statement specifies that data is to be entered from the device or file corresponding to logical unit 2, in the format specified by the FORMAT statement labeled 100. When opening a file, use the UNIT specifier to indicate the unit number.

Fortran 95/90 (Fortran) programs are inherently device-independent. The association between the logical unit number and the physical file can occur at run-time. Instead of changing the logical unit numbers specified in the source program, you can change this association at run time to match the needs of the program and the available resources. For example, before running the program, a script file can set the appropriate environment variable or allow the terminal user to type a directory path, file name, or both.

Use the same logical unit number specified in the OPEN statement for other I/O statements to be applied to the opened file, such as READ and WRITE.

The OPEN statement connects a unit number with an **external file** and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers (all files except internal files are called external files).

Certain unit numbers are **preconnected** to standard devices. Unit number 5 is associated with `stdin`, unit 6 with `stdout`, and unit 0 with `stderr`. At run time, if units 5 and 6 are specified by a record I/O statement (such as READ or WRITE) without having been explicitly opened by an OPEN statement, Compaq Fortran implicitly opens units 5, 6, and 0 and associates them with their respective operating system standard I/O files (if the corresponding FORT*n* environment variable is not set).

For More Information:

On the OPEN statement and preconnected files, see Section 7.5.

7.2 Types of I/O Statements

Table 7–1 lists the Compaq Fortran I/O statements.

Table 7–1 Summary of I/O Statements

Category and Statement Name	Description
File Connection	
OPEN	Connects a unit number with an external file and specifies file connection characteristics.
CLOSE	Disconnects a unit number from an external file.
File Inquiry	
INQUIRE	Returns information about a named file, a connection to a unit, or the length of an output item list.
Record Position	
BACKSPACE	Moves the record position to the beginning of the previous record (sequential access only).
ENDFILE	Writes an end-of-file marker after the current record (sequential access only).
REWIND	Sets the record position to the beginning of the file (sequential access only).
Record Input	
READ	Transfers data from an external file record or an internal file to internal storage.
Record Output	
WRITE	Transfers data from internal storage to an external file record or to an internal file.
PRINT	Transfers data from internal storage to `stdout` (standard output device). Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.
Compaq Fortran Extensions	
ACCEPT	Reads input from `stdin`. Unlike READ, ACCEPT only provides formatted sequential output and does not specify a unit number.

(continued on next page)

Table 7–1 (Cont.) Summary of I/O Statements

Category and Statement Name	Description
DELETE	Marks a record at the current record position in a relative file as deleted (direct access only).
REWRITE	Transfers data from internal storage to an external file record at the current record position (direct access relative files only).
UNLOCK	Releases a lock held on the current record when file sharing was requested when the file was opened. Ignored in the current version of Compaq Fortran for Tru64 UNIX and Linux Systems (see Section 7.8.2).
TYPE	Writes record output to `stdout` (same as PRINT).
DEFINE FILE	Specifies file characteristics for a direct access relative file and connects the unit number to the file, similar to an OPEN statement. Provided for compatibility with compilers before FORTRAN-77.
FIND	Changes the record position in a direct access file. Provided for compatibility with compilers older than FORTRAN-77.

In addition to the READ, WRITE, REWRITE, TYPE, and PRINT statements, other I/O record-related statements are limited to a specific file organization. For instance:

- The DELETE statement only applies to relative files.[1]
- The BACKSPACE statement only applies to sequential files open for sequential access.
- The REWIND statement only applies to sequential files open for sequential access and to direct access files.
- The ENDFILE statement only applies to certain types of sequential files open for sequential access and to direct access files.

The file-related statements (OPEN, INQUIRE, and CLOSE) apply to any relative or sequential file.

[1] Detecting deleted records is only available if the `-vms` option was specified when the program was compiled. For more information on the `-vms` option, see Section 3.98.

7.3 Forms of I/O Statements

Each type of record I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. When opening a file, specify the form using the FORM specifier. The following are the forms of I/O statements:

- **Formatted I/O statements** contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.
- **List-directed** and **namelist I/O statements** are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist I/O statements use data types.
- **Unformatted I/O statements** do not contain format specifiers and therefore do not translate the data being transferred (important when writing data that will be read later).

Formatted, list-directed, and namelist I/O forms require translation of data from internal (binary) form within a program to external (readable character) form in the records. Consider using unformatted I/O for the following reasons:

- Unformatted data avoids the translation process, so I/O tends to be faster.
- Unformatted data avoids the loss of precision in floating-point numbers when the output data will subsequently be used as input data.
- Unformatted data conserves file storage space (stored in binary form).

To write data to a file using formatted, list-directed, or namelist I/O statements, specify FORM='FORMATTED' when opening the file. To write data to a file using unformatted I/O statements, specify FORM='UNFORMATTED' when opening the file.

Data written using formatted, list-directed, or namelist I/O statements is referred to as **formatted data**; data written using unformatted I/O statements is referred to as **unformatted data**.

When reading data from a file, you should use the same I/O statement form that was used to write the data to the file. For instance, if data was written to a file with a formatted I/O statement, you should read data from that file with a formatted I/O statement.

Although I/O statement form is usually the same for reading and writing data in a file, a program can read a file containing unformatted data (using unformatted input) and write it to a separate file containing formatted data (using formatted output). Similarly, a program can read a file containing formatted data and write it to a different file containing unformatted data.

As described in Section 7.8.2, you can access records in any sequential or relative file using sequential access. For relative files and certain (fixed-length) sequential files, you can also access records using direct access.

Table 7–2 shows the main record I/O statements, by category, that can be used in Compaq Fortran programs.

Table 7–2 Available I/O Statements and Record I/O Forms

File Type, Access, and I/O Form	Available Statements
External file, sequential access	
Formatted	READ, WRITE, PRINT, ACCEPT[1], TYPE[1], REWRITE[1,2]
List-Directed	READ, WRITE, PRINT, ACCEPT[1], TYPE[1], REWRITE[1,2]
Namelist	READ, WRITE, PRINT, ACCEPT[1], TYPE[1], REWRITE[1,2]
Unformatted	READ, WRITE, REWRITE[1,2]
External file, direct access	
Formatted	READ, WRITE, and REWRITE[1,2]
Unformatted	READ, WRITE, and REWRITE[1,2]
Internal file[3]	
Formatted	READ, WRITE
List-Directed	READ, WRITE
Unformatted	None

[1]This statement is a Compaq extension to the Fortran 95/90 standard.

[2]You can use the REWRITE statement only for relative files, using direct access.

[3]An internal file is a way to reference character data in a buffer using sequential access (see Section 7.4.2).

7.4 Types of Files and File Characteristics

This section discusses file organization, internal and scratch files, record type, record length, and other file characteristics.

7.4.1 File Organizations

File organization refers to the way records are physically arranged on a storage device.

Compaq Fortran supports two kinds of file organizations:

- **Sequential organization**
- **Relative organization**

The default file organization is always ORGANIZATION='SEQUENTIAL' for an OPEN statement. The organization of a file is specified by means of the ORGANIZATION specifier in the OPEN statement, as described in the *Compaq Fortran Language Reference Manual*.

You must store relative files on a disk device. You can store sequential files on magnetic tape or disk devices, and can use other peripheral devices, such as terminals, pipes, and line printers as sequential files.

Sequential Organization

A sequentially organized file consists of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). As a result, records can be added only at the end of the file.

Sequential files are usually read sequentially, starting with the first record in the file. Sequential files with a fixed-length record type that are stored on disk can also be accessed by relative record number (direct access).

Relative Organization

Within a relative file are numbered positions, called **cells**. These cells are of fixed equal length and are consecutively numbered from 1 to *n*, where 1 is the first cell, and *n* is the last available cell in the file. Each cell either contains a single record or is empty.

Records in a relative file are accessed according to cell number. A cell number is a record's relative record number (its location relative to the beginning of the file). By specifying relative record numbers, you can directly retrieve, add, or delete records regardless of their locations (direct access). (Detecting deleted records is only available if you specified the `-vms` option when the program was compiled. For information on the `-vms` option, see Section 3.98.)

When creating a relative file, specify the RECL value to determine the size of the fixed-length cells. Within the cells, you can store records of varying length, as long as their size does not exceed the cell size.

7.4.2 Internal Files and Scratch Files

Compaq Fortran also supports two other types of files that are not file organizations:

- Internal files
- Scratch files

Internal Files

When you use sequential access, you can use an internal file to reference character data in a buffer. The transfer occurs between internal storage and internal storage (unlike external files), such as between character variables and a character array.

An internal file consists of any of the following:

- Character variable
- Character-array element
- Character array
- Character substring
- Character array section without a vector subscript

Instead of specifying a unit number for the READ or WRITE statement, use an internal file specifier in the form of a character scalar memory reference or a character-array name reference.

An internal file is a designated internal storage space (variable buffer) of characters that is treated as a sequential file of fixed-length records. To perform internal I/O, use formatted and list-directed sequential READ and WRITE statements. You cannot use file-related statements such as OPEN and INQUIRE on an internal file (no unit number is used).

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the character variable, array element, or substring it contains. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined (a value has been assigned to the record).

Prior to each READ and WRITE statement, an internal file is always positioned at the beginning of the first record.

Scratch Files

Scratch files are created by specifying STATUS='SCRATCH' in an OPEN statement. By default, these temporary files are created in (and later deleted from) the directory specified in the OPEN statement DEFAULTFILE (if specified). To request a different directory to contain the scratch (temporary) files, set the `TMPDIR` environment variable.

7.4.3 Record Types, Record Overhead, and Maximum Record Length

Record type refers to whether records in a file are all the same length, are of varying length, or use other conventions to define where one record ends and another begins.

You can use fixed-length and variable-length record types with sequential or relative files. You can use any of the record types with sequential files. Relative files require the fixed-length record type.

Records are stored in one of the types described in Table 7–3. When creating a new file or opening an existing file, specify one of the record types listed in this table.

Table 7–3 Compaq Fortran Record Types

Record Type	Description
Fixed-length	Records in a file must contain the same length. You must specify the record length (RECL) when the file is opened and can obtain it before opening the file with unformatted data using a form of the INQUIRE statement (see Section 7.6.3). See Section 7.10.1, Fixed-Length Records.
Variable-length	Records in a file can vary in length. This is the most portable record type across multivendor U*X platforms (such as Compaq Tru64 UNIX). Record length information is stored as control bytes at the beginning and end of each record. See Section 7.10.2, Variable-Length Records.
Segmented	This pertains to a single logical record containing one or more unformatted records of varying length, which can only be used for unformatted sequential access. The segmented record type is unique to Compaq Fortran products. Avoid the segmented record type when the application requires that the same file be used for programs written in languages other than Fortran and for non-Compaq platforms. However, because the segmented record type is unique to Compaq Fortran products, segmented files can be ported across Compaq Fortran platforms. See Section 7.10.3, Segmented Records.
Stream	A stream file is not grouped into records and uses no record delimiters. Stream files contain character or binary data that is read or written to the extent of the variables specified. Specify CARRIAGECONTROL='NONE' for stream files. See Section 7.10.4, Stream File Data.
Stream_LF and Stream_CR	Records are of varying length where the line feed (LF) or the carriage return (CR) character serve as record delimiters (LF for Stream_LF files and CR for Stream_CR files). Stream_LF files must *not* contain embedded LF characters or use CARRIAGECONTROL='LIST'. Instead, specify CARRIAGECONTROL='NONE'. Stream_CR files must not contain embedded CR characters. The Stream_LF record type is the usual record type for text files. See Section 7.10.5, Stream_CR and Stream_LF Records.

Before you choose a record type, consider whether your application will use formatted or unformatted data. If you are using formatted data, you can choose any record type except segmented. If you are using unformatted data, avoid the stream, stream_CR, and stream_LF record types.

The segmented record type can only be used for unformatted sequential access with sequential files. You should not use segmented records for files that are read by programs written in languages other than Compaq Fortran products.

The stream, stream_CR, stream_LF, and segmented record types can be used only with sequential files.

7.4.3.1 Portability Considerations of Record Types

Consider the following portability needs when choosing a record type:

- Data files from Compaq Fortran on Compaq Tru64 UNIX and Linux Alpha systems, and Compaq Fortran 77 on Compaq Tru64 UNIX systems, are interchangeable.
- Data files from Compaq Fortran on Tru64 UNIX Alpha systems and Compaq Fortran on Linux Alpha systems are interchangeable.
- When using files with Compaq Fortran and Compaq Fortran 77 on Tru64 UNIX, Linux, and OpenVMS systems, use:
  - The segmented record type for unformatted data. However, be aware that the segmented record type is used only by Compaq Fortran and Compaq Fortran 77 but not by other Compaq languages.
  - The stream_LF format for formatted data.
- The stream_LF record type is usually the most portable on U*X systems.
- Any record type except segmented with other non-Fortran Compaq languages.

For more information on porting Compaq OpenVMS Fortran data, see Section A.4.

The default record type (RECORDTYPE) depends on the values for the ACCESS and FORM specifiers for the OPEN statement, as described in the *Compaq Fortran Language Reference Manual*.

The record type of the file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

For information on choosing the most efficient record type, see Section 5.6.9.

An I/O record is a collection of fields (data items) that are logically related and are usually processed as a unit.

Unless you specify nonadvancing I/O (ADVANCE specifier), each Compaq Fortran I/O statement transfers one record. The exceptions are described in Section 7.8.5.

7.4.3.2 Record Overhead

Record overhead refers to bytes associated with each record that are used internally by the file system and are not available when a record is read or written. Knowing the record overhead helps when estimating the storage requirements for an application. Although the overhead bytes exist on the storage media, do not include them when specifying the record length with the RECL specifier in an OPEN statement.

The various record types each require a different number of bytes for record overhead, as described in Table 7–4.

Table 7–4 Bytes Required for Record Overhead

Record Type	Organization	Record Overhead
Fixed-length	Sequential files	None.
Fixed-length	Relative files	None if the `-vms` option was omitted (the default). One byte if the `-vms` option was specified.
Variable-length	Sequential files	Eight bytes per record for overhead.
Segmented	Sequential files	Four bytes per record for overhead. One additional padding byte (space) is added if the specified record size is an odd number.
Stream	Sequential files	None required.
Stream_CR	Sequential files	One byte per record for overhead.
Stream_LF	Sequential files	One byte per record for overhead.

7.4.3.3 Maximum Record Length

For all but variable-length sequential records on 64-bit addressable systems, the maximum record length is 2.147 billion bytes (2,147,483,647 minus the bytes for record overhead). For variable-length sequential records on 64-bit addressable systems, the theoretical maximum record length is about 17,000 gigabytes. When considering very large record sizes, also consider limiting factors such as system virtual memory.

For More Information:

- On the detailed format of each record type, see Section 7.10.
- On converting nonnative numeric unformatted data, see Chapter 10.
- On I/O performance considerations, see Section 5.6.

- On porting data files from other vendors that can use different units for the record length, see Section 7.4.4.

7.4.4 Other File Characteristics

Other file characteristics include:

- Carriage control attributes of each record (CARRIAGECONTROL specifier)
- Whether formatted or unformatted data is contained in the records (FORM specifier)
- The record length (RECL specifier)

When you need to display or print formatted data that uses Fortran carriage control, consider using the `fpr` command as a filter through a pipe to reformat the records into operating system line printer format.

The units used for specifying record length depend on the form of the data:

- For formatted files (FORM='FORMATTED'), specify the record length in *bytes*.
- For unformatted files (FORM='UNFORMATTED'), specify the record length in *4-byte units*, unless you specify the `-assume byterecl` option to request 1-byte units (see Section 3.7).

For More Information:

- On statement syntax and specifier values (including defaults), see the *Compaq Fortran Language Reference Manual*.
- On file characteristics, see Section 7.5.2 and the OPEN statement in the *Compaq Fortran Language Reference Manual*.
- On the `fpr` command and carriage control characters, see `fpr`(1).
- On I/O performance considerations, see Section 5.6.

7.5 Opening Files: OPEN Statement

To open a file, you should use a preconnected file (such as for terminal output) or explicitly OPEN files. Although you can also implicitly open a file, this prevents you from using the OPEN statement to specify the file connection characteristics and other information.

7.5.1 Using Preconnected Standard I/O Files

If you do not use an OPEN statement to open logical unit 5, 6, or 0 and do not set the appropriate environment variable (FORT*n*), unit number 5 is associated with `stdin`, unit 6 with `stdout`, and unit 0 with `stderr`. At run time, Compaq Fortran implicitly opens (preconnects) units 5, 6, and 0 and associates them with their respective operating system standard I/O files if the corresponding FORT*n* environment variable is not set.

You can change these preconnected files by using one of the following:

- An OPEN statement to open unit 5, 6, or 0.

  When you explicitly OPEN a file for unit 5, 6, or 0, the OPEN statement keywords specify the file-related information to be used instead of the preconnected standard I/O file.

- Set the appropriate environment variable (FORT*n*) to redirect I/O to an external disk file.

  If you set the corresponding Compaq Fortran FORT*n* environment variable, the file specified by that Compaq Fortran environment variable is used.

Table 7–5 lists the Compaq Fortran environment variables and the standard I/O file associations for the preconnected files.

Table 7–5 Environment Variables and Preconnected Files

Unit	Compaq Fortran Environment Variable	Equivalent Tru64 UNIX Standard I/O File
5	FORT5	Standard input, `stdin`
6	FORT6	Standard output, `stdout`
0	FORT0	Standard error, `stderr`

To change the characteristics of the connection to a preconnected unit, explicitly open the preconnected unit number.

To redirect input or output from the standard preconnected files at run-time, you can set the appropriate Compaq Fortran I/O environment variable (see Section 7.5.7) or use the appropriate shell redirection character in a pipe (such as > or <).

For More Information:

- On the shell commands you can use to set or unset environment variables, see Appendix B.
- Other I/O environment variables recognized by Compaq Fortran, see Section 7.5.7.

7.5.2 OPEN Statement Specifiers

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. Once you open a file, you should close it before opening it again unless it is a preconnected file.

If you open a unit number that was opened previously (without being closed), one of the following occurs:

- If you specify a file specification that *does not* match the one specified for the original open, the Compaq Fortran run-time system closes the file and then reopens it.

  This resets the current record position for the second file.

- If you specify a file specification that *does* match the one specified for the original open, the file is reconnected without the internal equivalent of the CLOSE and OPEN.

  This lets you change one or more OPEN statement run-time specifiers while maintaining the record position context.

You can use the INQUIRE statement (see Section 7.6) to obtain information about a whether or not a file is opened by your program.

Especially when creating a new file using the OPEN statement, examine the defaults (see the description of the OPEN statement in the *Compaq Fortran Language Reference Manual*) or explicitly specify file attributes with the appropriate OPEN statement specifiers.

Table 7–6 lists the OPEN statement functions and their specifiers.

Table 7–6 OPEN Statement Functions and Specifiers

Category, Functions, and OPEN Statement Specifiers
Identify File and Unit
UNIT specifies the logical unit number.
FILE (or NAME[1]) and DEFAULTFILE[1] specify the directory and/or file name of an external file.
STATUS or TYPE[1] indicates whether to create a new file, overwrite an existing file, open an existing file, or use a scratch file.
STATUS or DISPOSE[1] specifies the file existence status after CLOSE.
File and Record Characteristics
ORGANIZATION[1] indicates the file organization (sequential or relative).
RECORDTYPE[1] indicates which record type to use.
FORM indicates whether records are formatted or unformatted.
CARRIAGECONTROL[1] indicates the terminal control type.
RECL or RECORDSIZE[1] specifies the record size (see Section 7.4.4).
Special File Open Routine
USEROPEN[1] names the routine that will open the file to establish special context that changes the effect of subsequent Compaq Fortran I/O statements (see Section 7.9).
File Access, Processing, and Position
ACCESS indicates the access mode (direct or sequential).
SHARED[1] indicates that other users can access the same file and activates record locking. Ignored in the current version of Compaq Fortran for Tru64 UNIX and Linux Systems (see Section 7.8.2).
POSITION indicates whether to position the file at the beginning of file, before the end-of-file record, or leave it as is (unchanged).
ACTION or READONLY[1] indicates whether statements will be used to only read records, only write records, or both read and write records.
MAXREC[1] specifies the maximum record number for direct access.
ASSOCIATEVARIABLE[1] specifies the variable containing next record number for direct access.
Record Transfer Characteristics
BLANK indicates whether to ignore blanks in numeric fields.

[1]This specifier is a Compaq Fortran extension.

(continued on next page)

Table 7–6 (Cont.) OPEN Statement Functions and Specifiers

Category, Functions, and OPEN Statement Specifiers
DELIM specifies the delimiter character for character constants in list-directed or namelist output.
PAD, when reading formatted records, indicates whether padding characters should be added if the item list and format specification require more data than the record contains.
BLOCKSIZE[1] specifies the block physical I/O buffer size.
BUFFERCOUNT[1] specifies the number of physical I/O buffers.
CONVERT[1] specifies the format of unformatted numeric data (see Chapter 10).
Error Handling Capabilities
ERR specifies a label to branch to if an error occurs.
IOSTAT specifies the integer variable to receive the error (IOSTAT) number if an error occurs.
File Close Action
DISPOSE identifies the action to take when the file is closed.

[1]This specifier is a Compaq Fortran extension.

For More Information:

- On specifier syntax and complete information, see the *Compaq Fortran Language Reference Manual*.
- On the UNIT, FILE, and DEFAULTFILE specifiers, see Section 7.5.3 to Section 7.5.7.
- On the FORM specifier, see Section 7.3.
- On file organizations, see Section 7.4.1.
- On available record types, see Section 7.4.3 and Section 7.10.
- On the CARRIAGECONTROL specifier, see Section 7.4.4 and `fpr`(1).
- On the RECL (record length) specifier, see Section 7.4.4.
- On the ERR and IOSTAT specifiers, see Chapter 8.
- On obtaining file information using the INQUIRE statement, see Section 7.6.
- On closing files, see Section 7.7.
- Record I/O transfer, see Section 7.8.5.

- Record advancement, see Section 7.8.4.
- Record positioning, see Section 7.8.3.
- On I/O performance considerations, see Section 5.6.

7.5.3 Methods to Specify the Unit, File Name, and Directory

You can choose to assign files to logical units by using one of the following methods:

- By using default values, such as a preconnected unit. In the following example, the PRINT statement is associated with a preconnected unit (`stdout`) by default.

  ```
  PRINT *,100
  ```

  The READ statement associates the logical unit 7 with the file `fort.7` (because the FILE specifier was omitted) by default:

  ```
  OPEN (UNIT=7,STATUS='NEW')
  READ (7,100)
  ```

- By supplying a file name (and possibly a directory) in an OPEN statement. For example:

  ```
  OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
  ```

- By using environment variables. You can also use shell commands to set the appropriate environment variable to a value that indicates a directory (if needed) and a file name to associate a unit with an external file.

The following sections discuss these methods.

7.5.4 Accessing Files: Implied and Explicit File and Pathnames

Most I/O operations involve a disk file, keyboard, or screen display. Other devices can also be used:

- Sockets can be read from or written to if a USEROPEN routine (usually written in C) is used to open the socket.
- Pipes opened for read and write access block (wait until data is available) if you issue a READ to an empty pipe.
- Pipes opened for read-only access return EOF if you issue a READ to an empty pipe.

For information on USEROPEN routines, see Section 7.9.

You can access the terminal screen or keyboard by using preconnected files, as described in Section 7.5. Otherwise, this chapter discusses disk files.

A complete file specification consists of a file name usually preceded by a pathname that specifies a directory. The pathname can be in one of two forms:

- An **absolute pathname**, where the directory is specified relative to the root directory. The first character is a slash (/). For example, the following directory and file name refers to the file named testdata in the /usr/users/gdata directory:

  ```
  /usr/users/gdata/testdata
  ```

- A **relative pathname**, where the specified directory is relative to the current directory. Relative pathnames do not begin with a slash (/). The following example uses a relative pathname from the current directory /usr/users to refer to the same file testdata in the gdata/ subdirectory:

  ```
  gdata/testdata
  ```

Directory names and file names should not contain any operating system wildcard characters (such as *, ?, and the [ ] construct). You can use the tilde (~) character as the first character in a pathname to refer to a top-level directory as in the C shell.

When specifying files, keep in mind that trailing and leading blanks are removed from character expression names, but not from Hollerith (numeric array) names.

File names are case sensitive and can consist of uppercase and lowercase letters. For example, the following file names represent three different files:

```
myfile.for
MYfile.for
MYFILE.for
```

You can associate a logical unit with a directory (if needed) and file name by using an environment variable assignment (see Section 7.5.7) or by using the OPEN statement (see Section 7.5.6). When an OPEN statement provides a pathname that contains only a directory (no file name) and an environment variable provides the file name, the OPEN statement and environment variable can work together to provide the complete directory and file name.

7.5.5 How Compaq Fortran Applies a Default Pathname and File Name

Compaq Fortran provides the following possible ways of specifying all or part of a file specification (directory and file name), such as /usr/proj/testdata:

- The FILE specifier in an OPEN statement typically specifies only a file name (such as testdata) or contains both a directory and file name (such as /usr/proj/testdata).

- The DEFAULTFILE specifier (a Compaq extension) in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).
- If you used an implied OPEN or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name (see Section 7.5.7).

Compaq Fortran recognizes environment variables for each logical I/O unit number in the form of FORT*n*, where *n* is the logical I/O unit number. If a file name is not specified in the OPEN statement and the corresponding FORT*n* environment variable is not set for that unit number, Compaq Fortran generates a file name in the form `fort.`*n*, where *n* is the logical unit number.

Certain Compaq Fortran environment variables are recognized and preconnected files exist for certain unit numbers, as described in Section 7.5.7.

When using scratch files, you can use the TMPDIR environment variable to specify where the scratch file gets created (see Section 7.4.2).

Performing an implied OPEN means that the FILE and DEFAULTFILE specifier values are not specified and an environment variable is used, if present.

Examples of Applying Default Pathnames and File Names

For example, for an implied OPEN of unit number 3, Compaq Fortran would check the environment variable FORT3. If the environment variable FORT3 was set, its value is used. If it is not set, the system supplies the file name `fort.3`.

In Table 7–7, assume the current directory is `/usr/smith` and the I/O uses unit 1, as in the statement `READ (1,100)`.

Table 7–7 Examples of Applying Default Pathnames and File Names

OPEN FILE Value	OPEN DEFAULTFILE Value	FORT1 Environment Variable Value	Resulting Pathname
not specified	not specified	not specified	/usr/smith/fort.1 ❶
not specified	not specified	test.dat	/usr/smith/test.dat ❷
not specified	not checked	/usr/tmp/t.dat	/usr/tmp/t.dat ❸
not specified	/tmp	not specified	/tmp/fort.1 ❹
not specified	/tmp	testdata	/tmp/testdata ❺
not specified	/usr	lib/testdata	/usr/lib/testdata ❻
file.dat	/usr/group	not checked	/usr/group/file.dat ❼
/tmp/file.dat	not checked	not checked	/tmp/file.dat❽
file.dat	not specified	not specified	/usr/smith/file.dat❾

❶ The current directory is used and the unit number determines the file name.

❷ The current directory is used and the environment variable provides the file name.

❸ The environment variable provides both the directory and file name.

❹ The directory is provided by the OPEN DEFAULTFILE specifier value, and the unit number determines the file name.

❺ The directory is provided by the OPEN DEFAULTFILE specifier value, and the environment variable provides the file name.

❻ The directory is provided by the OPEN DEFAULTFILE specifier value, and the environment variable provides a subdirectory and file name.

❼ The directory is provided by the OPEN DEFAULTFILE specifier value, and the file name is provided by the OPEN FILE specifier value.

❽ The directory and file name are provided by the OPEN FILE specifier value.

❾ The current directory is used and the OPEN FILE specifier value provides the file name.

When the resulting file pathname begins with a tilde character (~), C shell style pathname substitution is used (regardless of what shell is being used), such as a top-level directory (below the root). For additional information on tilde pathname substitution, see csh(1).

Rules for Applying Default Pathnames and File Names

Compaq Fortran determines file name and the directory path based on certain rules. It determines a file name string as follows:

- If the FILE specifier is present, its value is used.
- If the FILE specifier is not present, Compaq Fortran examines the corresponding environment variable.
  - If the corresponding environment variable is set, that value is used.
  - If the corresponding environment variable is not set, a file name in the form `fort.`*n* is used.

Once Compaq Fortran determines the resulting file name string, it determines the directory (which optionally precedes the file name) as follows:

- If the resulting file name string contains an absolute pathname, it is used and the DEFAULTFILE specifier, environment variable, and current directory values are ignored.
- If the resulting file name string does not contain an absolute pathname, Compaq Fortran examines the DEFAULTFILE specifier and current directory value:
  - If the corresponding environment variable is set and specifies an absolute pathname, Compaq Fortran uses that value.
  - The DEFAULTFILE specifier value is examined and, if present, Compaq Fortran uses its value.
  - If the DEFAULTFILE specifier is not present, Compaq Fortran uses the current directory as an absolute pathname.

7.5.6 Coding File Locations in an OPEN Statement

You can use the FILE and DEFAULTFILE specifiers of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (The *Compaq Fortran Language Reference Manual* describes the OPEN statement in greater detail.) For example:

```
OPEN (UNIT=4, FILE='/usr/users/smith/test.dat', STATUS='OLD')
```

The file `test.dat` in directory `/usr/users/smith` is opened on logical unit 4. No defaults are applied since both the directory and file name were specified. The value of the FILE specifier can be a character constant, variable, or expression.

In the following interactive example, the user supplies the file name and the DEFAULTFILE specifier supplies the default values for the full pathname string. The file to be opened is in /usr/users/smith and is concatenated with the file name typed by the user into the variable DOC:

```
CHARACTER(LEN=9) DOC
WRITE (6,*)  'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC, DEFAULTFILE='/usr/users/smith',STATUS='OLD')
```

A slash is appended to the end of the default file string if it does not have one.

For an example program that reads a typed file name, uses the typed name to open a file, and handles such errors as the "file not found" error, see Example 8–2.

7.5.7 Using Environment Variables

You can use the environment variable mechanism of the operating system and shells to associate logical units with external files. For example, setting the environment variable FORT6 to a file lets you redirect stdout to the specified file (see Table 7–5).

Compaq Fortran attempts to use certain environment variables in the absence of a file name.

When using scratch files, you can use the TMPDIR environment variable to specify where the scratch file gets created (see Section 7.4.2).

Setting and Unsetting Environment Variables

Before program execution, you can use shell commands to specify a value for an environment variable. This specified value might be a directory and/or file name of an external file you want to associate with a preconnected unit or a specific unit number.

With the C Shell, use the setenv command to set an environment variable:

```
% setenv FORT8 /usr/users/smith/test.dat
```

To remove the association of an environment variable and an external file with the C shell, use the unsetenv command.

```
% unsetenv FORT8
```

With the Bourne shell (sh) and Korn shell (ksh) and bash shell *(L*X only)*, use the export command and assignment command to set the environment variable:

```
$ export FORT8
$ FORT8=/usr/users/smith/test.dat
```

To remove the association of an environment variable and an external file with the Bourne shell or Korn shell or bash shell *(L*X only)*, use the unset command:

```
$ unset FORT8
```

Implied Compaq Fortran Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and the use of an asterisk (*) in place of a unit number in READ and WRITE statements, do not include an explicit logical unit number. Each of these Fortran 95/90 statements uses an implicit internal logical unit number and environment variable. Each environment variable is in turn associated by default with one of the Fortran 95/90 file names that are associated with standard I/O files. Table 7–8 shows these relationships.

Table 7–8 Implicit Compaq Fortran Logical Units

Compaq Fortran Statement	Environment Variable When -vms Specified	Environment Variable When -vms Omitted	Standard I/O File Name
READ (*,f) iolist	FOR_READ	FORT5	stdin
READ f,iolist	FOR_READ	FORT5	stdin
ACCEPT f,iolist	FOR_ACCEPT	FORT5	stdin
WRITE (*,f) iolist	FOR_PRINT	FORT6	stdout
PRINT f,iolist	FOR_PRINT	FORT6	stdout
TYPE f,iolist	FOR_TYPE	FORT6	stdout

You can change the file associated with these Compaq Fortran environment variables, as you would any other environment variable, by means of the environment variable assignment command. For example, with the C shell:

```
% setenv FOR_READ /usr/users/smith/test.dat
```

After executing the preceding command, the environment variable for the READ statement using an asterisk refers to file test.dat in directory /usr/users/smith.

For More Information:

- On a list of Compaq Fortran I/O statements, see Table 7–1.
- On the shell commands you can use to set or unset environment variables, see Appendix B.
- On record I/O, see Section 7.8.

- On Compaq Fortran I/O statements and specifier values, including defaults, see Table 7–1.
- On statement syntax, see the *Compaq Fortran Language Reference Manual*.
- On the ERR and IOSTAT specifiers, see Chapter 8.
- On closing files, see Section 7.7.

7.6 Obtaining File Information: INQUIRE Statement

The INQUIRE statement returns information about a file and has three forms:

- Inquiry by unit
- Inquiry by file name
- Inquiry by output item list

7.6.1 Inquiry by Unit

An inquiry by unit is usually done for an opened (connected) file. An inquiry by unit causes the Compaq Fortran RTL to check whether the specified unit is connected or not. One of the following occurs:

- If the unit *is* connected:
  - The EXIST and OPENED specifier variables indicate a true value
  - The pathname and file name are returned in the NAME specifier variable (if the file is named)
  - Other information requested on the previously connected file is returned
  - Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement (see Table 7–6)
  - The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units (see Section 3.7).
- If the unit is *not* connected:
  - The OPENED specifier indicates a false value
  - The unit NUMBER specifier variable is returned as a value of –1
  - Any other information returned will be undefined or default values for the various specifiers

For example, the following INQUIRE statement shows whether unit 3 has a file connected (OPENED specifier) in logical variable I_OPENED, the name (case sensitive) in character variable I_NAME, and whether the file is opened for READ, WRITE, or READWRITE access in character variable I_ACTION:

```
INQUIRE (3, OPENED=I_OPENED, NAME=I_NAME, ACTION=I_ACTION)
```

7.6.2 Inquiry by File Name

An inquiry by name causes the Compaq Fortran RTL to scan its list of open files for a matching file name. One of the following occurs:

- If a match occurs:
  - The EXIST and OPENED specifier variables indicate a true value.
  - The pathname and file name are returned in the NAME specifier variable.
  - The UNIT number is returned in the NUMBER specifier variable.
  - Other information requested on the previously connected file is returned.
  - Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement (see Table 7–6).
  - The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units (see Section 3.7).
- If no match occurs:
  - The OPENED specifier variable indicates a false value.
  - The unit NUMBER specifier variable is returned as a value of –1.
  - The EXIST specifier variable indicates (true or false) whether the named file exists on the device or not.
  - If the file does exist, the NAME specifier variable contains the pathname and file name.
  - Any other information returned will be default values for the various specifiers, based on any information specified when calling INQUIRE.

The following INQUIRE statement returns whether the file named log_file is a file connected in logical variable I_OPEN, whether the file exists in logical variable I_EXIST, and the unit number in integer variable I_NUMBER:

```
INQUIRE (FILE='log_file', OPENED=I_OPEN, EXIST=I_EXIST, NUMBER=I_NUMBER)
```

7.6.3 Inquiry by Output Item List

Unlike inquiry by unit or inquiry by name, inquiry by output item list does not attempt to access any external file. It returns the length of a record for a list of variables that would be used for unformatted WRITE, READ, and REWRITE statements (REWRITE is a Compaq Fortran extension).

The following INQUIRE statement returns the maximum record length of the variable list in variable I_RECLENGTH. This variable is then used to specify the RECL value in the OPEN statement:

```
INQUIRE (IOLENGTH=I_RECLENGTH) A, B, H
OPEN (FILE='test.dat', FORM='UNFORMATTED', RECL=I_RECLENGTH, UNIT=9)
```

For an unformatted file, the RECL value is returned using 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

For More Information:

- On the INQUIRE statement and its specifiers, see the *Compaq Fortran Language Reference Manual*.
- On record I/O, see Section 7.8.
- On OPEN statement specifiers, see Table 7–6.
- On the `-assume byterecl` option, see Section 3.7.

7.7 Closing a File: CLOSE Statement

Usually, any external file opened should be closed by the same program before it completes. The CLOSE statement disconnects the unit and its external file. You must specify the unit number (UNIT specifier) to be closed.

You can also specify:

- Whether the file should be deleted or kept (STATUS specifier)
- Error handling information (ERR and IOSTAT specifiers)

To delete a file when closing it:

- In the OPEN statement, specify the ACTION keyword (such as ACTION='READ'). Avoid using the READONLY keyword, because a file opened using the READONLY keyword cannot be deleted when it is closed.
- In the CLOSE statement, specify the keyword STATUS='DELETE'.

If you opened an external file and did an inquire by unit, but do not like the default value for the ACCESS specifier, you can close the file and then reopen it, explicitly specifying the ACCESS desired.

There usually is no need to close preconnected units. Internal files are neither opened nor closed.

For More Information:

- On a list of Compaq Fortran I/O statements, see Table 7–1.
- On opening files using a C routine (USEROPEN) and then using Compaq Fortran I/O statements, see Section 7.9.
- On OPEN statement specifiers, see Section 7.5.2.
- On statement syntax and specifier values, see the *Compaq Fortran Language Reference Manual*.

7.8 Record Operations

After you open a file or use a preconnected file, you can use the following statements:

- READ, WRITE and PRINT to perform record I/O.
- BACKSPACE, ENDFILE, REWIND to set record position within the file.
- ACCEPT, DELETE, REWRITE, TYPE, DEFINE FILE, and FIND to perform various operations. These statements are Compaq extensions.

These statements are described in Section 7.2 and in the *Compaq Fortran Language Reference Manual*.

The record I/O statement must use the appropriate record I/O form (formatted, list-directed, namelist, or unformatted), as described in Section 7.3.

7.8.1 Record I/O Statement Specifiers

You can use the following specifiers with the READ and WRITE record I/O statements:

- UNIT specifies the unit number to or from which input or output will occur.
- END specifies a label to branch to if an error occurs; only applies to input statements like READ.
- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies an integer variable to contain the IOSTAT number if an error occurs.
- FMT specifies a label of a FORMAT statement.
- NML specifies a label of a NAMELIST statement.
- REC specifies a record number for direct access.

When using nonadvancing I/O, use the ADVANCE, EOR, and SIZE specifiers, as described in Section 7.8.4.

When using the REWRITE statement (a Compaq Fortran extension), you can use the UNIT, FMT, ERR, and IOSTAT specifiers.

For More Information:

- On specifier syntax and complete information, see the *Compaq Fortran Language Reference Manual*.
- On available record types, see Section 7.4.3 and Section 7.10.
- On the error-related record I/O specifiers ERR, END, and IOSTAT, see Chapter 8.
- On the ADVANCE, EOR, and SIZE specifiers, see Section 7.8.4.
- Record positioning, see Section 7.8.3.
- Record I/O transfer, see Section 7.8.5.
- Record advancement, see Section 7.8.4.

7.8.2 Record Access Modes and File Sharing

Record access refers to how records will be read from or written to a file, regardless of its organization. Record access is specified each time you open a file; it can be different each time. The type of record access permitted is determined by the combination of file organization and record type.

For instance, you can:

- Add records to a sequential file with ORGANIZATION='SEQUENTIAL' and POSITION='APPEND' (or use the Compaq extension ACCESS='APPEND').
- Add records sequentially by using multiple WRITE statements, close the file, and then open it again with ORGANIZATION='SEQUENTIAL' and ACCESS='SEQUENTIAL' (or ACCESS='DIRECT' if the sequential file has fixed-length records).

7.8.2.1 Sequential Access

Sequential access transfers records sequentially to or from files or I/O devices such as terminals. You can use sequential I/O with any type of supported file organization and record type.

If you select sequential access mode for files with sequential or relative organization, records are written to or read from the file starting at the beginning of the file and continuing through it, one record after another. A particular record can be retrieved only after all of the records preceding it have been read; new records can be written only at the end of the file.

7.8.2.2 Direct Access

Direct access transfers records selected by record number to and from either sequential files stored on disk with a fixed-length record type or relative organization files.

If you select direct access mode, you can determine the order in which records are read or written. Each READ or WRITE statement must include the relative record number, indicating the record to be read or written.

You can directly access a sequential disk file only if it contains fixed-length records. Because direct access uses cell numbers to find records, you can enter successive READ or WRITE statements requesting records that either precede or follow previously requested records. For example, the first of the following statements reads record 24; the second reads record 10:

```
READ (12,REC=24) I
READ (12,REC=10) J
```

7.8.2.3 Limitations of Record Access by File Organization and Record Type

You can use both access modes on sequential and relative files. However, direct access to a sequential organization file can only be done if the file resides on disk and contains fixed-length records.

Table 7–9 summarizes the types of access permitted for the various combinations of file organizations and record types.

Table 7–9 Allowed Record Access for File Organizations and Record Types

Organization	Record Type	Sequential Access	Direct Access
Sequential file	Fixed	Yes	Yes[1]
	Variable	Yes	No
	Segmented	Yes	No
	Stream	Yes	No
	Stream_CR	Yes	No
	Stream_LF	Yes	No
Relative file[1]	Fixed	Yes	Yes

[1]Direct access and relative files require that the file resides on a disk device.

7.8.2.4 File Sharing

Depending on the value specified by the ACTION (or READONLY) specifier in the OPEN statement, the file will be opened by your program for reading, writing, or both reading and writing records. This simply checks that the program itself executes the type of statements intended.

For performance reasons, record-locking and shared-file checking are not supported by the current version of Compaq Fortran on Compaq Tru64 UNIX and Linux systems. When you open the file, access is always granted, regardless of whether:

- The OPEN statement SHARED specifier was specified
- Other processes have already opened the file

Similarly, the UNLOCK statement is ignored using the current version of Compaq Fortran.

You might open a file for writing records (or reading and writing records) and know another process might simultaneously have the file open and be writing records. In this case, you need to coordinate access times among those processes to handle the possibility of simultaneous WRITE and REWRITE statements on the same record positions.

7.8.3 Specifying the Initial Record Position

When you open a disk file, you can use the OPEN statement's POSITION specifier to request one of the following initial record positions within the file:

- The initial position before the first record (POSITION='REWIND'). A sequential access READ or WRITE statement will read or write the first record in the file.
- A point beyond the last record in the file (POSITION='APPEND'), just before the end-of-file record, if one exists. For a new file, this is the initial position before the first record (same as 'REWIND'). You might specify 'APPEND' before you write records to an existing sequential file using sequential access.
- The current position (ASIS). This is usually used only to maintain the current record position when reconnecting a file. The second OPEN specifies the same unit number and specifies the same file name (or omits it), which leaves the file open, retaining the current record position.

  However, if the second OPEN specifies a different file name for the same unit number, the file will be closed and then opened, causing a loss of current record position.

The following I/O statements allow you to change the current record position:

- REWIND sets the record position to the initial position before the first record. A sequential access READ or WRITE statement would read or write the first record in the file.
- BACKSPACE sets the record position to the previous record in a file. Using sequential access, if you wrote record 5, issued a BACKSPACE to that unit, and then read from that unit, you would read record 5.
- ENDFILE writes an end-of-file marker. This is typically done after writing records using sequential access just before you close the file.

Unless you use nonadvancing I/O (see Section 7.8.4), reading and writing records usually advances the current record position by one record. As discussed in Section 7.8.5, more than one record might be transferred using a single record I/O statement.

7.8.4 Advancing and Nonadvancing Record I/O

After you open a file, if you omit the ADVANCE specifier (or specify ADVANCE='YES') in READ and WRITE statements, advancing I/O (normal FORTRAN-77 I/O) will be used for record access. When using advancing I/O:

- Record I/O statements transfer one entire record (or multiple records).
- Record I/O statements advance the current record position to a position before the next record.

You can request nonadvancing I/O for the file by specifying the ADVANCE='NO' specifier in a READ and WRITE statement. You can use nonadvancing I/O only for sequential access to external files using formatted I/O (not list-directed or namelist).

When you use nonadvancing I/O, the current record position does not change, and part of the record might be transferred, unlike advancing I/O where one entire record or records are always transferred.

You can alternate between advancing and nonadvancing I/O by specifying different values for the ADVANCE specifier ('YES' and 'NO') in the READ and WRITE record I/O statements.

When reading records with either advancing or nonadvancing I/O, you can use the END branch specifier to branch to a specified label when the end of the file is read.

Because nonadvancing I/O might not read an entire record, it also supports an EOR branch specifier to branch to a specified label when the end of the record is read. If you omit the EOR and the IOSTAT specifiers when using nonadvancing I/O, an error results when the end-of-record is read.

When using nonadvancing input, you can use the SIZE specifier to return the number of characters read. For example, in the following READ statement, SIZE=X (where variable X is an integer) returns the number of characters read in X and an end-of-record condition causes a branch to label 700:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

7.8.5 Record Transfer

I/O statements transfer all data as records. The amount of data that a record can contain depends on the following circumstances:

- With formatted I/O (except for fixed-length records), the number of items in the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.
- With namelist and list-directed output, the items listed in the NAMELIST statement or I/O statement list (in conjunction with the NAMELIST or list-directed formatting rules) determine the amount of data to be transferred.
- With unformatted I/O (except for fixed-length records), the I/O statement alone specifies the amount of data to be transferred.
- When you specify fixed-length records (RECORDTYPE='FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding).

Typically, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for a single I/O statement to transfer data from or to more than one record, depending on the form of I/O used.

7.8.5.1 Input Record Transfer

When using advancing I/O, if an input statement specifies fewer data fields (less data) than the record contains, the remaining fields are ignored.

If an input statement specifies more data fields than the record contains, one of the following occurs:

- For formatted input using advancing I/O, if the file was opened with PAD='YES', additional fields are read as spaces. If the file is opened with PAD='NO', an error occurs (the input statement should not specify more data fields than the record contains).

  For formatted input using nonadvancing I/O (ADVANCE='NO'), an end-of-record (EOR) condition is returned. If the file was opened with PAD='YES', additional fields are read as spaces.
- For list-directed input, another record is read.
- For namelist input, another record is read.
- For unformatted input, an error occurs.

7.8.5.2 Output Record Transfer

If an output statement specifies fewer data fields than the record contains (less data than required to fill a record), the following occurs:

- With fixed-length records (RECORDTYPE='FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding) in the form of spaces (for a formatted record) or zeros (for an unformatted record).
- With other record types, the fields present are written and those omitted are not written (might result in a short record).

If the output statement specifies more data than the record can contain, an error occurs, as follows:

- With formatted or unformatted output using fixed-length records.

  If the items in the output statement and its associated format specifier result in a number of bytes that exceed the maximum record length (RECL), an error occurs.
- With formatted or unformatted output *not* using fixed-length records.

  If the items in the output statement and its associated format specifier result in a number of bytes that exceed the maximum record length (RECL), the Compaq Fortran RTL attempts to increase the RECL value and write the longer record. To obtain the RECL value, use an INQUIRE statement.
- For list-directed output and namelist output, if the data specified exceeds the maximum record length (RECL), another record is written.

For More Information:

- On Compaq Fortran I/O statements, see Table 7–1.
- On record I/O specifiers, see Section 7.8.1.
- On statement syntax and specifier values, see the *Compaq Fortran Language Reference Manual*.
- On improving Compaq Fortran I/O performance, see Section 5.6.

7.9 User-Supplied OPEN Procedures: USEROPEN Specifier

You can use the USEROPEN specifier in a Compaq Fortran OPEN statement to pass control to a routine that directly opens a file. The called routine can use system calls or library routines to open the file and establish special context that changes the effect of subsequent Compaq Fortran I/O statements.

The Compaq Fortran Run-Time Library (RTL) I/O support routines call the USEROPEN function in place of the system calls usually used when the file is first opened for I/O. The USEROPEN specifier in an OPEN statement specifies the name of a function to receive control. The called function must open the file (or pipe) and return the file descriptor of the file when it returns control to the calling Compaq Fortran program.

When opening the file, the called function usually specifies options different from those provided by a normal OPEN statement.

You can obtain the file descriptor from the Compaq Fortran Run-Time Library (RTL) for a specific unit number with the `getfd` routine, described in `getfd`(3f).

Although the called function can be written in other languages (such as Fortran), C is usually the best choice for making system calls, such as `open` or `create`.

The USEROPEN specifier for the OPEN statement has the form:

```
USEROPEN = function-name
```

The function-name value represents the name of an external open function. In the calling Compaq Fortran program, the function must be declared in an EXTERNAL statement. For example, the following Compaq Fortran code might be used to call the USEROPEN procedure UOPEN (known to the linker as `uopen_`):

```
EXTERNAL  UOPEN
INTEGER   UOPEN
.
.
.
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW', USEROPEN=UOPEN)
```

After the OPEN statement, the uopen_ function receives control. The function opens the file, may perform other operations, and subsequently returns control (with the file descriptor) to the calling Compaq Fortran program.

If the USEROPEN function is written in C, declare it as a C function that returns a 4-byte integer (int) result to contain the file descriptor. For example:

```
int   uopen_ (                  ❶
      char  *file_name,         ❷
      int   *open_flags,        ❸
      int   *create_mode,       ❹
      int   *lun,               ❺
      int   file_length);       ❻
```

The function definition and the arguments passed from the Compaq Fortran RTL are as follows:

❶ The function must be declared as a 4-byte integer (int).

❷ The first argument is the pathname (includes the file name) to be opened.

❸ The open flags are described in the header file `/usr/include/sys/file.h` or `open`(2).

❹ The create mode (protection needed when creating a file), is described in `open`(2).

❺ The logical unit number.

❻ The fifth (last) argument is the pathname length (hidden length argument of the pathname).

Of the arguments, the `open` system call (see `open`(2)) requires the passed pathname, the open flags (that define the type access needed, whether the file exists, and so on), and the create mode. The logical unit number specified in the Compaq Fortran OPEN statement is passed in case the called function needs it. The hidden length of the pathname is also passed.

When creating a new file, the `create` system call might be used in place of `open` (see `create`(2)). You can usually use other appropriate system calls or library routines within the called function; restrictions are listed in Section 7.9.1.

In most cases, the called function modifies the open flags argument passed by the Compaq Fortran RTL or uses a new value before the `open` (or `create`) system call. After the called function opens the file, it must return control to the main Compaq Fortran program, which can do I/O with Fortran 95/90 statements to the file.

The `open` system call returns the file descriptor, which must be returned as a 4-byte integer to the Compaq Fortran program (and Compaq Fortran RTL). After control (and the file descriptor) is returned from the called function, the main Compaq Fortran program can perform I/O to that logical unit with Fortran 95/90 statements and eventually close it.

If the USEROPEN function is written in Fortran, declare it as a FUNCTION with an INTEGER (KIND=4) result, perhaps with an interface block. In any case, the called function must return the file descriptor as a 4-byte integer to the calling Compaq Fortran program.

If your application requires that you use C to perform the file open and close, as well as all record operations, call the appropriate C procedure from the Compaq Fortran program without using the Fortran OPEN statement. For more information on calling between Fortran and C, see Section 11.3.

7.9.1 Restrictions of Called USEROPEN Functions

The Compaq Fortran RTL uses exactly one file descriptor per logical unit, which must be returned by the called function. Because of this, only certain Compaq Tru64 UNIX system calls or library routines can be used to open the file.

System calls and library routines that do *not* return a file descriptor include `mknod` (see `mknod`(2)) and `fopen` (see `fopen`(3)). For example, the `fopen` routine returns a file pointer instead of a file descriptor.

7.9.2 Example USEROPEN Program and Function

The following Compaq Fortran code calls the USEROPEN function named UOPEN:

```
EXTERNAL  UOPEN
INTEGER   UOPEN
  .
  .
  .
OPEN (UNIT=1,FILE='ex1.dat',STATUS='NEW',USEROPEN=UOPEN, ERR=9,IOSTAT=errnum)
```

If the default `f90` options are used, the external name is passed using lowercase letters with an appended trailing underscore (_). In the preceding example, the external function UOPEN would be known as uopen_ to the linker and must be declared in C as uopen_. The function might be given the file name `uopen_.c`.

Compiling and Linking the C and Compaq Fortran Programs

Use a single `f90` command to compile the called uopen_ C function `uopen_.c` and the Compaq Fortran calling program `ex1.f`. The same command also links both object files by using the appropriate libraries to create the file `a.out` file, as follows:

```
% f90 ex1.f uopen_.c
```

If appropriate for large applications, you can specify object modules (`.o` files) on the `f90` command line. For more information on retaining object files, see Section 2.1.5.

Source Code for the C Function and Header File

Example 7–1 shows the C language function called `uopen_` and its associated header file.

Example 7–1 C Function Called by USEROPEN Procedure

```
/*
** File: uopen.h -- header file for uopen_.c
*/

#ifndef UOPEN
#define UOPEN 1

/*
**
**    Function Prototypes
**
*/
int   uopen_ (
      char  *file_name,    /* access read: name of the file to open. */
      int   *open_flags,   /* access read: READ/WRITE, see file.h or open(2)*/
      int   *create_mode,  /* access read: set if new file (to be created).*/
      int   *lun,          /* access read: logical unit file opened on.*/
      int  file_length);   /* access read: number of characters in file_name*/

#endif

/* End of file uopen.h */

/*
** File: uopen_.c
*/

/*
** This routine opens a file using data passed by Compaq Fortran RTL.
**
**  INCLUDE FILES
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include "uopen.h"/* Include file for this module */

int uopen_ (file_name, open_flags, create_mode, lun, file_length)
```

(continued on next page)

Example 7–1 (Cont.) C Function Called by USEROPEN Procedure

```
/*
** Open a file using the parameters passed by the calling Compaq
**   Fortran 95/90 program.
**
**  Formal Parameters:
*/

char  *file_name;    /* access read: name of the file to open. */
int   *open_flags;   /* access read: READ/WRITE, see file.h */
int   *create_mode;  /* access read: set if new file (to be created). */
int   *lun;          /* access read: logical unit number file opened on. */
int   file_length;   /* access read: number of characters in file_name. */

/*
**  Function Value/Completion Code
**
** Whatever is returned by open, is immediately returned to the
** Fortran OPEN.  the returned value is the following :
**    value >= 0 is a valid fd.
**    value <  0 is an error.
**
**  Modify open flags (logical OR) to specify the file be opened for
**  write access only, with records appended at the end (such as
**  writing to a shared log file).
*/

{
       int    result ;           /* Function result value */

       *open_flags =
             O_CREAT  |
             O_WRONLY |
             O_APPEND;

       result = open (file_name, *open_flags, *create_mode) ;

       return (result) ;         /* return file descriptor or error */

       }/* End of routine uopen_ */

       /* End of file uopen_.c */
```

Source Code for the Calling Compaq Fortran Program

Example 7–2 shows the Fortran 95/90 program that calls the uopen_ C function and then performs I/O.

Example 7–2 Compaq Fortran USEROPEN Main Calling Program

```
C
C  Program EX1 opens a file using USEROPEN and writes records to it.
C  It closes and re-opens the file (without USEROPEN) and reads 10 records.

 PROGRAM EX1

     EXTERNAL    UOPEN         ! The USEROPEN function.
     INTEGER     ERRNUM, CTR, I

 1   FORMAT (I)

     ERRNUM = 0

     WRITE (6,*) 'EX1. Access data using formatted I/O.'
     WRITE (6,*) 'EX1. Open file with USEROPEN and put some data in it.'
     OPEN (UNIT=1, FILE='ex1.dat', STATUS='NEW', USEROPEN=UOPEN, ERR=9, &
          IOSTAT=errnum)

     DO CTR=1,10
       WRITE (1,1) CTR
     END DO

     WRITE (6,*) 'EX1. Close and re-open without USEROPEN.'

     CLOSE (UNIT=1)

     OPEN (UNIT=1, FILE='ex1.dat', STATUS='OLD', FORM='FORMATTED', ERR=99, &
          IOSTAT=errnum)

     WRITE (6,*) 'EX1. Read and display what is in file.'

     DO CTR=1,10
        READ (1,1) i
        WRITE (6,*) i
     END DO

     WRITE (6,*) 'EX1.  Successful if 10 records shown.'

     CLOSE (UNIT=1,STATUS='DELETE')
     STOP

   9 WRITE (6,*) 'EX1.  Error on USEROPEN is ', errnum
     STOP

  99 WRITE (6,*) 'EX1.  Error on 2nd open is ', errnum

 END PROGRAM EX1
```

7.10 Format of Compaq Fortran Record Types

For general information, see Section 7.4.3, Record Types, Record Overhead, and Maximum Record Length and Table 7–3, Compaq Fortran Record Types.

7.10.1 Fixed-Length Records

When you specify fixed-length records, all records in the file contain the same number of bytes. When you open a file that is to contain fixed-length records, you must specify the record size by using the RECL specifier. A sequentially organized file opened for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

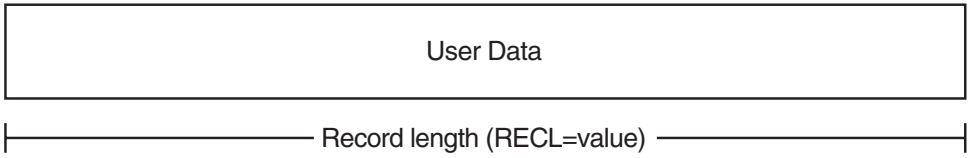
For relative files, the layout and overhead of fixed-length records depends on whether the program accessing the file was compiled with the `-vms` option or whether the `-vms` option was omitted:

- For relative files where the `-vms` option was omitted (the default), each record has no control information.
- For relative files where the `-vms` option was specified, each record has one byte of control information at the beginning of the record.

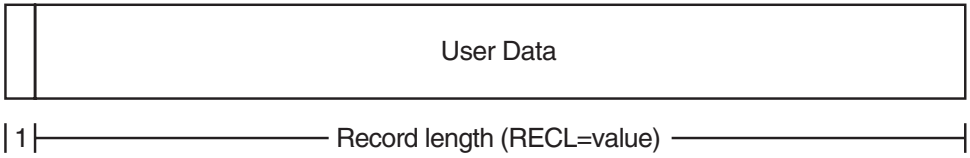
Figure 7–1 shows the record layout of fixed-length records.

Figure 7–1 Fixed-Length Records

For all sequential files and for relative files where the –vms option was omitted:



For relative files where the –vms option was specified:



ZK–9819–GE

For More Information:

- On the default value and size limit for fixed-length records, see the RECL specifier for the OPEN statement in the *Compaq Fortran Language Reference Manual*.

7.10.2 Variable-Length Records

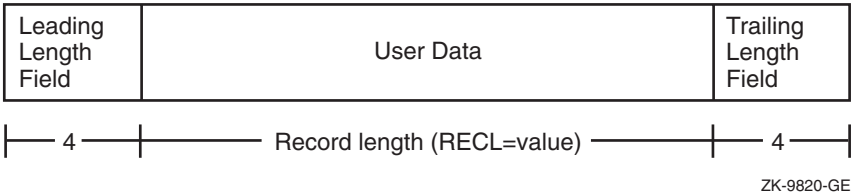
Variable-length records can contain any number of bytes up to a specified maximum record length, and apply only to sequential files.

Variable-length records are prefixed and suffixed by four bytes of control information containing length fields. The trailing length field allows a BACKSPACE request to skip back over records efficiently. The 4-byte integer value stored in each length field indicates the number of data bytes (excluding overhead bytes) in that particular variable-length record.

The character count field of a variable-length record is available when you read the record by issuing a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

The record layout of variable-length records that are less than two gigabytes appears in Figure 7–2.

Figure 7–2 Variable-Length Records Less Than 2 Gigabytes



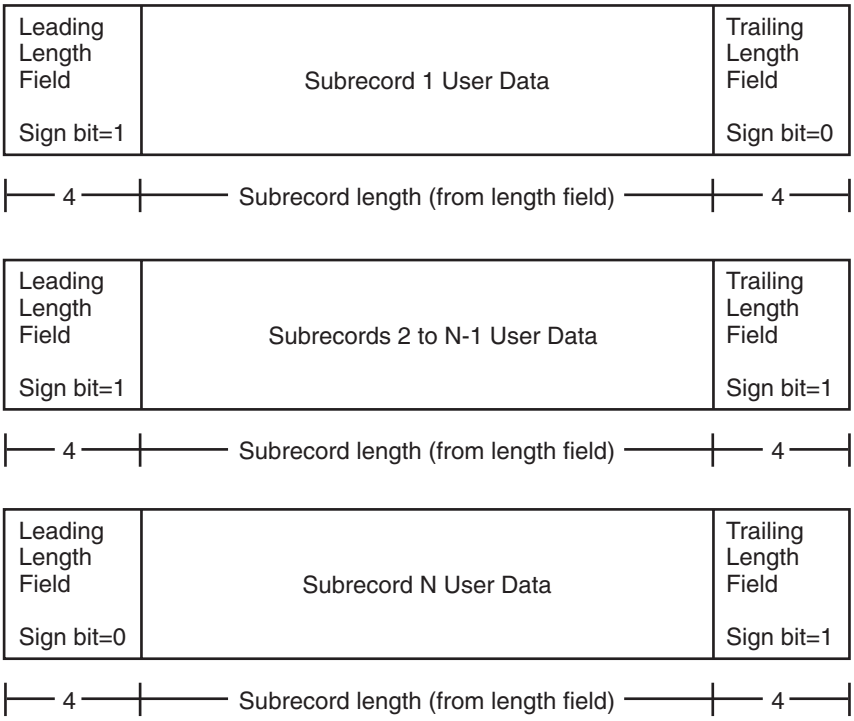
For a record length greater than 2,147,483,639 bytes, the record is divided into **subrecords**. The subrecord can be of any length from 1 to 2,147,483,639, inclusive.

The sign bit of the leading length field indicates whether the record is continued or not. The sign bit of the trailing length field indicates the presence of a preceding subrecord. The position of the sign bit is determined by the endian format of the file.

A subrecord that is continued has a leading length field with a sign bit value of 1. The last subrecord that makes up a record has a leading length field with a sign bit value of 0. A subrecord that has a preceding subrecord has a trailing length field with a sign bit value of 1. The first subrecord that makes up a record has a trailing length field with a sign bit value of 0.

The record layout of variable-length records that are greater than two gigabytes appears in Figure 7–3.

Figure 7–3 Variable-Length Records Greater Than 2 Gigabytes



Files written with variable-length records by Compaq Fortran programs usually cannot be accessed as text files. Instead, use the Stream_LF record format for text files with records of varying length.

7.10.3 Segmented Records

A segmented record is a single **logical record** consisting of one or more variable-length, unformatted records in a sequentially organized disk file. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

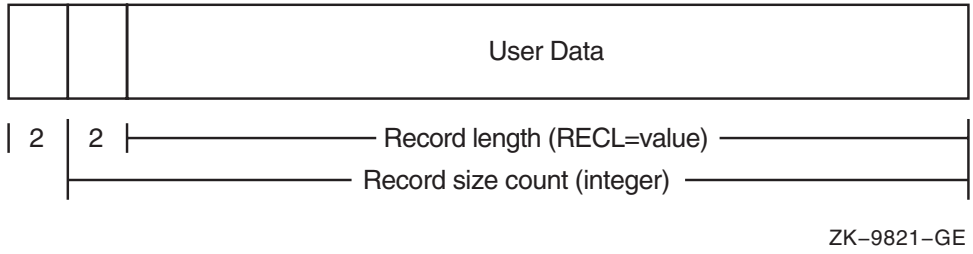
Segmented records are useful when you want to write exceptionally long records but cannot or do not wish to define one long variable-length record, perhaps because virtual memory limitations can prevent program execution. By using smaller, segmented records, you reduce the chance of problems caused by virtual memory limitations on systems on which the program may execute.

For disk files, the segmented record is a single logical record that consists of one or more segments. Each segment is a **physical record**. A segmented (logical) record can exceed the absolute maximum record length (2.14 billion bytes), but each segment (physical record) individually cannot exceed the maximum record length.

To access an unformatted sequential file that contains segmented records, specify FORM='UNFORMATTED' and RECORDTYPE='SEGMENTED' when you open the file. Otherwise, the file may be processed erroneously.

As shown in Figure 7–4, the layout of segmented records consists of four bytes of control information followed by the user data.

Figure 7–4 Segmented Records



The control information consists of a 2-byte integer record size count (includes the two bytes used by the segment identifier), followed by a 2-byte integer segment identifier that identifies this segment as one of the following:

Identifier Value	Segment Identified
0	One of the segments between the first and last segments.
1	First segment.
2	Last segment.
3	Only segment.

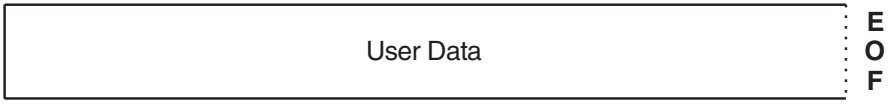
If the specified record length is an odd number, the user data will be padded with a single blank (one byte), but this extra byte is not added to the 2-byte integer record size count.

7.10.4 Stream File Data

A Stream file is not grouped into records and contains no control information. Stream files are used with CARRIAGECONTROL='NONE' and contain character or binary data that is read or written only to the extent of the variables specified on the input or output statement.

The layout of a Stream file appears in Figure 7–5.

Figure 7–5 Stream File Records



ZK–9822–GE

7.10.5 Stream_CR and Stream_LF Records

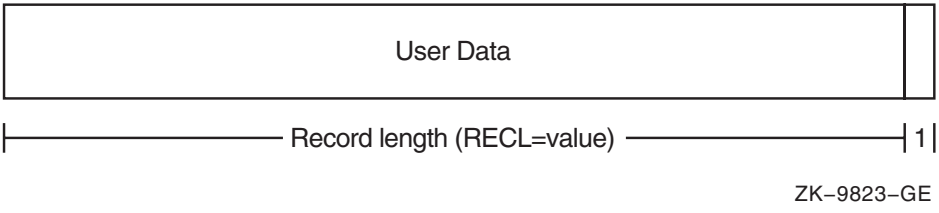
A Stream_CR or Stream_LF record is a variable-length record whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

Each variety uses a different 1-byte record terminator:

- Stream_CR files use only a carriage-return as the terminator, so Stream_CR files must not contain embedded carriage-return characters.
- Stream_LF files use only a line-feed (new line) as the terminator, so Stream_LF files must not contain embedded line-feed (new line) characters. This is the usual operating system text file record type.

The layout of Stream_CR and Stream_LF records appears in Figure 7–6.

Figure 7–6 Stream_CR and Stream_LF Records



ZK–9823–GE

8

Run-Time Errors and Signals

This chapter contains the following topics:

- Section 8.1, Compaq Fortran Run-Time Library Default Error Processing
- Section 8.2, Handling Run-Time Errors
- Section 8.3, Signal Handling
- Section 8.4, Run-Time Error Messages

8.1 Compaq Fortran Run-Time Library Default Error Processing

During execution, your program may encounter errors or exception conditions. These conditions can result from any of the following:

- Errors that occur during I/O operations
- Invalid input data
- Argument errors in calls to the mathematical library
- Arithmetic errors
- Other system-detected errors

The Compaq Fortran Run-Time Library (RTL) generates appropriate messages and takes action to recover from errors whenever possible.

A default action is defined for each error recognized by the Compaq Fortran RTL. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

The way in which the Compaq Fortran RTL actually processes errors depends upon the following factors:

- The severity of the error. For instance, the program usually continues executing when an error message with a severity level of warning or info (informational) is detected.

- For certain errors associated with I/O statements, whether or not an I/O error-handling specifier was specified.
- For certain errors, whether or not the default action of an associated signal was changed.
- For certain errors related to arithmetic operations (including floating-point exceptions), compilation options can determine whether the error is reported and the severity of the reported error.

How arithmetic exception conditions are reported and handled depends on the cause of the exception and how the program was compiled. Unless the program was compiled to handle exceptions, the exception might not be reported until *after* the instruction that caused the exception condition. The following `f90` command-line options are related to handling errors and exceptions:

- The `-check bounds`, `-check overflow`, and `-check underflow` options generate extra code to catch certain conditions. For example, the `-check overflow` option generates extra code to catch integer overflow conditions.
- The `-check noformat`, `-check nooutput_conversion`, and `-check nopower` options reduce the severity level of the associated run-time error to allow program continuation.
- The `-fpe`*n* options and the `-check underflow` option control the handling and reporting of floating-point arithmetic exceptions at run time.
- The `-synchronous_exceptions` option (and certain `-fpe`*n* option) influence the reporting of floating-point arithmetic exceptions at run time.
- The `-warn` *xxxx*, `-u`, `-nowarn -w`, and `-w1` options control compile-time warning messages, which in some circumstances can help determine the cause of a run-time error.

For More Information:

- On the `f90` command `-fpe`*n* options and the `for_set_fpe` routine that control how floating-point exceptional conditions are handled at run time, see Section 3.44 and Chapter 14.
- On the `-check bounds` option, see Section 3.23.
- On the `-check noformat` option, see Section 3.24.
- On the `-check nooutput_conversion` option, see Section 3.27.
- On the `-check overflow` option, see Section 3.28.
- On the `-check nopower` option, see Section 3.25.
- On the `-check underflow` option, see Section 3.29.

- On the f90 options that control warning messages, see Section 3.99.
- On IEEE floating-point data types and exceptional values, see Section 9.4.
- On f90 and fort command-line options and their categories, see Table 3–1.
- On Compaq Fortran intrinsic data types and their ranges, see Chapter 9.

8.1.1 Run-Time Message Format

When errors occur during program execution (run time) of a scalar (nonparallel) program, the Compaq Fortran RTL issues diagnostic messages. These run-time messages have the following format:

forrtl: *severity* (*nnn*): *message-text*

Run-time messages provide the following information:

Contents	Information Given
forrtl	Identifies the source as the Compaq Fortran RTL.
severity	The severity levels are: severe, error, warning, or info. (See Table 8–1, Severity Levels of Run-Time Messages.)
nnn	This is the message number, also the IOSTAT value for I/O statements.
message_text	Explains the event that caused the message.

Table 8–1 explains the severity levels of run-time messages, in the order of greatest to least severity.

Table 8–1 Severity Levels of Run-Time Messages

Severity	Description
severe	Must be corrected. The program's execution is terminated when the error is encountered, unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier (see Section 8.2.1 and Section 8.2.2).
error	Should be corrected. The program might continue execution, but the output from this execution may be incorrect.
warning	Should be investigated. The program continues execution, but output from this execution may be incorrect.
info	For informational purposes only. The program continues.

On Tru64 UNIX systems, for severe errors stack trace information is produced by default, unless the environment variable FOR_DISABLE_STACK_TRACE is set.

If symbols are in the image (that is, if the command-line option -g1 or higher is set and the image is not stripped), the stack trace information contains program counters set to symbolic information. Otherwise, the information contains merely hexadecimal program counter information.

In some cases, stack trace information is also produced by the compiled code at run time to provide details about the creation of array temporaries.

If FOR_DISABLE_STACK_TRACE is set, no stack trace information is produced.

Stack trace information is not produced on Linux Alpha systems.

See Example 8–1.

Example 8–1 Example of Stack Trace Information

```
program ovf                                    ❶
real*4 x(5),y(5)
integer*4 i

x(1) = -1e32
x(2) = 1e38
x(3) = 1e38
x(4) = 1e38
x(5) = -36.0

do i=1,5
   y(i) = 100.0*(x(i))
   print *, 'x = ', x(i), ' x*100.0 = ',y(i)
end do

end

> f90 -O0 ovf.for -o ovf.exe
> ovf.exe
 x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34    ❷
forrtl: error (72): floating overflow
   0: _call_remove_gp_range [0x3ff81a6c374]
   1: _call_remove_gp_range [0x3ff81a74464]
   2: _call_remove_gp_range [0x3ff800d8c60]
   3: ovf_ [ovf.for: 12, 0x1200019c4]
   4: main [for_main.c: 203, 0x1200018dc]
   5: __start [0x120001858]
Abort process
```

(continued on next page)

Example 8–1 (Cont.) Example of Stack Trace Information

```
> strip ovf.exe
> ovf.exe
 x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34    ❸
forrtl: error (72): floating overflow
Symbol table not present, doing non-symbolic traceback
   0: [0x3ff81a6c374]
   1: [0x3ff81a74464]
   2: [0x3ff800d8c60]
   3: [0x1200019c4]
   4: [0x1200018dc]
   5: [0x120001858]
Abort process

> setenv FOR_DISABLE_STACK_TRACE "TRUE"
> ovf.exe
 x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34 ❹
forrtl: error (72): floating overflow
Abort process
```

❶ Sample program that generates an error (at line 12).

❷ Stack trace information when the symbol table is present.

❸ Stack trace information when the image is stripped.

❹ No stack trace information, because the FOR_DISABLE_STACK_TRACE environment variable is set.

8.1.2 Message Catalog Location

The Compaq Fortran RTL uses a **message catalog file** to store the text associated with each run-time message. When a run-time error occurs, the Compaq Fortran RTL uses the environment variable NLSPATH to locate the message catalog file, from which it obtains the text of the appropriate message. If the file is not found at the position indicated by NLSPATH, the RTL searches for the message catalog at the following location:

/usr/lib/nls/msg/en_US.ISO8859-1/for_msg.cat *(TU*X only)*
/usr/lib/for_msg.cat *(L*X only)*

Before executing a Compaq Fortran program on a system where Compaq Fortran is *not* installed, you need to install the appropriate Compaq Fortran run-time subset. For instructions on installing Compaq Fortran run-time support, see the *Compaq Fortran Installation Guide for Tru64 UNIX Systems*.

When a run-time error occurs on a system where the message file is not found, the following messages may appear on a Tru64 UNIX system:

```
forrtl: info: Fortran error message number is nnn.

forrtl: warning: Could not open message catalog: for_msg.cat.

forrtl: info: Check environment variable NLSPATH and protection of
              usr/lib/nls/msg/en_US.ISO8859-1/for_msg.cat
```

The Compaq Fortran RTL returns an error number (displayed after the severity level) that the calling program can use with an IOSTAT variable to handle various I/O conditions, as described in Section 8.2.2.

For More Information:

- On NLSPATH, see the reference page environ(5).

8.1.3 Values Returned to the Shell at Program Termination

A Compaq Fortran program can terminate in one of several ways:

- The program runs to normal completion. A value of zero is returned to the shell.
- The program stops with a STOP or a PAUSE statement. A value of zero is returned to the shell.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of 1 is returned to the shell.
- The program stops because of a severe run-time error. The error number for that run-time error is returned to the shell. Error numbers are listed in Table 8–3.
- The program stops with a CALL EXIT statement. The value passed to EXIT is returned to the shell.

8.1.4 Forcing a Core Dump for Severe Errors

You can force a core dump for severe errors that do not usually cause a core file to be created. Before running the program, set the decfort_dump_flag environment variable to any of the common TRUE values (Y, y, Yes, yEs, True, and so forth) to cause severe errors to create a core file. For instance, the following C shell command sets the decfort_dump_flag environment variable:

```
% setenv decfort_dump_flag y
```

The core file is written to the current directory and can be examined using a debugger.

For More Information:

- On the shell commands you can use to set or unset environment variables, see Appendix B.
- On `core` files *(TU*X only)*, see `core`(4).
- On language syntax, see the *Compaq Fortran Language Reference Manual*.
- On file operations, see Section 7.5, Section 7.6, and Section 7.7.
- On record operations, see Section 7.8.

8.2 Handling Run-Time Errors

Whenever possible, the Compaq Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors. You can explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the ERR, EOR, and END branch specifiers in I/O statements (Section 8.2.1)
- To identify Fortran-specific I/O errors based on the value of Compaq Fortran RTL error codes, use the I/O status specifier (IOSTAT) in I/O statements (or call the ERRSNS subroutine) (Section 8.2.2)
- Obtain system-level error codes by using the appropriate 3f library routines (Section 8.2.3)
- For certain error conditions, use the signal handling facility to change the default action to be taken (Section 8.3)

These error-processing methods are complementary; you can use any or all of them within the same program to obtain Compaq Fortran run-time and Compaq Tru64 UNIX system error codes.

If your program generates an exception, use the `-synchronous_exceptions` option and recompile and relink your application (see Section 4.9).

8.2.1 Using the END, EOR, and ERR Branch Specifiers

When a severe error occurs during Compaq Fortran program execution, the default action is to display an error message and terminate the program. To override this default action, there are three branch specifiers you can use in I/O statements to transfer control to a specified point in the program:

- The END branch specifier handles an end-of-file condition.

- The EOR branch specifier handles an end-of-record condition for nonadvancing reads.
- The ERR branch specifier handles all error conditions.

If you use the END, EOR, or ERR branch specifiers, no error message is displayed and execution continues at the designated statement, usually an error-handling routine.

You might encounter an unexpected error that the error-handling routine cannot handle. In this case, do one of the following:

- Modify the error-handling routine to display the error message number
- Remove the END, EOR, or ERR branch specifiers from the I/O statement that causes the error

After you modify the source code, compile, link, and run the program to display the error message. For example:

```
READ (8,50,ERR=400)
```

If any severe error occurs during execution of this statement, the Compaq Fortran RTL transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
READ (12,70,END=550)
```

When using nonadvancing I/O, use the EOR specifier to handle the end-of-record condition. For example:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

You can also use ERR as a specifier in an OPEN, CLOSE, or INQUIRE statement. For example:

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this OPEN statement, control transfers to the statement at label 999.

For More Information:

- On advancing and nonadvancing record I/O, see Section 7.8.4.
- On language syntax, see the *Compaq Fortran Language Reference Manual*.
- On file operations, see Section 7.5, Section 7.6, and Section 7.7.
- On record operations, see Section 7.8.

8.2.2 Using the IOSTAT Specifier

You can use the IOSTAT specifier to continue program execution after an I/O error and to return information about I/O operations, whether the program is executing as a scalar or parallel program (I/O is done in a scalar fashion). As described in Table 8–3, certain errors are *not* returned in IOSTAT.

Although the IOSTAT specifier transfers control, it can only return information returned by the Compaq Fortran RTL.

The IOSTAT specifier can supplement or replace the END, EOR, and ERR branch transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses the display of an error message and defines the specified integer variable, array element, or scalar field reference as one of the following:

- A value of –2 if an end-of-record condition occurs with nonadvancing reads.
- A value of –1 if an end-of-file condition occurs.
- A value of 0 for normal completion (not an error condition, end-of-file, or end-of-record condition).
- A positive integer value if an error condition occurs. (This value is one of the Fortran-specific IOSTAT numbers listed in Table 8–3.)

Following the execution of the I/O statement and assignment of an IOSTAT value, control transfers to the END, EOR, or ERR statement label, if any. If there is no control transfer, normal execution continues.

You can include `/usr/include/foriosdef.f` in your program to obtain symbolic definitions for the values of IOSTAT.

Example 8–2 uses the IOSTAT specifier and the `foriosdef.f` file to handle an OPEN statement error (in the FILE specifier).

Example 8–2 Error Handling OPEN Statement File Name

```
CHARACTER(LEN=40) :: FILNM
INCLUDE '/usr/include/foriosdef.f'
```

(continued on next page)

Example 8–2 (Cont.) Error Handling OPEN Statement File Name

```
     DO I=1,4
       FILNM = ''
       WRITE (6,*)  'Type file name '
       READ (5,*) FILNM
       OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR, ERR=100)
       WRITE (6,*) 'Opening file: ', FILNM
!       (process the input file)
       CLOSE (UNIT=1)
       STOP

  100  IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
         WRITE (6,*) 'File: ', FILNM, ' does not exist '
       ELSE IF (IERR .EQ. FOR$IOS_FILNAMSPE) THEN
          WRITE (6,*) 'File: ', FILNM, ' was bad, enter new file name'
       ELSE
         PRINT *, 'Unrecoverable error, code =', IERR
         STOP
       END IF
     END DO
     WRITE (6,*) 'File not found. Type ls to find file and run again'
   END PROGRAM
```

Another way to obtain information about an error is the ERRSNS subroutine (a Compaq extension), which allows you to obtain the last I/O system error code associated with a Compaq Fortran RTL error (see the *Compaq Fortran Language Reference Manual*).

8.2.3 Using the 3f Library Routines to Return Operating System Errors

You can obtain the most recent operating system error codes or display associated error messages by using the following 3f library routines:

- The `ierrno` routine returns the most recent error code, as described in `ierrno`(3f). For a list of operating system error codes, see `errno`(2).
- The `gerror` and `perror` routines display a system error message, as described in `gerror`(3f) and `perror`(3f).

For More Information:

- On using the 3f routines, see Chapter 12 and `intro`(3f).
- On language syntax, see the *Compaq Fortran Language Reference Manual*.
- On file operations, see Section 7.5, Section 7.6, and Section 7.7.
- On record operations, see Section 7.8.

- On 3f library routines, see Chapter 12.

8.3 Signal Handling

A **signal** is an abnormal event generated by one of various sources, such as:

- A user of a terminal
- Program or hardware error
- Request of another program
- When a process is stopped to allow access to the control terminal

You can optionally set certain events to issue signals, for example:

- When a process resumes after being stopped
- When the status of a child process changes
- When input is ready at the terminal

Some signals terminate the receiving process if no action is taken (optionally creating a `core` file), while others are simply ignored unless the process has requested otherwise.

Except for certain signals, calling the `signal` or `sigaction` routine (see `signal`(3f)) allows specified signals to be ignored or causes an interrupt (transfer of control) to the location of a user-written signal handler.

You can establish one of the following actions for a signal with a call to `signal`:

- Ignore the specified signal (identified by number).
- Use the default action for the specified signal, which can reset a previously established action.
- Transfer control from the specified signal to a procedure to receive the signal, specified by name.

Calling the `signal` routine lets you change the action for a signal, such as intercepting an operating system signal and preventing the process from being stopped.

Table 8–2 shows the signals that the Compaq Fortran RTL arranges to catch when a program is started.

Table 8–2 Signals Caught by the Compaq Fortran Run-Time Library

Signal	Compaq Fortran RTL Message
SIGFPE	`Floating-point exception` (number 75)
SIGILL *(L*X only)*	`Illegal instruction (SIGILL)` and related SIGILL messages indicate various breakpoint exceptions (numbers 130-139)
SIGINT	`Process interrupted` (number 69)
SIGIOT	`IOT trap signal` (number 76)
SIGQUIT	`Process quit` (number 79)
SIGSEGV	`Segmentation fault` (number 174)
SIGTERM	`Process killed` (number 78)
SIGTRAP *(TU*X only)*	`Array index out of bounds (SIGTRAP)` and related SIGTRAP messages indicate various breakpoint exceptions (numbers 130-139)

Calling the `signal` routine (specifying the numbers for these signals) results in overwriting the signal-handling facility set up by the Compaq Fortran RTL. The only way to restore the default action is to save the returned value from the first call to `signal`.

When using a debugger, it may be necessary to enter a command to allow the Compaq Fortran RTL to receive and handle the appropriate signals, as described in Section 4.5.

For More Information:

- On the calling syntax of the `signal` jacket routine, see `signal`(3f).
- On using the 3f routines, such as `signal`, see Chapter 12 and `intro`(3f).
- On all signal names, their respective numbers, and a brief description of each signal, see `signal`(3) reference page or `/usr/include/signal.h`.

8.4 Run-Time Error Messages

Table 8–3 lists the errors processed by the Compaq Fortran RTL. For each error, the table provides the error number, the severity code, error message text, condition symbol name, and a detailed description of the errors.

To define the condition symbol values (PARAMETER statements) in your program, include the following file:

```
/usr/include/foriosdef.f
```

As described in Table 8–1, the severity of the message determines whether program execution continues (`info` and `warning`), the results may be incorrect (`error`), or program execution stops unless a recovery method is specified (`severe`).

When a `severe` error occurs for which no recovery method is specified, an error message is displayed and execution stops. To prevent program termination, you must include either an appropriate I/O error-handling specifier (see Section 8.2.1 and Section 8.2.2) and recompile or, for certain errors, change the default action of a signal (see Section 8.3) before you run the program again.

In Table 8–3, the first column lists error numbers returned to IOSTAT variables when an I/O error is detected. The Compaq Fortran error numbers are compatible with Compaq Fortran 77. Certain messages are specific to Compaq Fortran on Compaq Tru64 UNIX systems, including most signal-related errors (such as error number 79, `Process quit (SIGQUIT)` and errors 130-139) listed in Table 8–3.

The first line of the second column provides the message as it is displayed (following `forrtl:`), including the severity level, message number, and the message text. The following lines of the second column contain the status condition symbol (such as FOR$IOS_INCRECTYP) and an explanation of the message.

This table is also provided in the online release notes file in text format. To view the explanation of a message, use the `more` command to display the file and enter a slash (`/`) followed by the number or part of the message text (to initiate a search). For example, the following `more` command searches for number 68 (`Variable format expression value error`):

```
% more /68
```

Table 8–3 Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
None[1]	`info: Fortran error message number is` *nnn* The Compaq Fortran message catalog file was not found on this system. For information about the message file location, see Section 8.1.2 or the *Compaq Fortran Installation Guide for Tru64 UNIX Systems*. This error has no condition symbol.
None[1]	`warning: Could not open message catalog: for_msg.cat` The Compaq Fortran message catalog file was not found on this system. See Section 8.1.2 or the *Compaq Fortran Installation Guide for Tru64 UNIX Systems* for more information. This error has no condition symbol.
None[1]	`info: Check environment variable NLSPATH and protection of` *path-name*`/for_msg.dat` The Compaq Fortran message catalog file was not found. See Section 8.1.2 or the *Compaq Fortran Installation Guide for Tru64 UNIX Systems* for more information. This error has no condition symbol.
None[1]	`Insufficient memory to open Fortran RTL catalog: message 41` The Compaq Fortran message catalog file could not be opened because of insufficient virtual memory. To overcome this problem, increase the per-process data limit by using the `limit` (C shell) or `ulimit` (Bourne and Korn and bash *(L*X only)* shells) commands (see Section 1.1) before running the program again. For more information, see error 41. This error has no condition symbol.
1[1]	`severe (1): Not a Fortran-specific error` FOR$IOS_NOTFORSPE. An error in the user program or in the RTL was not a Compaq Fortran-specific error and was not reportable through any other Compaq Fortran run-time messages. If you call ERRSNS, an error of this kind returns a value of 1 (for more information on the ERRSNS subroutine, see the *Compaq Fortran Language Reference Manual*).
8	`severe (8): Internal consistency check failure` FOR$IOS_BUG_CHECK. Internal error. Please check that the program is correct. Recompile if an error existed in the program. If this error persists, submit a problem report.
9	`severe (9): Permission to access file denied` FOR$IOS_PERACCFIL. Check the mode (protection) of the specified file. Make sure the correct file was being accessed. Change the protection, specified file, or process used before rerunning program.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
10	`severe (10): Cannot overwrite existing file` FOR$IOS_CAOVEEXI. Specified file *xxx* already exists when OPEN statement specified STATUS='NEW' (create new file) using I/O unit *x*. Make sure correct file name, directory path, unit, and so forth were specified in the source program. Decide whether to: • Rename or remove the existing file before rerunning the program. • Modify the source file to specify different file specification, I/O unit, or OPEN statement STATUS.
11	`info (11)`[1]`: Unit not connected` FOR$IOS_UNINOTCON. The specified unit was not open at the time of the attempted I/O operation. Check if correct unit number was specified. If appropriate, use an OPEN statement to explicitly open the file (connect the file to the unit number).
17	`severe (17): Syntax error in NAMELIST input` FOR$IOS_SYNERRNAM. The syntax of input to a namelist-directed READ statement was incorrect.
18	`severe (18): Too many values for NAMELIST variable` FOR$IOS_TOOMANVAL. An attempt was made to assign too many values to a variable during a namelist READ statement.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
19	`severe (19): Invalid reference to variable in NAMELIST input` FOR$IOS_INVREFVAR. One of the following conditions occurred: • The variable was not a member of the namelist group. • An attempt was made to subscript a scalar variable. • A subscript of the array variable was out-of-bounds. • An array variable was specified with too many or too few subscripts for the variable. • An attempt was made to specify a substring of a noncharacter variable or array name. • A substring specifier of the character variable was out-of-bounds. • A subscript or substring specifier of the variable was not an integer constant. • An attempt was made to specify a substring by using an unsubscripted array variable.
20	`severe (20): REWIND error` FOR$IOS_REWERR. One of the following conditions occurred: • The file was not a sequential file. • The file was not opened for sequential or append access. • The Compaq Fortran RTL I/O system detected an error condition during execution of a REWIND statement.
21	`severe (21): Duplicate file specifications` FOR$IOS_DUPFILSPE. Multiple attempts were made to specify file attributes without an intervening close operation. A DEFINE FILE statement was followed by another DEFINE FILE statement or an OPEN statement
22	`severe (22): Input record too long` FOR$IOS_INPRECTOO. A record was read that exceeded the explicit or default record length specified when the file was opened. To read the file, use an OPEN statement with a RECL= value (record length) of the appropriate size.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
23	`severe (23): BACKSPACE error` FOR$IOS_BACERR. The Compaq Fortran RTL I/O system detected an error condition during execution of a BACKSPACE statement.
24[1]	`severe (24): End-of-file during read` FOR$IOS_ENDDURREA. One of the following conditions occurred: • A Compaq Fortran RTL I/O system end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. This error is returned by END and ERRSNS.
25	`severe (25): Record number outside range` FOR$IOS_RECNUMOUT. A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was opened.
26	`severe (26): OPEN or DEFINE FILE required` FOR$IOS_OPEDEFREQ. A direct access READ, WRITE, or FIND statement was attempted for a file when no prior DEFINE FILE or OPEN statement with ACCESS='DIRECT' was performed for that file.
27	`severe (27): Too many records in I/O statement` FOR$IOS_TOOMANREC. An attempt was made to do one of the following: • Read or write more than one record with an ENCODE or DECODE statement. • Write more records than existed.
28	`severe (28): CLOSE error` FOR$IOS_CLOERR. An error condition was detected by the Compaq Fortran RTL I/O system during execution of a CLOSE statement.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
29	`severe (29): File not found` FOR$IOS_FILNOTFOU. A file with the specified name could not be found during an open operation.
30	`severe (30): Open failure` FOR$IOS_OPEFAI. An error was detected by the Compaq Fortran RTL I/O system while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is issued when the error condition is not one of the more common conditions for which specific error messages are provided. It can occur when an OPEN operation was attempted for one of the following: • Segmented file that was not on a disk or a raw magnetic tape • Standard I/O file that had been closed
31	`severe (31): Mixed file access modes` FOR$IOS_MIXFILACC. An attempt was made to use any of the following combinations: • Formatted and unformatted operations on the same unit • An invalid combination of access modes on a unit, such as direct and sequential • A Compaq Fortran RTL I/O statement on a logical unit that was opened by a program coded in another language
32	`severe (32): Invalid logical unit number` FOR$IOS_INVLOGUNI. A logical unit number greater than 2,147,483,647 or less than zero was used in an I/O statement.
33	`severe (33): ENDFILE error` FOR$IOS_ENDFILERR. One of the following conditions occurred: • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential or append access. • An unformatted file did not contain segmented records. • The Compaq Fortran RTL I/O system detected an error during execution of an ENDFILE statement.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
34	`severe (34): Unit already open` FOR$IOS_UNIALROPE. A DEFINE FILE statement specified a logical unit that was already opened.
35	`severe (35): Segmented record format error` FOR$IOS_SEGRECFOR. An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was created by a program written in a language other than Fortran.
36	`severe (36): Attempt to access non-existent record` FOR$IOS_ATTACCNON. A direct-access READ or FIND statement attempted to access beyond the end of a relative file (or a sequential file on disk with fixed-length records) or access a record that was previously deleted in a relative file.
37	`severe (37): Inconsistent record length` FOR$IOS_INCRECLEN. An attempt was made to open a direct access file without specifying a record length.
38	`severe (38): Error during write` FOR$IOS_ERRDURWRI. The Compaq Fortran RTL I/O system detected an error condition during execution of a WRITE statement.
39	`severe (39): Error during read` FOR$IOS_ERRDURREA. The Compaq Fortran RTL I/O system detected an error condition during execution of a READ statement.
40	`severe (40): Recursive I/O operation` FOR$IOS_RECIO_OPE. While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted, such as a function subprogram that performs I/O to the same logical unit that was referenced in an expression in an I/O list or variable format expression.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
41	`severe (41): Insufficient virtual memory` FOR$IOS_INSVIRMEM. The Compaq Fortran RTL attempted to exceed its available virtual memory while dynamically allocating space. To overcome this problem, increase the per-process data limit by using the `limit` (C shell) or `ulimit` (Bourne and Korn and bash *(L*X only)* shell) commands before you run this program again (see Section 1.1). To determine whether the maximum per-process data size is already allocated, check the value of the *maxdsiz* parameter in the sysconfigtab or system configuration file. If necessary, increase its value. Changes to do not take effect until the system has been rebooted (you do not need to rebuild the kernel if you modify sysconfigtab). For more information about system configuration parameters on Tru64 UNIX systems, see the *Compaq Tru64 UNIX System Tuning and Performance Management* guide. Before you try to run this program again, wait until the new system resources take effect.
42	`severe (42): No such device` FOR$IOS_NO_SUCDEV. A pathname included an invalid or unknown device name when an OPEN operation was attempted.
43	`severe (43): File name specification error` FOR$IOS_FILNAMSPE. A pathname or file name given to an OPEN or INQUIRE statement was not acceptable to the Compaq Fortran RTL I/O system.
44	`severe (44): Inconsistent record type` FOR$IOS_INCRECTYP. The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.
45	`severe (45): Keyword value error in OPEN statement` FOR$IOS_KEYVALERR. An improper value was specified for an OPEN or CLOSE statement specifier requiring a value.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
46	`severe (46): Inconsistent OPEN/CLOSE parameters` FOR$IOS_INCOPECLO. Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow: • READONLY or ACTION='READ' with STATUS='NEW' or STATUS='SCRATCH' • READONLY with STATUS='REPLACE', ACTION='WRITE', or ACTION='READWRITE' • ACCESS='APPEND' with READONLY, ACTION='READ', STATUS='NEW', or STATUS='SCRATCH' • DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH' • DISPOSE='DELETE' with READONLY • CLOSE statement STATUS='DELETE' with OPEN statement READONLY • ACCESS='APPEND' with STATUS='REPLACE' • ACCESS='DIRECT' or 'KEYED' with POSITION='APPEND', 'ASIS', or 'REWIND'
47	`severe (47): Write to READONLY file` FOR$IOS_WRIREAFIL. A write operation was attempted to a file that was declared ACTION='READ' or READONLY in the OPEN statement that is currently in effect.
48	`severe (48): Invalid argument to Fortran Run-Time Library` FOR$IOS_INVARGFOR. The compiler passed an invalid or improperly coded argument to the Compaq Fortran RTL. This can occur if the compiler is newer than the RTL in use.
51	`severe (51): Inconsistent file organization` FOR$IOS_INCFILORG. The file organization specified in an OPEN statement did not match the organization of the existing file.
53	`severe (53): No current record` FOR$IOS_NO_CURREC. Attempted to execute a REWRITE statement to rewrite a record when the current record was undefined. To define the current record, execute a successful READ statement. You can optionally perform an INQUIRE statement on the logical unit after the READ statement and before the REWRITE statement. No other operations on the logical unit may be performed between the READ and REWRITE statements.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
55	`severe (55): DELETE error` FOR$IOS_DELERR. An error condition was detected by the Compaq Fortran RTL I/O system during execution of a DELETE statement.
57	`severe (57): FIND error` FOR$IOS_FINERR. The Compaq Fortran RTL I/O system detected an error condition during execution of a FIND statement.
58[1]	`info (58): Format syntax error at or near` *xx* FOR$IOS_FMTSYN. Check the statement containing *xx*, a character substring from the format string, for a format syntax error. For information about FORMAT statements, refer to the *Compaq Fortran Language Reference Manual*.
59	`severe (59): List-directed I/O syntax error` FOR$IOS_LISIO_SYN[2]. The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged.
60	`severe (60): Infinite format loop` FOR$IOS_INFFORLOO. The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
61	`severe` or `info`[3] `(61): Format/variable-type mismatch` FOR$IOS_FORVARMIS[2]. An attempt was made either to read or write a real variable with an integer field descriptor (I, L, O, Z, B), or to read or write an integer or logical variable with a real field descriptor (D, E, or F). To suppress this error message, see Section 3.24.
62	`severe (62): Syntax error in format` FOR$IOS_SYNERRFOR. A syntax error was encountered while the RTL was processing a format stored in an array or character variable.

[1]Identifies errors not returned by IOSTAT.

[2]The ERR transfer is taken after completion of the I/O statement for error numbers 59, 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

[3]For errors 61 and 63, the severity depends on the `-check` keywords used during compilation (`f90` command).

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
63	`error or info`[3] `(63): Output conversion error` FOR$IOS_OUTCONERR[2]. During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. When this situation is encountered, the overflowed field is filled with asterisks to indicate the error in the output record. If no ERR address has been defined for this error, the program continues after the error message is displayed. To suppress this error message, see Section 3.27.
64	`severe (64): Input conversion error` FOR$IOS_INPCONERR[2]. During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero.
65	`error (65): Floating invalid` FOR$IOS_FLTINV. During an arithmetic operation, the floating-point values used in a calculation were invalid for the type of operation requested or invalid exceptional values. For example, when requesting a log of the floating-point values 0.0 or a negative number. For certain arithmetic expressions, specifying the `-check nopower` option can suppress this message (see Section 3.25). For information on allowing exceptional IEEE values, see Section 3.44.
66	`severe (66): Output statement overflows record` FOR$IOS_OUTSTAOVE. An output statement attempted to transfer more data than would fit in the maximum record size.
67	`severe (67): Input statement requires too much data` FOR$IOS_INPSTAREQ. Attempted to read more data than exists in a record with an unformatted READ statement or with a formatted sequential READ statement from a file opened with a PAD specifier value of `'NO'`.
68	`severe (68): Variable format expression value error` FOR$IOS_VFEVALERR[2]. The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of 1 was assumed, except for a P edit descriptor, for which a value of zero was assumed.

[2]The ERR transfer is taken after completion of the I/O statement for error numbers 59, 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

[3]For errors 61 and 63, the severity depends on the `-check` keywords used during compilation (`f90` command).

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
69[1]	`error (69): Process interrupted (SIGINT)` FOR$IOS_SIGINT. The process received the signal SIGINT. Determine source of this interrupt signal (described in `signal`(3)).
70[1]	`severe (70): Integer overflow` FOR$IOS_INTOVF. During an arithmetic operation, an integer value exceeded byte, word, or longword range. The result of the operation was the correct low-order part. See Chapter 9 for ranges of the various integer data types. Consider specifying a larger integer data size (modify source program or, for an INTEGER declaration, possibly use the `f90` option `-integer_size` *nn*).
71[1]	`severe (71): Integer divide by zero` FOR$IOS_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. The result of the operation was set to the dividend, which is equivalent to division by 1.
72[1]	`error (72): Floating overflow` FOR$IOS_FLTOVF. During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type. See Chapter 9 for ranges of the various data types.
73[1]	`error (73): Floating divide by zero` FOR$IOS_FLTDIV. During a floating-point arithmetic operation, an attempt was made to divide by zero.
74[1]	`error (74): Floating underflow` FOR$IOS_FLTUND. During an arithmetic operation, a floating-point value became less than the smallest finite value for that data type. Depending on the values of the `-fpe`*n* option (see Section 3.44), the underflowed result was either set to zero or allowed to gradually underflow. See Chapter 9 for ranges of the various data types.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
75[1]	`error (75): Floating point exception` FOR$IOS_SIGFPE. A floating-point exception occurred. Core dump file created. Possible causes include: • Division by zero • Overflow • Invalid operation, such as subtraction of infinite values, multiplication of zero by infinity (without signs), division of zero by zero or infinity by infinity • Conversion of floating-point to fixed-point format when an overflow prevents conversion
76[1]	`error (76): IOT trap signal` FOR$IOS_SIGIOT. Core dump file created. Examine core dump for possible cause of this IOT signal. For more information about signals, see Section 8.3.
77[1]	`severe (77): Subscript out of range` FOR$IOS_SUBRNG. An array reference was detected outside the declared array bounds.
78[1]	`error (78): Process killed (SIGTERM)` FOR$IOS_SIGTERM. The process received the signal SIGTERM. Determine source of this software termination signal (described in `signal`(3)).
79[1]	`error (79): Process quit (SIGQUIT)` FOR$IOS_SIGQUIT. The process received the signal SIGQUIT. Core dump file created. Determine source of this quit signal (described in `signal`(3)).

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
95[1]	`info (95): Floating-point conversion failed` FOR$IOS_FLOCONFAI. The attempted unformatted read or write of nonnative floating-point data failed because the floating-point value: • Exceeded the allowable maximum value for the equivalent native format and was set equal to infinity (plus or minus) • Was infinity (plus or minus) and was set to infinity (plus or minus) • Was invalid and was set to not a number (NaN) Very small numbers are set to zero (0). This error could be caused by the specified nonnative floating-point format not matching the floating-point format found in the specified file. Check the following: • The correct file was specified. • The record layout matches the format Compaq Fortran is expecting. • The ranges for the data being used (Chapter 9) • The correct nonnative floating-point data format was specified (Chapter 10).
108	`severe (108): Cannot stat file` FOR$IOS_CANSTAFIL. Attempted stat operation on the indicated file failed. Make sure correct file and unit were specified.
120	`severe (120): Operation requires seek ability` FOR$IOS_OPEREQSEE. Attempted an operation on a file that requires the ability to perform seek operations on that file. Make sure the correct unit, directory path, and file were specified.
130[1] *(TU*X only)*	`severe (130): User breakpoint (SIGTRAP)` FOR$IOS_BRK_USERBP. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. Examine core dump for possible cause.
131[1] *(TU*X only)*	`severe (131): Kernel breakpoint (SIGTRAP)` FOR$IOS_BRK_KERNELBP. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. Examine core dump for possible cause.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
135[1] *(TU*X only)*	`severe (135): User single step (SIGTRAP)` FOR$IOS_BRK_SSTEPBP. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. Examine core dump for possible cause.
136[1] *(TU*X only)*	`severe (136): Overflow check (SIGTRAP)` FOR$IOS_BRK_OVERFLOW. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. The cause is an integer overflow. Try recompiling with the `-check overflow` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the core dump file to determine the source code in error.
137[1] *(TU*X only)*	`severe (137): Divide by zero check (SIGTRAP)` FOR$IOS_BRK_DIVZERO. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. Examine core dump file for possible cause.
138[1]	`severe (138): Array index out of bounds (SIGTRAP on TU*X, SIGILL on L*UX)` FOR$IOS_BRK_RANGE. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. The cause is an array subscript that is outside the dimensioned boundaries of that array. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the core dump file to determine the source code in error.
139[1]	`severe (139): Array index out of bounds for index` *nn* `(SIGTRAP on TU*X, SIGILL on L*UX)` FOR$IOS_BRK_RANGE2. Break exception generated a SIGTRAP signal (described in `signal`(3)). Core dump file created. The cause is an array subscript that is outside the dimensioned boundaries of the array index *n*. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the core dump file to determine the source code in error.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
140[1]	`severe (140): Floating inexact`
	FOR$IOS_FLTINE. A floating-point arithmetic or conversion operation gave a result that differs from the mathematically exact result. This trap is reported if the rounded result of an IEEE operation is not exact.
144[1]	`severe (144): reserved operand`
	FOR$IOS_ROPRAND. The Compaq Fortran RTL encountered a reserved operand. Please report the problem to Compaq.
145[1]	`severe (145): Assertion error`
	FOR$IOS_ASSERTERR. The Compaq Fortran RTL encountered an assertion error. Please report the problem to Compaq.
146[1]	`severe (146): Null pointer error`
	FOR$IOS_NULPTRERR. Attempted to use a pointer that does not contain an address. Modify the source program, recompile, and relink.
147[1]	`severe (147): stack overflow`
	FOR$IOS_STKOVF. The Compaq Fortran RTL encountered a stack overflow while executing your program.
148[1]	`severe (148): String length error`
	FOR$IOS_STRLENERR. During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range.
	Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the `core` file to determine the source code causing the error.
149[1]	`severe (149): Substring error`
	FOR$IOS_SUBSTRERR. An array subscript is outside the dimensioned boundaries of an array.
	Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the `core` file to determine the source code causing the error.
150[1]	`severe (150): Range error`
	FOR$IOS_RANGEERR. An integer value appears in a context where the value of the integer is outside the permissible range.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
151[1]	`severe (151): Allocatable array is already allocated` FOR$IOS_INVREALLOC. An allocatable array must not already be allocated when you attempt to allocate it. You must deallocate the array before it can again be allocated.
152[1]	`severe (152): Unresolved contention for Compaq Fortran RTL global resource` FOR$IOS_RESACQFAI. Failed to acquire a Compaq Fortran RTL global resource for a reentrant routine. For a multithreaded program, the requested global resource is held by a different thread in your program. For a program using asynchronous handlers, the requested global resource is held by the calling part of the program (such as main program) and your asynchronous handler attempted to acquire the same global resource.
153[1]	`severe (153): Allocatable array or pointer is not allocated` FOR$IOS_INVDEALLOC. A Fortran-90 allocatable array or pointer must already be allocated when you attempt to deallocate it. You must allocate the array or pointer before it can again be deallocated.
173[1]	`severe (173): A pointer passed to DEALLOCATE points to an array that cannot be deallocated` FOR$IOS_INVDEALLOC2. A pointer that was passed to DEALLOCATE pointed to an explicit array, an array slice, or some other type of memory that could not be deallocated in a DEALLOCATE statement. Only whole arrays previous allocated with an ALLOCATE statement can be validly passed to DEALLOCATE.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

174[1]	`severe (174): SIGSEGV,` *message-text*
	FOR$IOS_SIGSEGV. One of two possible messages occurs for this error number:
	• `severe (174): SIGSEGV, segmentation fault occurred`
	For both Compaq Tru64 UNIX and Linux systems, this message indicates that the program attempted an invalid memory reference. Check the program for possible errors.
	• `severe (174): SIGSEGV, possible program stack overflow occurred`
	On Compaq Tru64 UNIX systems, the following explanatory text also appears:
	`Program requirements exceed the maximum available stacksize resource limit. Contact your system administrator for help.`
	On Linux systems, the following explanatory text also appears:
	`Program requirements exceed current stacksize resource limit. Superusers may try increasing this resource by using the limit stacksize` *xxx* `command, where` *xxx* `is unlimited or something larger than your current limit. Other users should contact your system administrator for help.`

175[1]	`severe (175): DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8` FOR$IOS_SHORTDATEARG. The number of characters associated with the DATE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 8 characters in length. Verify that the TIME and ZONE arguments also meet their minimum lengths.
176[1]	`severe (176): TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10` FOR$IOS_SHORTTIMEARG. The number of characters associated with the TIME argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 10 characters in length. Verify that the DATE and ZONE arguments also meet their minimum lengths.

[1]Identifies errors not returned by IOSTAT.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
177[1]	`severe(177): ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5`
	FOR$IOS_SHORTZONEARG. The number of characters associated with the ZONE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 5 characters in length. Verify that the DATE and TIME arguments also meet their minimum lengths.
178[1]	`severe(178): Divide by zero`
	FOR$IOS_DIV. A floating-point or integer divide-by-zero exception occurred.
179[1,4]	`severe(179): Cannot allocate array---overflow on array size calculation`
	FOR$IOS_ARRSIZEOVF. An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory.
256	`severe (256): Unformatted I/O to unit open for formatted transfers`
	FOR$IOS_UNFIO_FMT. Attempted unformatted I/O to a unit where the OPEN statement (FORM specifier) indicated the file was formatted. Check that the correct unit (file) was specified.
	If the FORM specifier was not present in the OPEN statement and the file contains unformatted data, specify FORM='UNFORMATTED' in the OPEN statement. Otherwise, if appropriate, use formatted I/O (such as list-directed or namelist I/O).
257	`severe (257): Formatted I/O to unit open for unformatted transfers`
	FOR$IOS_FMTIO_UNF. Attempted formatted I/O (such as list-directed or namelist I/O) to a unit where the OPEN statement indicated the file was unformatted (FORM specifier). Check that the correct unit (file) was specified.
	If the FORM specifier was not present in the OPEN statement and the file contains formatted data, specify FORM='FORMATTED' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.
264	`severe (264): operation requires file to be on disk or tape`
	FOR$IOS_OPERREQDIS. Attempted to use a BACKSPACE statement on such devices as a terminal or pipe.
265	`severe (265): operation requires sequential file organization and access`
	FOR$IOS_OPEREQSEQ. Attempted to use a BACKSPACE statement on a file whose organization was not sequential or whose access was not sequential. A BACKSPACE statement can only be used for sequential files opened for sequential access.

[1]Identifies errors not returned by IOSTAT.

[4]Identifies errors that can be returned by STAT in an ALLOCATE statement.

(continued on next page)

Table 8–3 (Cont.) Run-Time Error Messages and Explanations

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
266[1]	`error (266): Fortran abort routine called` FOR$IOS_PROABOUSE. The program called `abort` to terminate the program. For more information on signals, see Section 8.3.
268[1]	`severe (268): End of record during read` FOR$IOS_ENDRECDUR. An end-of-record condition was encountered during execution of a nonadvancing I/O READ statement that did not specify the EOR branch specifier.
297[1]	`info (297):` *nn* `floating invalid traps` FOR$IOS_FLOINVEXC. The total number of floating-point invalid data traps encountered during program execution was *nn*. This message appears at program completion.
298[1]	`info (298):` *nn* `floating overflow traps` FOR$IOS_FLOOVFEXC. The total number of floating-point overflow traps encountered during program execution was *nn*. This message appears at program completion.
299[1]	`info (299):` *nn* `floating divide-by-zero traps` FOR$IOS_FLODIV0EXC. The total number of floating-point divide-by-zero traps encountered during program execution was *nn*. This message appears at program completion.
300[1]	`info (300):` *nn* `floating underflow traps` FOR$IOS_FLOUNDEXC. The total number of floating-point underflow traps encountered during program execution was *nn*. This message appears at program completion.

[1]Identifies errors not returned by IOSTAT.

For More Information:

- On locating exceptions within the debugger, see Section 4.9.
- On file operations, see Section 7.5, Section 7.6, and Section 7.7.
- On record operations, see Section 7.8.
- On `f90` and `fort` command-line options, see Chapter 3.
- On data types, see Chapter 9.

9
Data Types and Representation

This chapter contains the following topics:

- Section 9.1, Summary of Data Types and Characteristics
- Section 9.2, Integer Data Representations
- Section 9.3, Logical Data Representations
- Section 9.4, Native IEEE Floating-Point Representations and Exceptional Values
- Section 9.5, Character Representation
- Section 9.6, Hollerith Representation

Note

In figures in this chapter, the symbol `:A` specifies the address of the byte containing bit 0, which is the starting address of the represented data element.

Compaq Fortran expects numeric data to be in native little endian order, in which the least-significant, rightmost bit (bit 0) or byte has a lower address than the most-significant, leftmost bit (or byte).

For More Information:

- On using nonnative big endian and VAX floating-point formats, see Chapter 10.
- On using Compaq Fortran I/O, see Chapter 7.

9.1 Summary of Data Types and Characteristics

Table 9–1 lists the intrinsic data types provided by Compaq Fortran, the storage required, and valid numeric ranges.

Table 9–1 Compaq Fortran Intrinsic Data Types, Storage, and Numeric Ranges

Data Type	Bytes	Description
BYTE (INTEGER*1)	1 (8 bits)	A BYTE declaration is a signed integer data type equivalent to INTEGER*1 or INTEGER (KIND=1).
INTEGER	1, 2, 4, or 8	Signed integer whose size is controlled by a kind type parameter or, if a kind type parameter (or size specifier) is omitted, certain `f90` or `fort` command options (see Section 9.2.1).
INTEGER (KIND=1) INTEGER*1	1 (8 bits)	Signed integer value from –128 to 127 (–2**7 to 2**7–1). Unsigned values from 0 to 255 (2**8-1)[1].
INTEGER (KIND=2) INTEGER*2	2 (16 bits)	Signed integer value from –32,768 to 32,767 (–2**15 to 2**15–1). Unsigned values from 0 to 65535 (2**16-1)[1].
INTEGER (KIND=4) INTEGER*4	4 (32 bits)	Signed integer value from –2,147,483,648 to 2,147,483,647 (–2**31 to 2**31–1). Unsigned values from 0 to 4,294,967,295 (2**32-1)[1].
INTEGER (KIND=8) INTEGER*8	8 (64 bits)	Signed integer value from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (–2**63 to 2**63–1).
LOGICAL	1, 2, 4, or 8	Logical value whose size is controlled by a kind type parameter or, if a kind type parameter (or size specifier) is omitted, certain `f90` command options (see Section 9.3).
LOGICAL (KIND=1) LOGICAL*1	1 (8 bits)	Logical values .TRUE. or .FALSE.[2]
LOGICAL (KIND=2) LOGICAL*2	2 (16 bits)	Logical values .TRUE. or .FALSE.[2]
LOGICAL (KIND=4) LOGICAL*4	4 (32 bits)	Logical values .TRUE. or .FALSE.[2]
LOGICAL (KIND=8) LOGICAL*8	8 (64 bits)	Logical values .TRUE. or .FALSE.[2]

[1]This range is allowed for assignment to variables of this type, but the data type is treated as signed in arithmetic operations.

[2]Logical data type ranges correspond to their comparable integer data type ranges. For example, in LOGICAL(KIND=2) L, the range for L is the same as the range for INTEGER(KIND=2) integers.

(continued on next page)

Table 9–1 (Cont.) Compaq Fortran Intrinsic Data Types, Storage, and Numeric Ranges

Data Type	Bytes	Description
REAL	4, 8, or 16	Real floating-point numbers whose size is controlled by a kind type parameter or, if a kind type parameter (or size specifier) is omitted, certain `f90` command options (see Section 9.4.1).
REAL (KIND=4) REAL*4	4 (32 bits)	Single-precision real floating-point values in IEEE S_float format ranging from 1.17549435E–38 to 3.40282347E38. Values between 1.17549429E–38 and 1.40129846E–45 are denormalized[3]. You cannot write a constant for a denormalized number.
DOUBLE PRECISION REAL (KIND=8) REAL*8	8 (64 bits)	Double-precision real floating-point values in IEEE T_ float format ranging from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D–308 and 4.94065645841246544D–324 are denormalized[3]. You cannot write a constant for a denormalized number.
REAL (KIND=16) REAL*16	16 (128 bits)	Extended-precision real floating-point values in Compaq IEEE style X_float format ranging from 6.475175119438025110924438958227646552 4996Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
COMPLEX	8, 16, or 32	Complex floating-point numbers whose size is controlled by a kind type parameter or, if a kind type parameter (or size specifier) is omitted, the `f90` command options described in Section 9.4.1.
COMPLEX (KIND=4) COMPLEX*8	8 (64 bits)	Single-precision complex floating-point values in a pair of IEEE S_float format parts: real and imaginary. The real and imaginary parts range from 1.17549435E–38 to 3.40282347E38. Values between 1.17549429E–38 and 1.40129846E–45 are denormalized[3]. You cannot write a constant for a denormalized number.
DOUBLE COMPLEX COMPLEX (KIND=8) COMPLEX*16	16 (128 bits)	Double-precision complex floating-point values in a pair of IEEE T_float format parts: real and imaginary. The real and imaginary parts each range from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D–308 and 4.94065645841246544D–324 are denormalized[3]. You cannot write a constant for a denormalized number.

[3]For more information on floating-point underflow, see Section 3.44.

(continued on next page)

Table 9–1 (Cont.) Compaq Fortran Intrinsic Data Types, Storage, and Numeric Ranges

Data Type	Bytes	Description
COMPLEX (KIND=16) COMPLEX*32	32 (256 bits)	Extended-precision complex floating-point values in a pair of Compaq IEEE style X_float format parts: real and imaginary. The real and imaginary parts each range from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
CHARACTER	1 byte (8 bits) per character	Character data represented by character code convention. Character declarations can be in the form CHARACTER(LEN=*n*), CHARACTER(*n*), or CHARACTER*n, where n is the number of bytes or n can be (*) to indicate passed-length format.
HOLLERITH	1 byte (8 bits) per Hollerith character	Hollerith constants.

In addition to the intrinsic numeric data types, you can also define nondecimal (binary, octal, or hexadecimal) constants as explained in the *Compaq Fortran Language Reference Manual*.

9.2 Integer Data Representations

Integer data lengths can be one, two, four, or eight bytes in length.

Integer data is signed with the sign bit being 0 (zero) for positive numbers and 1 for negative numbers.

To improve performance, avoid using 2-byte or 1-byte integer declarations (see Chapter 5).

9.2.1 Integer Declarations and f90/fort Compiler Options

The default size used for an INTEGER data declaration without a kind parameter is INTEGER (KIND=4) (same as INTEGER*4), unless you do one of the following:

- Explicitly declare the length of an INTEGER by using a kind parameter, such as INTEGER (KIND=8). Compaq Fortran provides intrinsic INTEGER kinds of 1, 2, 4, and 8. Each INTEGER kind number corresponds to number of bytes used by that intrinsic representation.

  To obtain the kind of a variable, use the KIND intrinsic function. You can also use a size specifier, such as INTEGER*4, but be aware this is an extension to the Fortran 95/90 standards.

- Use the `f90` command `-i2`, `-integer_size` *nn*, or `-i8` options to control the size of all INTEGER declarations without a kind parameter (see Section 3.53).

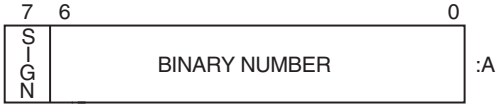
9.2.2 INTEGER (KIND=1) or INTEGER*1 Representation

Intrinsic INTEGER (KIND=1) or INTEGER*1 signed values range from –128 to 127 and are stored in a two's complement representation. For example:

```
+22 = 16(hex)
 -7 = F9(hex)
```

INTEGER (KIND=1) or INTEGER*1 values are stored in one byte, as shown in Figure 9–1.

Figure 9–1 INTEGER (KIND=1) or INTEGER*1 Representation



ZK–9814–GE

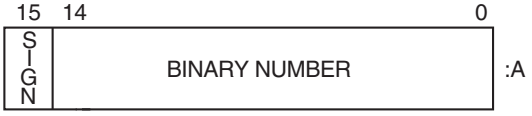
9.2.3 INTEGER (KIND=2) or INTEGER*2 Representation

Intrinsic INTEGER (KIND=2) or INTEGER*2 signed values range from –32,768 to 32,767 and are stored in a two's complement representation. For example:

```
+22 = 0016(hex)
 -7 = FFF9(hex)
```

INTEGER (KIND=2) or INTEGER*2 values are stored in two contiguous bytes, as shown in Figure 9–2.

Figure 9–2 INTEGER (KIND=2) or INTEGER*2 Representation

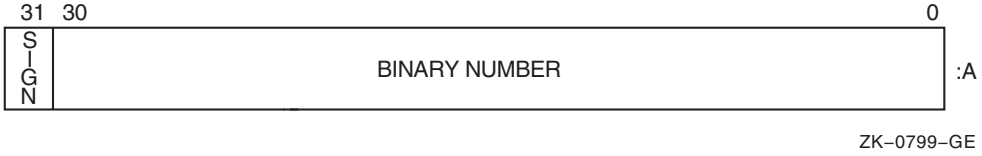


ZK–0798–GE

9.2.4 INTEGER (KIND=4) or INTEGER*4 Representation

Intrinsic INTEGER (KIND=4) or INTEGER*4 signed values range from –2,147,483,648 to 2,147,483,647 and are stored in a two's complement representation. INTEGER (KIND=4) or INTEGER*4 values are stored in four contiguous bytes, as shown in Figure 9–3.

Figure 9–3 INTEGER (KIND=4) or INTEGER*4 Representation

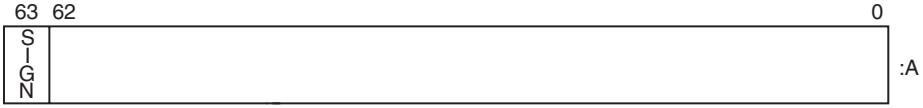


ZK–0799–GE

9.2.5 INTEGER (KIND=8) or INTEGER*8 Representation

Intrinsic INTEGER (KIND=8) or INTEGER*8 signed values range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and are stored in a two's complement representation. INTEGER*8 or INTEGER (KIND=8) values are stored in eight contiguous bytes, as shown in Figure 9–4.

Figure 9–4 INTEGER (KIND=8) or INTEGER*8 Representation



ZK–5299A–GE

For More Information:

- On defining constants and assigning values to variables, see the *Compaq Fortran Language Reference Manual*.
- On intrinsic functions related to the various data types, such as SELECTED_INT_KIND, see the *Compaq Fortran Language Reference Manual*.
- On the `f90` command options that control the size of default INTEGER declarations, see Section 3.53.

9.3 Logical Data Representations

Logical data can be one, two, four, or eight bytes in length.

The default size used for a LOGICAL data declaration without a kind parameter (or size specifier) is LOGICAL (KIND=4) (same as LOGICAL*4), unless you do one of the following:

- Explicitly declare the length of a LOGICAL declaration by using a kind parameter, such as LOGICAL (KIND=4). Compaq Fortran provides intrinsic LOGICAL kinds of 1, 2, 4, and 8. Each LOGICAL kind number corresponds to number of bytes used by that intrinsic representation.

  To obtain the kind of a variable, use the KIND intrinsic function. You can also use a size specifier, such as LOGICAL*4, but be aware this is an extension to the Fortran 95/90 standards.

- Use the `f90` command `-i2`, `-integer_size` *nn*, or `-i8` options to control the size of LOGICAL declarations without a kind parameter or size specifier (see Section 3.53).

To improve default performance, avoid using 2-byte or 1-byte logical declarations (see Chapter 5).

Intrinsic LOGICAL*1 or LOGICAL (KIND=1) values are stored in a single byte.

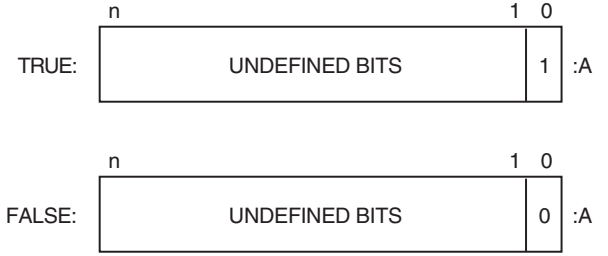
Logical (intrinsic) values can also be stored in the following sizes of contiguous bytes starting on an arbitrary byte boundary:

- Two bytes (LOGICAL (KIND=2) or LOGICAL*2)
- Four bytes (LOGICAL (KIND=4) or LOGICAL*4)
- Eight bytes (LOGICAL (KIND=8) or LOGICAL*8)

The low-order bit determines whether the logical value is true or false. Logical variables can also be interpreted as integer data (an extension to the Fortran 95/90 standards). For example, in addition to having logical values .TRUE. and .FALSE., LOGICAL*1 data can also have values in the range –128 to 127.

LOGICAL*1, LOGICAL*2, LOGICAL*4, and LOGICAL*8 data representations appear in Figure 9–5.

Figure 9–5 LOGICAL Representations



Key: n = 7, 15, 31, or 63 depending on LOGICAL declaration size

ZK–5300A–GE

For More Information:

- On defining constants and assigning values to variables, see the *Compaq Fortran Language Reference Manual*.
- On intrinsic functions related to the various data types, see the *Compaq Fortran Language Reference Manual*.
- On the `f90` command options that control the size of default LOGICAL declarations, see Section 3.53.

9.4 Native IEEE Floating-Point Representations and Exceptional Values

Floating-point numbers are stored on Compaq Tru64 UNIX systems in standard IEEE little endian floating-point notation, as follows:

- REAL (KIND=4) or REAL*4 declarations are stored in standard IEEE S_float little endian format.
- REAL (KIND=8) or REAL*8 declarations are stored in standard IEEE T_float little endian format.
- REAL (KIND=16) declarations are stored in Compaq IEEE style X_float binary little endian format.

COMPLEX numbers use a pair of little endian REAL values to denote the real and imaginary parts of the data, as follows:

- COMPLEX (KIND=4) or COMPLEX*8 declarations are stored in IEEE S_float format using two REAL (KIND=4) or REAL*4 values.

- COMPLEX (KIND=8) or COMPLEX*16 declarations are stored in IEEE T_float format using two REAL (KIND=8) or REAL*8 values.
- COMPLEX (KIND=16) or COMPLEX*32 declarations are stored in Compaq IEEE X_float format using two REAL (KIND=16) or REAL*16 values.

All floating-point formats represent fractions in sign-magnitude notation, with the binary radix point to the right of the most-significant bit. Fractions are assumed to be normalized, and therefore the most-significant bit is not stored. This is called **hidden bit normalization**. The hidden bit is assumed to be 1 unless the exponent is 0. If the exponent equals 0, then the value represented is **denormalized** (subnormal) or plus or minus 0 (zero).

For an explanation of the representation of NaN, Infinity, and related IEEE exceptional values on Alpha systems, see Section 9.4.8.

9.4.1 REAL and COMPLEX Declarations and f90/fort Compiler Options

The default sizes for REAL and COMPLEX data declarations are as follows:

- For REAL data declarations without a kind parameter (or size specifier), the default size is REAL (KIND=4) (same as REAL*4).
- For COMPLEX data declarations without a kind parameter (or size specifier), the default data size is COMPLEX (KIND=4) (same as COMPLEX*8).

To control the size of all REAL or COMPLEX declarations without a kind parameter, use the `f90` command `-real_size` *nn*, `-r8`, or `-r16` (see Section 3.78).

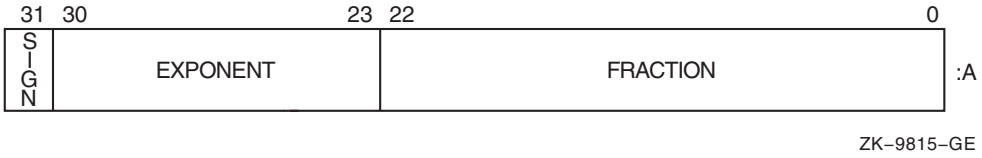
You can explicitly declare the length of a REAL or a COMPLEX declaration using a kind parameter or specify DOUBLE PRECISION or DOUBLE COMPLEX. To control the size of all DOUBLE PRECISION and DOUBLE COMPLEX declarations, use the `f90` command `-double_size` *nn* (see Section 3.78).

Intrinsic REAL kinds are 4 (single precision), 8 (double precision), and 16 (extended precision). Intrinsic COMPLEX kinds are also 4 (single precision), 8 (double precision), and 16 (extended precision), such as REAL (KIND=4) for single-precision floating-point data. To obtain the kind of a variable, use the KIND intrinsic function. You can also use a size specifier, such as REAL*4, but be aware this is an extension to the Fortran 95/90 standards.

9.4.2 REAL (KIND=4) or REAL*4 Representation

Intrinsic REAL (KIND=4) or REAL*4 (single precision REAL) data occupies four contiguous bytes stored in IEEE S_float format. Bits are labeled from the right, 0 through 31, as shown in Figure 9–6.

Figure 9–6 REAL (KIND=4) or REAL*4 Representation



ZK–9815–GE

The form of REAL (KIND=4) or REAL*4 data is sign magnitude, with:

- Bit 31 the sign bit (0 for positive numbers, 1 for negative numbers)
- Bits 30:23 a binary exponent in excess 127 notation
- Bits 22:0 a normalized 24-bit fraction including the redundant most-significant fraction bit not represented

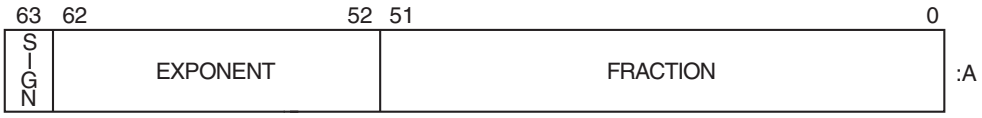
The value of data is in the approximate range: 1.17549435E–38 (normalized) to 3.40282347E38. The IEEE denormalized limit is 1.40129846E–45.

The precision is approximately one part in 2**23, typically seven decimal digits.

9.4.3 REAL (KIND=8) or REAL*8 Representation

Intrinsic REAL (KIND=8) or REAL*8 (same as DOUBLE PRECISION) data occupies eight contiguous bytes stored in IEEE T_float format. Bits are labeled from the right, 0 through 63, as shown in Figure 9–7.

Figure 9–7 REAL (KIND=8) or REAL*8 Representation



ZK–9816–GE

The form of REAL (KIND=8) or REAL*8 data is sign magnitude, with:

- Bit 63 the sign bit (0 for positive numbers, 1 for negative numbers)

- Bits 62:52 a binary exponent in excess 1023 notation
- Bits 51:0 a normalized 53-bit fraction including the redundant most-significant fraction bit not represented

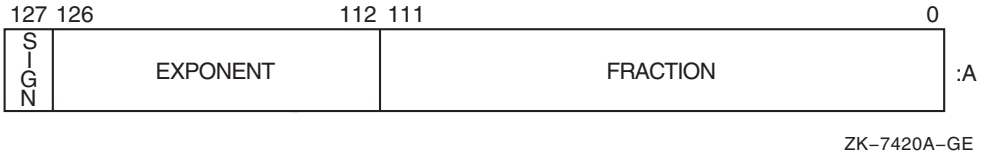
The value of data is in the approximate range: 2.2250738585072013D-308 (normalized) to 1.7976931348623158D308. The IEEE denormalized limit is 4.94065645841246544D-324.

The precision is approximately one part in 2**52, typically 15 decimal digits.

9.4.4 REAL (KIND=16) or REAL*16 Representation

Intrinsic REAL (KIND=16) or REAL*16 (extended precision) data occupies 16 contiguous bytes stored in Compaq IEEE style X_float format. Bits are labeled from the right, 0 through 127, as shown in Figure 9–8.

Figure 9–8 REAL (KIND=16) or REAL*16 Representation



The form of REAL*16 data is sign magnitude, with:

- Bit 127 the sign bit (0 for positive numbers, 1 for negative numbers)
- Bits 126:112 a binary exponent in excess 16383 notation
- Bits 111:0 a normalized 113-bit fraction including the redundant most-significant fraction bit not represented

The value of data is in the approximate range:
6.475175119438025110924438958227646552 4996Q-4966 to
1.189731495357231765085759326628007016196477Q4932.

Unlike other floating-point formats, there is little if any performance penalty from using denormalized extended-precision numbers. This is because accessing denormalized REAL (KIND=16) numbers does not result in an arithmetic trap (the extended-precision format is emulated in software). The smallest normalized number is 3.362103143112093506262677817321753Q-4932.

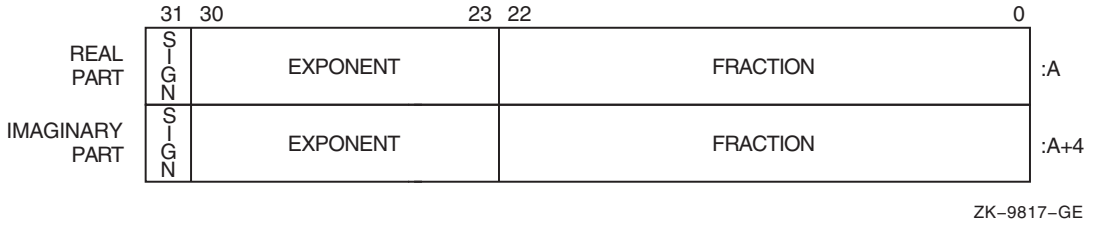
The precision is approximately one part in 2**112 or typically 33 decimal digits.

9.4.5 COMPLEX (KIND=4) or COMPLEX*8 Representation

Intrinsic COMPLEX (KIND=4) or COMPLEX*8 (single-precision COMPLEX) data is eight contiguous bytes containing a pair of REAL (KIND=4) or REAL*4 values stored in IEEE S_float format.

The low-order four bytes contain REAL (KIND=4) data that represents the real part of the complex number. The high-order four bytes contain REAL (KIND=4) data that represents the imaginary part of the complex number, as shown in Figure 9–9.

Figure 9–9 COMPLEX (KIND=4) or COMPLEX*8 Representation



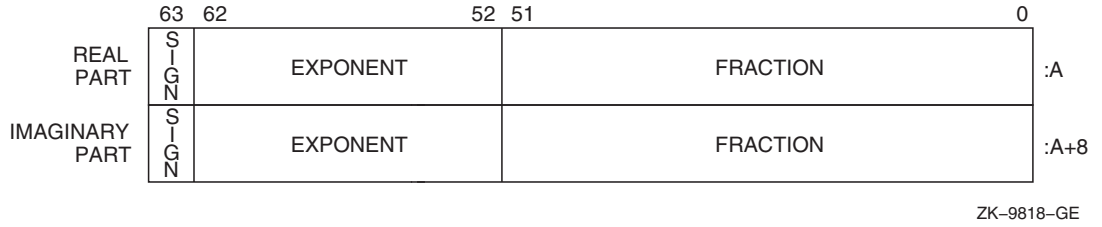
The limits and underflow characteristics for REAL (KIND=4) or REAL*4 apply to the two separate real and imaginary parts of a COMPLEX (KIND=4) or COMPLEX*8 number. Like REAL (KIND=4) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

9.4.6 COMPLEX (KIND=8) or COMPLEX*16 Representation

Intrinsic COMPLEX (KIND=8) or COMPLEX*16 (same as DOUBLE COMPLEX) data is 16 contiguous bytes containing a pair of REAL*8 values stored in IEEE T_float format.

The low-order eight bytes contain REAL (KIND=8) data that represents the real part of the complex data. The high-order eight bytes contain REAL (KIND=8) data that represents the imaginary part of the complex data, as shown in Figure 9–10.

Figure 9–10 COMPLEX (KIND=8) or COMPLEX*16 Representation



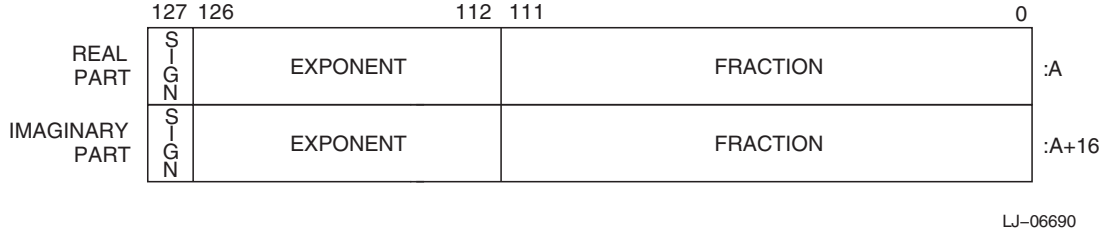
The limits and underflow characteristics for REAL (KIND=8) apply to the two separate real and imaginary parts of a COMPLEX (KIND=8) or COMPLEX*16 number. Like REAL (KIND=8) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

9.4.7 COMPLEX (KIND=16) or COMPLEX*32 Representation

Intrinsic COMPLEX (KIND=16) or COMPLEX*32 (extended precision) data is 32 contiguous bytes containing a pair of REAL*16 values stored in Compaq IEEE style X_float.

The low-order 16 bytes contain REAL (KIND=16) data that represents the real part of the complex data. The high-order 16 bytes contain REAL (KIND=16) data that represents the imaginary part of the complex data, as shown in Figure 9–11.

Figure 9–11 COMPLEX (KIND=16) or COMPLEX*32 Representation



The limits and underflow characteristics for REAL (KIND=16) apply to the two separate real and imaginary parts of a COMPLEX (KIND=16) or COMPLEX*32 number. Like REAL (KIND=16) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

For More Information:

- On converting unformatted data, see Chapter 10.
- On defining constants and assigning values to variables, see the *Compaq Fortran Language Reference Manual*.
- On intrinsic functions related to the various data types, such as SELECTED_REAL_KIND, see the *Compaq Fortran Language Reference Manual*.
- On VAX (OpenVMS) floating-point data types (provided for those converting OpenVMS data), see Section A.4.3.
- On the `f90` command options that control the size of REAL and COMPLEX declarations (without a kind parameter or size specifier), see Section 3.78.
- On the `f90` command options that control the size of DOUBLE PRECISION declarations, see Section 3.34.
- On IEEE binary floating-point, see ANSI/IEEE Standard 754-1985.

9.4.8 Exceptional Floating-Point Representations

Exceptional values usually result from a computation and include plus infinity, minus infinity, NaN, and denormalized numbers.

Floating-point numbers can be one of the following:

- **Finite number**—A floating-point number that represents a valid number (bit pattern) within the normalized ranges of a particular data type, including –*max* to –*min*, –zero, +zero, +*min* to +*max*.

  For any native IEEE floating-point data type, the values of *min* or *max* are listed in Section 9.4.2 (single precision), Section 9.4.3 (double precision), and Section 9.4.4 (extended precision).

  Special bit patterns that are *not* finite numbers represent exceptional values.

- **Infinity**—An IEEE floating-point bit pattern that represents plus or minus infinity. Compaq Fortran identifies infinity values with the letters "Infinity" or asterisks (******) in output statements (depends on field width) or certain hexadecimal values (fraction of 0 and exponent of all 1 values).

- **Not-a-Number (NaN)**—An IEEE floating-point bit pattern that represents something other than a number. Compaq Fortran identifies NaN values

with the letters “NaN” in output statements. A NaN can be a signaling NaN or a quiet NaN:

  - A quiet NaN might occur as a result of a calculation, such as 0./0. and has an exponent of all 1 values and initial fraction bit of 1.
  - A signaling NaN must be set intentionally (does not result from calculations) and has an exponent of all 1 values and initial fraction bit of 0 (with one or more other fraction bits of 1).

- **Denormal**—Identifies an IEEE floating-point bit pattern that represents a number whose value falls between zero and the smallest finite (normalized) number for that data type. The exponent field contains all zeros.

  For negative numbers, denormalized numbers range from the next representable value larger than minus zero to the representable value that is one bit less than the smallest finite (normalized) negative number. For positive numbers, denormalized numbers range from the next representable value larger than positive zero to the representable value that is one bit less than the smallest finite (normalized) positive number.

- **Zero**—Can be the value +0 (all zero bits, also called true zero) or -0 (all zero bits except the sign bit, such as `Z'8000000000000000'`).

A NaN or infinity value might result from a calculation that contains a divide by zero, overflow, or invalid data.

A denormalized number occurs when the result of a calculation falls within the denormalized range for that data type (subnormal value).

To control floating-point exception handling at run time for the main program, use the appropriate `-fpe`*n* option. The callable `for_set_fpe` routine allows further control for subprogram use or conditional use during program execution.

If an exceptional value is used in a calculation, an unrecoverable exception can occur unless you specify the appropriate `-fpe`*n* option or use the `for_set_fpe` routine. Denormalized numbers can be processed as is, set equal to zero with program continuation or a program stop, and generate warning messages (see Section 3.44).

Table 9–2 lists the hexadecimal (hex) values of the IEEE exceptional floating-point numbers for S_float (single precision), T_float (double precision), and X_float (extended precision) formats.

Table 9–2 Exceptional Floating-Point Numbers

Exceptional Number	Hex Value
S_float Representation	
Infinity (+)	Z'7F800000'
Infinity (–)	Z'FF800000'
Zero (+0)	Z'00000000'
Zero (–0)	Z'80000000'
Quiet NaN (+)	From Z'7FC00000' to Z'7FFFFFFF'
Quiet NaN (–)	From Z'FFC00000' to Z'FFFFFFFF'
Signaling NaN (+)	From Z'7F800001' to Z'7FBFFFFF'
Signaling NaN (–)	From Z'FF800001' to Z'FFBFFFFF'
T_float Representation	
Infinity (+)	Z'7FF0000000000000'
Infinity (–)	Z'FFF0000000000000'
Zero (+0)	Z'0000000000000000'
Zero (-0)	Z'8000000000000000'
Quiet NaN (+)	From Z'7FF8000000000000' to Z'7FFFFFFFFFFFFFFF'
Quiet NaN (–)	From Z'FFF8000000000000' to Z'FFFFFFFFFFFFFFFF'
Signaling NaN (+)	From Z'7FF0000000000001' to Z'7FF7FFFFFFFFFFFF'
Signaling NaN (–)	From Z'FFF0000000000001' to Z'FFF7FFFFFFFFFFFF'

(continued on next page)

Table 9–2 (Cont.) Exceptional Floating-Point Numbers

Exceptional Number	Hex Value
X_float Representation	
Infinity (+)	Z'7FFF0000000000000000000000000000'
Infinity (–)	Z'FFFF0000000000000000000000000000'
Zero (+0)	Z'00000000000000000000000000000000'
Zero (–0)	Z'80000000000000000000000000000000'
Quiet NaN (+)	From Z'7FFF8000000000000000000000000000' to Z'7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
Quiet NaN (–)	From Z'FFFF8000000000000000000000000000' to Z'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
Signaling NaN (+)	From Z'7FFF0000000000000000000000000001' to Z'7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF'
Signaling NaN (–)	From Z'FFFF0000000000000000000000000001' to Z'FFFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF'

Compaq Fortran supports IEEE exception handling, allowing you to test for infinity by using a comparison of floating-point data (such as generating positive infinity by using a calculation like x=1.0/0 and comparing x to the calculated number).

The appropriate `f90` command `-fpe`*n* options or calling the `for_set_fpe` routine with appropriate arguments allows program continuation when a calculation results in a divide by zero, overflow, or invalid data arithmetic exception, generating an exceptional value (a NaN or Infinity (+ or –)).

To test for a NaN when Compaq Fortran allows continuation for arithmetic exceptions, you can use the ISNAN intrinsic function.

For example, you might use the following code to test a DOUBLE PRECISION (REAL (KIND=8)) value:

```
      DOUBLE PRECISION A, B, F
      A = 0.
      B = 0.

!     Perform calculations with variables A and B
      .
      .
      .

!     f contains the value to check against a particular NaN

      F = A / B
```

```
      IF (ISNAN(F)) THEN
         WRITE (6,*) '--> Variable F contains a NaN value <--'
      ENDIF
!     Inform user that f has the hardware quiet NaN value
!     Perform calculations with variable F (or stop program early)
      END PROGRAM
```

This program might be compiled with `-fpe2` or `-fpe4` to allow:

- Continuation when a NaN (or other exceptional value) is encountered in a calculation
- A summary message explaining the number and types of arithmetic exceptions encountered:

```
% f90 -fpe2 isnan.for
% a.out
forrtl: error: floating invalid
 --> Variable F contains a NaN value <--
forrtl: info: 1 floating invalid traps
```

The FP_CLASS intrinsic function is also available to check for exceptional values (see the *Compaq Fortran Language Reference Manual* and the file `/usr/include/fordef.f`).

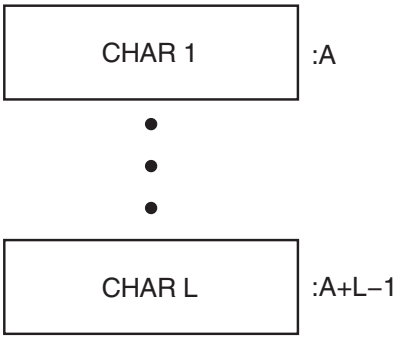
For More Information:

- On using the `f90` command `-fpe`*n* options and the `for_set_fpe` routine to control arithmetic exception handling, see Section 3.44 and Section 14.3.2 respectively.
- On exceptional values, see the *Alpha Architecture Reference Manual*.
- On IEEE binary floating-point exception handling, see the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985) and `ieee`(3).
- On Compaq Fortran floating-point exception handling, see Chapter 14.

9.5 Character Representation

A character string is a contiguous sequence of bytes in memory, as shown in Figure 9–12.

Figure 9–12 CHARACTER Data Representation



ZK–0809–GE

A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 1 through 65,535.

For More Information:

- On defining constants, assigning values to variables, using substring expressions, and concatenation, see the *Compaq Fortran Language Reference Manual*.
- On intrinsic functions related to the various data types, see the *Compaq Fortran Language Reference Manual*.

9.6 Hollerith Representation

Hollerith constants are stored internally, one character per byte. When Hollerith constants contain the ASCII representation of characters, they resemble the storage of character data (see Figure 9–12).

When Hollerith constants store numeric data, they usually have a length of one, two, four, or eight bytes and resemble the corresponding numeric data type.

For More Information:

- On defining constants and assigning values to variables, see the *Compaq Fortran Language Reference Manual*.
- On intrinsic functions related to the various data types, see the *Compaq Fortran Language Reference Manual*.

10

Converting Unformatted Numeric Data

This chapter contains the following topics:

- Section 10.1, Endian Order of Numeric Formats
- Section 10.2, Little Endian Floating-Point Format
- Section 10.3, Native and Supported Nonnative Numeric Formats
- Section 10.4, Limitations of Numeric Conversion
- Section 10.5, Methods of Specifying the Unformatted Numeric Format
- Section 10.6, Additional Information on Nonnative Data

10.1 Endian Order of Numeric Formats

Data storage uses a convention of either **little endian** or **big endian** storage, depending on the computer. The storage convention generally applies to numeric values that span multiple bytes.

Little endian storage occurs when:

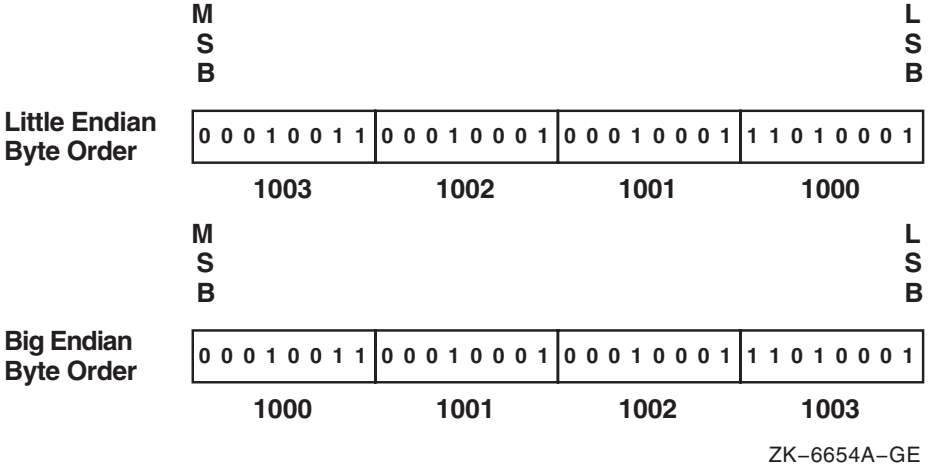
- The least significant bit (LSB) value is in the byte with the lowest address.
- The most significant bit (MSB) value is in the byte with the highest address.
- The address of the numeric value is the byte containing the LSB. Subsequent bytes with higher addresses contain more significant bits.

Big endian storage occurs when:

- The least significant bit (LSB) value is in the byte with the highest address.
- The most significant bit (MSB) value is in the byte with the lowest address.
- The address of the numeric value is the byte containing the MSB. Subsequent bytes with higher addresses contain less significant bits.

Figure 10–1 shows the difference between the two byte-ordering schemes.

Figure 10–1 Little Endian and Big Endian Storage of an INTEGER Value



Moving data files between little endian and big endian computers requires that the data be converted.

10.2 Little Endian Floating-Point Format

On Compaq Tru64 UNIX and Linux systems, Compaq Fortran supports the following little endian floating-point formats in memory:

Floating-Point Size	Format in Memory
KIND=4	IEEE S_float
KIND=8	IEEE T_float
KIND=16	Compaq IEEE style X_float

If your program needs to read or write unformatted data files containing a floating-point format that differs from the format in memory for that data size, you can request that the unformatted data be converted.

Converting unformatted data is generally faster than converting formatted data and is less likely to lose precision for floating-point numbers.

10.3 Native and Supported Nonnative Numeric Formats

Compaq Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big endian INTEGER or floating-point format.

When reading a nonnative unformatted format, the nonnative format on disk must be converted to native format in memory. Similarly, native data in memory can be written to a nonnative unformatted format. If a converted nonnative value is outside the range of the native data type, a run-time message appears (listed in Table 8–3).

Supported native and nonnative floating-point formats include:

- Standard IEEE little endian floating-point formats[1] and little endian integers. These formats are found on Compaq Tru64 UNIX Alpha systems, Compaq OpenVMS Alpha systems, Microsoft Windows NT systems, IBM-compatible PC systems, and Linux Alpha systems.

  On Tru64 UNIX and Linux systems, these are the native (in memory) floating-point and integer formats.

- Standard IEEE big endian floating-point formats[1] and big endian integers found on most Sun systems, most Hewlett-Packard systems (such as HP-UX systems), and IBM's RISC System/6000 systems.

- Compaq VAX little endian floating-point formats and little endian integers supported by Compaq Fortran for OpenVMS VAX systems and Compaq Fortran for OpenVMS Alpha systems.

- Big endian proprietary floating-point formats and big endian integers associated with CRAY (CRAY systems).

- Big endian proprietary floating-point formats and big endian integers associated with IBM (the IBM's System\370 and similar systems).

The native memory format uses little endian integers and little endian IEEE floating-point formats, as follows:

- INTEGER and LOGICAL declarations of one, two, four, or eight bytes (intrinsic kinds 1, 2, 4, and 8). You can specify the integer data length by using an explicit data declaration (kind parameter or size specifier). All INTEGER and LOGICAL declarations without a kind parameter or size specifier will be four bytes in length. To request an 8-byte size for all

[1] IEEE floating-point formats are defined in the IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

INTEGER and LOGICAL declarations without a kind parameter or size specifier, use an `f90` command-line option on Tru64 UNIX systems or and `fort` command-line option on Linux systems (see Section 9.2.1).

- IEEE S_float format for single-precision 4-byte REAL and 8-byte COMPLEX declarations (KIND=4). You can specify the real or complex data length by using an explicit data declaration (kind parameter or size specifier). For all REAL or COMPLEX declarations without a kind parameter or size specifier, this is the default size unless you use an `f90` command-line option to request double-precision sizes (see Section 9.4.1).
- IEEE T_float format for double-precision 8-byte REAL and 16-byte COMPLEX declarations (KIND=8). You can specify the real or complex data length by using an explicit data declaration (kind parameter or size specifier). To request double-precision sizes for all REAL or COMPLEX declarations without a kind parameter or size specifier, you can use an `f90` command-line option (see Section 9.4.1).
- Compaq IEEE style X_float format for extended-precision 16-byte REAL and 32-byte COMPLEX declarations (KIND=16). You can specify the real data length by using an explicit data declaration (kind parameter or size specifier). To request extended-precision sizes for all DOUBLE PRECISION or DOUBLE COMPLEX declarations, you can use an `f90` command-line option (see Section 9.4.1).

Table 10–1 lists the keywords for the supported unformatted file data formats. Use the appropriate keyword after the `-convert` option (such as `-convert cray`) or as an environment variable value (see Section 10.5.1 and Section 10.5.2).

Table 10–1 Unformatted Numeric Formats, Keywords, and Supported Data Types

Recognized Keyword[1]	Description
BIG_ENDIAN	Big endian integer data of the appropriate INTEGER size (one, two, four, or eight bytes) and big endian IEEE floating-point formats for REAL and COMPLEX single- and double- and extended-precision numbers. INTEGER (KIND=1) or INTEGER*1 data is the same for little endian and big endian.
CRAY	Big endian integer data of the appropriate INTEGER size (one, two, four, or eight bytes) and big endian CRAY proprietary floating-point format for REAL and COMPLEX single- and double-precision numbers.
FDX	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian Compaq proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
FGX	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian Compaq proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
IBM	Big endian integer data of the appropriate INTEGER size (one, two, or four bytes) and big endian IBM proprietary (System\370 and similar) floating-point format for REAL and COMPLEX single- and double-precision numbers.

[1]When using the data type as a `-convert` keyword option on the `f90` command line, the data type keyword must be in lowercase, such as `-convert big_endian`.

(continued on next page)

Table 10–1 (Cont.) Unformatted Numeric Formats, Keywords, and Supported Data Types

Recognized Keyword[1]	Description
LITTLE_ENDIAN	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following native little endian IEEE floating-point formats: • S_float for REAL (KIND=4) and COMPLEX (KIND=4) • T_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
NATIVE	No conversion occurs between memory and disk. This is the default for unformatted files.
VAXD	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX Compaq proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)
VAXG	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX Compaq proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)

[1]When using the data type as a `-convert` keyword option on the `f90` command line, the data type keyword must be in lowercase, such as `-convert big_endian`.

While this solution is not expected to fulfill all floating-point conversion needs, it provides the capability to read and write various types of unformatted nonnative floating-point data.

For More Information:

- On porting OpenVMS Fortran data files to a Tru64 UNIX Alpha or a Linux Alpha system for use by Compaq Fortran, see Section A.4.
- On ranges and the format of native IEEE floating-point data types, see Table 9–1 and Section 9.4.
- On ranges and the format of VAX floating-point data types, see Section A.4.3.
- On specifying the size of INTEGER declarations (without a kind) using an `f90` command-line option, see Section 9.2.1.
- On specifying the size of LOGICAL declarations (without a kind) using an `f90` command-line option, see Section 9.3.
- On specifying the size of REAL or COMPLEX declarations (without a kind) using an `f90` command-line option, see Section 9.4.1.
- On data declarations and other Compaq Fortran language information, see the *Compaq Fortran Language Reference Manual*.

10.4 Limitations of Numeric Conversion

The Compaq Fortran floating-point conversion solution is not expected to fulfill all floating-point conversion needs.

Data (variables) contained in derived types and record structures (specified in a STRUCTURE statement) are not converted. When the variables are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified.

If a program reads an I/O record containing multiple floating-point fields into an integer array (instead of their respective variables), the fields will not be converted. When the fields are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified.

With EQUIVALENCE statements, the data type of the variable named in the I/O statement is used.

10.5 Methods of Specifying the Unformatted Numeric Format

The five methods you can use to specify the type of nonnative (or native) format are as follows:

- Set an environment variable for a specific unit number before the file is opened. The environment variable is named `FORT_CONVERT`*n*, where *n* is the unit number.
- Set an environment variable for a specific file name extension before the file is opened. The environment variable is named `FORT_CONVERT.`*ext*, where *ext* is the file name extension (suffix).
- Add the CONVERT specifier to the OPEN statement for a specific unit number.
- Compile the program with an OPTIONS statement that specifies the /CONVERT=*keyword* qualifier. This method affects all unit numbers using unformatted data specified by the program.
- Compile the program with the appropriate command-line option `-convert` *keyword*. This method affects all unit numbers using unformatted data specified by the program.

If you specify more than one method, the order of precedence when you open a file with unformatted data is:

1. Check for a `FORT_CONVERT`*n* environment variable
2. Check for a `FORT_CONVERT.`*ext* environment variable
3. Check the OPEN statement CONVERT specifier
4. Check whether an OPTIONS statement with a /CONVERT=*keyword* qualifier was present when the program was compiled
5. Check whether the `f90 -convert` *keyword* option was used when the program was compiled

If none of these methods are specified, no conversion occurs between disk and memory. Data should therefore be in the native memory format (little endian integer and little endian IEEE format) or otherwise translated by the application program.

Any keyword listed in Table 10–1 can be used with any of these methods.

If you are uncertain about the format, you can do the following:

- Open the file (OPEN statement)

- Use an INQUIRE statement (by unit) that specifies the CONVERT specifier equated to a character variable
- Close the file (CLOSE statement)
- Open the file using the CONVERT variable value obtained from the INQUIRE statement

10.5.1 Environment Variable FORT_CONVERT*n* Method

You can use the environment variable method to specify multiple formats in a single program, usually one format for each unit number. You specify the numeric format at run time by setting the appropriate environment variable before you open that unit number. For example, to specify the numeric format for unit 9, set environment variable `FORT_CONVERT9` to the appropriate value (such as BIG_ENDIAN) before you run the program.

When you open the file, the environment variable is always used, since this method takes precedence over the `f90` command-line option methods. For instance, you might use this method to specify that different unformatted numeric formats for different unit numbers (perhaps in a script file that sets the environment variable before running the program).

For example, assume you have a previously compiled program that reads numeric data from unit 28 and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in native little endian format to unit 29.

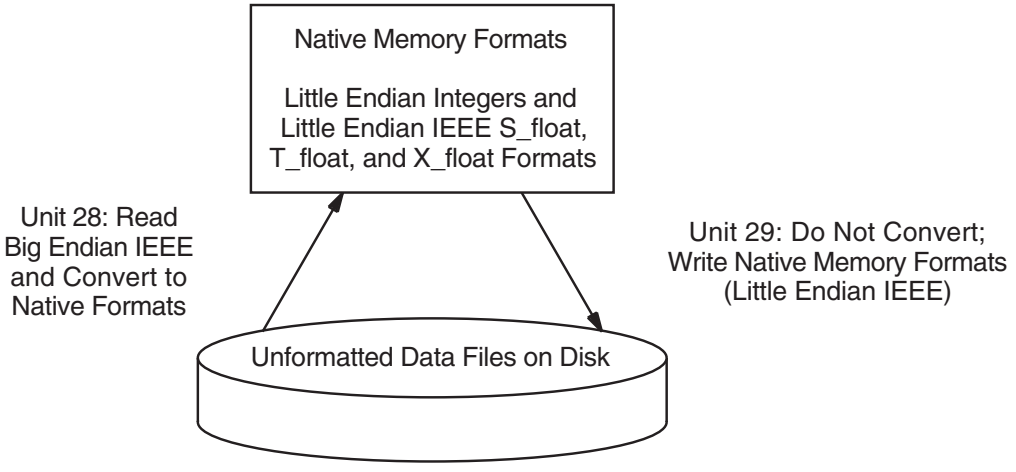
In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format (S_float, T_float, X_float) when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29.

Without requiring source code modification or recompilation of this program, the following C shell command sequence sets the appropriate environment variables before running the program (`/usr/userc/conv_ieee.out`):

```
% setenv FORT_CONVERT28 BIG_ENDIAN
% setenv FORT_CONVERT29 NATIVE
% /usr/userc/conv_ieee.out
```

Figure 10–2 shows the data formats used on disk and in memory when the example file `/usr/userc/conv_ieee.out` is run after the environment variables are set with shell commands.

Figure 10–2 Sample Unformatted File Conversion



ZK–6655A–GE

For More Information:

- On the shell commands you can use to set or unset environment variables, see Appendix B, Compaq Fortran Environment Variables.

10.5.2 Environment Variable FORT_CONVERT.*ext* Method

You can use this method to specify formats in a single program, usually one format for each specified file name extension (suffix). You specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit OPEN statement to one or more unformatted files.

For example, assume you have a previously compiled program that reads floating-point numeric data from one file and writes to another file using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from a file with a `.dat` file extension suffix and write that data in native little endian format to a file with a suffix of `.data`. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format when read from `file.dat`, and then written without conversion in native little endian IEEE format to the file with a suffix of `.data`, assuming that environment variables `FORT_CONVERT.DATA` and `FORT_CONVERT`*n* (for that unit number) are not defined.

Without requiring source code modification or recompilation of this program, the following command sequence sets the appropriate environment variables before running the program (/usr/userc/proj2/cvbigend.exe):

```
% setenv FORT_CONVERT.DAT BIG_ENDIAN
% /usr/userc/proj2/cvbigend.exe
```

The FORT_CONVERT*n* method takes precedence over this method. When the appropriate environment variable is set when you open the file, the FORT_CONVERT.*ext* environment variable is used if a FORT_CONVERT*n* environment variable is not set for the unit number.

The FORT_CONVERT*n* and FORT_CONVERT.*ext* environment variable methods take precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

The file name extension (suffix) is case-sensitive. The extension must be part of the file name (not the directory path).

10.5.3 OPEN Statement CONVERT='keyword' Method

You can use the OPEN statement method to specify multiple formats in a single program, usually one format for each specified unit number. This method requires an explicit file OPEN statement to specify the numeric format of the file for that unit number.

This method takes precedence over the OPTIONS statement or the -convert *keyword* method, but has a lower precedence than the environment variable methods.

The following source code shows an OPEN statement coded for unformatted VAXD numeric data (read from unit 15), and an OPEN statement coded for unformatted native little endian format (written to unit 20). The absence of the CONVERT specifier (in the second OPEN statement) or environment variable FORT_CONVERT20 indicates native little endian data for unit 20:

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
.
.
.
OPEN (FILE='graph3_ieee.dat', FORM='UNFORMATTED', UNIT=20)
```

A hard-coded OPEN statement CONVERT specifier keyword value cannot be changed after compile time. However, to allow selection of a particular format at run time, you can equate the CONVERT specifier to a variable and provide the user with a menu that allows selection of the appropriate format (menu choice sets the variable) before the OPEN occurs.

You can also select a particular format for a unit number at run time by using the environment variable method (see Section 10.5.1), which takes precedence over the OPEN statement CONVERT specifier method.

You can issue an INQUIRE statement (by unit number) to an opened file to obtain the current CONVERT option in use.

10.5.4 OPTIONS Statement /CONVERT=*keyword* Method

You can only specify one numeric file format for all unit numbers using this method, unless you also use one of the FORT_CONVERT environment variable methods or the OPEN statement CONVERT specifier method.

You specify the numeric format at compile time and must compile all routines under the same OPTIONS statement CONVERT=*keyword* qualifier. You can use one source program and compile it using different `f90` commands to create multiple executable programs that each read a certain format.

The environment variables and OPEN CONVERT specifier methods take precedence over this method. For instance, you might use the environment variables or OPEN CONVERT specifier method to specify each unit number that will use a format other than that specified using the `f90` command-line option method. This method takes precedence over the `f90` command `-convert` *keyword* option method.

You can use OPTIONS statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding `f90` command-line options. For example, to use VAX G_float (along with VAX F_float and VAX H_float) as the unformatted file format, specify the following OPTIONS statement:

```
OPTIONS /CONVERT=VAXG
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format using the OPTIONS statement method, unless you use it in combination with one of the environment variable methods or the OPEN statement CONVERT keyword method to specify a different format for a particular unit number.

For More Information:

- On the OPTIONS statement, see the *Compaq Fortran Language Reference Manual*.

10.5.5 Command-Line -convert *keyword* Option Method

You can specify only one numeric format for all unit numbers by using the command-line option method, unless you also use the environment variable method or CONVERT specifier method.

You specify the numeric format at compile time and must compile all routines under the same `-convert` *keyword* option. You can use one source program and compile it using different `f90` commands to create multiple executable programs that each read a certain format.

The other methods take precedence over this method. For instance, you might use the environment variables or OPEN CONVERT specifier method to specify each unit number that will use a format other than that specified using the `f90` command-line option method.

For example, the following shell commands compile program `file.f90` to use VAX D_float and F_float data. Data is converted between the file format and the little endian memory format (little endian integers and S_float, T_ float, and X_float little endian IEEE floating-point format). The created file, `vaxd_convert.out`, is then run:

```
% f90 -convert vaxd -o vaxd_convert.out file.f90
% vaxd_convert.out
```

Because this method affects all unit numbers, you cannot read or write data in different formats if you only use the `f90 -convert` *keyword* method. To specify a different format for a particular unit number, use the `f90 -convert` *keyword* method in combination with an environment variable method or the OPEN statement CONVERT specifier method.

10.6 Additional Information on Nonnative Data

The following information applies to porting nonnative data:

- When porting source code along with the unformatted data, vendors might use different units for specifying the record length (RECL specifier, see Section 7.4.4) of unformatted files. While formatted files are specified in units of characters (bytes), unformatted files are specified in longword units for Compaq Fortran and some other vendors. The Fortran 90 standard, in Section 9.3.4.5, states: "If the file is being connected for unformatted input/output, the length is measured in processor-dependent units."[1]

[1] American National Standard Fortran 90, ANSI X3.198-1991, and International Standards Organization standard ISO/IEC 1539:1991

- Certain vendors apply different OPEN statement defaults to determine the record type. The default record type (RECORDTYPE) with Compaq Fortran depends on the values for the ACCESS and FORM specifiers for the OPEN statement, as described in the *Compaq Fortran Language Reference Manual*.
- Certain vendors use a different identifier for the logical data types, such as hex FF instead of 01 to denote `true`.
- Source code being ported might be coded specifically for big endian use.

For More Information:

- On OPEN statement specifiers, see the *Compaq Fortran Language Reference Manual*.
- On Compaq Fortran file characteristics, see Section 7.4, Types of Files and File Characteristics.
- On Compaq Fortran record types, see Section 7.4.3, Record Types, Record Overhead, and Maximum Record Length and Section 7.10, Format of Compaq Fortran Record Types.

11

Procedure Data Interfaces and Mixed Language Programming

This chapter contains the following topics:

- Section 11.1, Compaq Fortran Procedures and Argument Passing
- Section 11.2, Using the cDEC$ ALIAS and cDEC$ ATTRIBUTES Directives
- Section 11.3, Calling Between Compaq Fortran and C

Note

For information about calling between Compaq Fortran 77 and Compaq Fortran, see Section A.5.

11.1 Compaq Fortran Procedures and Argument Passing

The bounds of the main program are usually defined by using PROGRAM and END or END PROGRAM statements. Within the main program, you can define entities related to calling a function or subroutine, including modules and interface blocks.

A function or subroutine is considered a **subprogram**. A subprogram can accept one or more data values passed from the calling routine; the values are called **arguments**.

There are two types of arguments:

- **Actual arguments** are specified in the subprogram call.
- **Dummy arguments** are variables within the function or subroutine that receive the values (from the actual arguments).

The following methods define the interface between procedures:

- Declare and name a function with a FUNCTION statement and terminate the function definition with an END FUNCTION statement. Set the value of the data to be returned to the calling routine by using the function name as a variable in an assignment statement (or by specifying RESULT in a FUNCTION statement).

  Reference a function by using its name in an expression.

- Declare and name a subroutine with a SUBROUTINE statement and terminate the subroutine definition with an END SUBROUTINE statement. No value is returned by a subroutine.

  Reference a subroutine by using its name in a CALL statement (or use a defined assignment statement).

- For an external subprogram, depending on the type of arguments or function return values, you may need to declare an explicit interface to the arguments and function return value by using an **interface block**.

  Declare and name an interface block with an INTERFACE statement and terminate the interface block definition with an END INTERFACE statement. The **interface body** that appears between these two statements consists of function or subroutine specification statements.

- You can make data, specifications, definitions, procedure interfaces, or procedures globally available to the appropriate parts of your program by using a module (use association).

  Declare a module with a MODULE statement and terminate the module definition with an END MODULE statement. Include the definitions and other information contained within the module in appropriate parts of your program with a USE statement. A module can contain interface blocks, function and subroutine declarations, data declarations, and other information.

For More Information:

- On the Compaq Fortran language, including statement functions and defined assignment statements not described in this manual, see the *Compaq Fortran Language Reference Manual*.

11.1.1 Explicit and Implicit Interfaces

An **explicit interface** occurs when the properties of the subprogram interface are *known* within the scope of the function or subroutine reference. For example, the function reference or CALL statement occurs at a point where the function or subroutine definition is known through host or use association. Intrinsic procedures also have an explicit interface.

An **implicit interface** occurs when the properties of the subprogram interface are *not known* within the scope of the function or subroutine reference. In this case, the procedure data interface is unknown to the compiler. For example, external routines (EXTERNAL statement) that have not been defined in an interface block have an implicit interface.

In most cases, you can use a procedure interface block to make an implicit interface an explicit one. An explicit interface provides the following advantages over an implicit interface:

- It provides better compile-time argument checking and fewer run-time errors.
- In some cases, it provides faster run-time performance.
- It provides ease of locating problems in source files since the features help to make the interface self-documenting.
- It allows use of some language features that require an explicit interface, such as array function return values.
- When passing certain types of arguments between Compaq Fortran and non-Fortran languages, an explicit interface may be needed. For example, detailed information about an assumed-shape array argument can be obtained from a Compaq Fortran array descriptor. An array descriptor is generated when an appropriate explicit interface is used for certain types of array arguments.

For More Information:

- See Section 11.1.7, Compaq Fortran Array Descriptor Format.

11.1.2 Types of Compaq Fortran Subprograms

There are three major types of subprograms:

- A subprogram might be local to a single program unit (known only within its host). Since the subprogram definition and all its references are contained within the same program unit, it is called an **internal subprogram**.

  An internal subprogram has an explicit interface.

- A subprogram needed in multiple program units should be placed within a module. To create a **module subprogram** within a module, add a CONTAINS statement followed by the subprogram code. A module subprogram can also contain internal subprograms.

  A module subprogram has an explicit interface in those program units that reference the module with a USE statement (unless it is declared PRIVATE).

- **External subprograms** are needed in multiple program units but cannot be placed in a module. This makes their procedure interface unknown in the program unit in which the reference occurs. Examples of external subprograms include general-purpose library routines in standard libraries and subprograms written in other languages, like C or Ada.

  Unless an external subprogram has an associated interface block, it has an implicit interface. To provide an explicit interface for an external subprogram, create a procedure interface block (see Section 11.1.3).

  For subprograms with no explicit interface, declare the subprogram name as external. You can do this by using the EXTERNAL statement within the program unit where the external subprogram reference occurs. This allows the linker to resolve the reference.

  An external subprogram must not contain PUBLIC or PRIVATE statements.

11.1.3 Using Procedure Interface Blocks

Procedure interface blocks allow you to specify an explicit interface for a subprogram as well as define generic procedure names. This section limits discussion to those interface blocks used to provide an explicit subprogram interface. For complete information on interface blocks, see the *Compaq Fortran Language Reference Manual*.

The components of a procedure interface block follow:

- Begin a procedure interface block with an INTERFACE statement. Unless you are defining a generic procedure name, user-defined operator, or user-defined assignment, only the word INTERFACE is needed.
- To provide the interface body, copy the procedure specification statements from the actual subprogram, including:
  - The FUNCTION or SUBROUTINE statements.

- The interface body. For a procedure interface block, this includes specification (declaration) statements for the dummy arguments and a function return value (omit data assignment, FORMAT, ENTRY, DATA, and related statements).

  The interface body can include USE statements to obtain definitions.

  In parallel HPF programs *(TU*X only)*, any DISTRIBUTE, ALIGN, and INHERIT (!HPF) directives for the dummy arguments must also be included.

- The END FUNCTION or END SUBROUTINE statements.

- Terminate the interface block with an END INTERFACE statement.
- To make the procedure interface block available to multiple program units, you can do one of the following:
  - Place the procedure interface block in a module. Reference the module with a USE statement in each program unit that references the subprogram (use association).
  - Place the procedure interface block in each program unit that references the subprogram.

For an example of a module that contains a procedure interface block, see Section 1.3.

11.1.4 Passing Arguments and Function Return Values

Compaq Fortran on Tru64 UNIX and Linux systems uses the same argument-passing conventions as Compaq Fortran 77 on Compaq Tru64 UNIX systems for non-pointer scalar variables and explicit-shape and assumed-size arrays.

When calling Compaq Fortran subprograms, be aware that Compaq Fortran expects to receive arguments the same way it passes them.

The main points about argument passing and function return values are as follows:

- Arguments are generally passed by reference: the address of the argument is passed.

  Unless explicitly specified otherwise (such as with the cDEC$ ATTRIBUTES directive or the %VAL built-in function), an argument contains the address of the data being passed, not the data itself.

  Assumed-shape arrays and deferred-shape arrays are passed by array descriptor.

Arguments omitted by adding an extra comma (,) are passed as a zero by immediate value. OPTIONAL arguments that are not passed are also handled this way.

- Function return data is usually passed by immediate value (function return contains a value). Certain types of data (such as array-valued functions) are passed by other means.

  The value being returned from a function call usually contains the actual data, not the address of the data.

- Character variables, explicit-shape character arrays, and assumed-size character arrays are passed as arguments by reference along with an extra argument, called a "hidden length" argument, that gets passed by value after all actual arguments.

  The hidden length is an integer value. If two character scalar variables are passed, the argument list contains two extra hidden length arguments that respectively contain the length of the passed character arguments. Dummy arguments for character data can use an assumed length. You can use the `-mixed_str_len_arg` option to place the hidden length directly after its corresponding character argument instead of sequentially at the end of the argument list. See Section 3.62.

  When passing character arguments to a C routine as strings, the character argument is not automatically null-terminated by the compiler. To null-terminate a string from Compaq Fortran, use the CHAR intrinsic function or the null character escape sequence (described in the *Compaq Fortran Language Reference Manual*).

- External names have a trailing underscore (_) appended to their name and are in lowercase. On Linux systems, if an external name has an underscore in it, then two underscores are appended. For example, `EXTERNAL foo_bar` becomes `EXTERNAL foo_bar__`.

  The Compaq Fortran compiler appends the underscore to any reference to an external procedure name, as well as Compaq Fortran procedure declarations in the object file. This is mostly a concern when the program uses routines in Compaq Fortran and C, as described in Section 11.3.2.

The arguments passed from a calling routine must match the dummy arguments declared in the called function or subroutine (or other procedure), as follows:

- Arguments are kept in the same position as they are specified by the user.

  The exception to same position placement is the use of argument keywords to associate dummy and actual arguments.

- Each corresponding argument or function return value must at least match in data type, kind, and rank, as follows:
  - The primary Compaq Fortran intrinsic data types are character, integer, logical, real, and complex.

    To convert data from one data type to another, use the appropriate intrinsic procedures described in the *Compaq Fortran Language Reference Manual*.

    Also, certain attributes of a data item may have to match. For example, if a dummy argument has the POINTER attribute, its corresponding actual argument must also have the POINTER attribute Section 11.1.6).
  - You can use the kind parameter to specify the length of each numeric intrinsic type, such as INTEGER (KIND=8). For character lengths, use the LEN specifier, perhaps with an assumed length for dummy character arguments (LEN=*).
  - The **rank** (as number of dimensions) of the actual argument is usually the same (or less than) the rank of the dummy argument, unless an assumed-size dummy array is used.

    When using an explicit interface, the rank of the actual argument must be the same as the rank of the dummy argument.

    For example, when passing a scalar actual argument to a scalar dummy argument (no more than one array element or a nonarray variable), the rank of both is 0.

    You can pass an actual array to a dummy array of the same rank or you can pass an array section. If you use an assumed-shape array, the extents of the dummy array argument are taken from the actual array argument.

    Other rules which apply to passing arrays and pointers are described in Section 11.1.5, Section 11.1.6, and the *Compaq Fortran Language Reference Manual*.
- The means by which the argument is passed and received (passing mechanism) must match.

  By default, Compaq Fortran arguments are passed by reference. (See Table 11–3 for information about passing arguments with the C property.)

  When calling functions or other routines that are intended to be called from another language (such as C), be aware that these languages may require data to be passed by other means, such as by value.

Most Compaq Fortran function return values are passed by value. Certain types of data (such as array-valued functions) are passed by other means.

In most cases, you can change the passing mechanism of actual arguments by using the following Compaq extensions:

- cDEC$ ATTRIBUTES directive (see Section 11.2.2)
- Built-in functions (see Section 11.1.8)

- To explicitly specify the procedure (argument or function return) interface, provide an explicit interface.

  You can use interface blocks and modules to specify INTENT and other attributes of arguments.

For More Information:

- On passing arguments, function return values, and the contents of registers on Compaq Tru64 UNIX systems, see the *Compaq Tru64 UNIX Calling Standard for Alpha Systems*.
- On intrinsic data types, see Chapter 9 and the *Compaq Fortran Language Reference Manual*.
- On intrinsic procedures and attributes available for array use, see the *Compaq Fortran Language Reference Manual*.
- On explicit interfaces and when they are required, see the *Compaq Fortran Language Reference Manual*.
- On a Compaq Fortran example program that uses an external subprogram and a module that contains a procedure interface block, see Example 1–3.

11.1.5 Passing Arrays as Arguments

Certain arguments or function return values require the use of an explicit interface, including assumed-shape dummy arguments, pointer dummy arguments, and function return values that are arrays. This is discussed in the *Compaq Fortran Language Reference Manual*.

When passing arrays as arguments, the rank and the extents (number of elements in a dimension) should agree, so the arrays have the same shape and are **conformable**. If you use an assumed-shape array, the rank is specified and extents of the dummy array argument are taken from the actual array argument.

If the rank and extent (shape) do not agree, the arrays are *not* conformable. The assignment of elements from the actual array to the noncomformable (assumed-size or explicit-shape) dummy array is done by using array element sequence association. Using array element sequence associations is discussed in the *Compaq Fortran Language Reference Manual*.

Certain combinations of actual and dummy array arguments are disallowed.

For More Information:

- On the types of arrays and passing array arguments, see the *Compaq Fortran Language Reference Manual*.
- On explicit interfaces and when they are required, see the *Compaq Fortran Language Reference Manual*.
- On array descriptors, see Section 11.1.7.

11.1.6 Passing Pointers as Arguments

Previous sections have discussed the case where the actual and dummy arguments have neither the POINTER attribute nor the TARGET attribute.

The argument passing rules of like type, kind, and rank (for conformable arrays) or array element sequence association (for noncomformable arrays) apply when:

- Both actual and dummy arguments have the POINTER attribute.
- Dummy arguments have the TARGET attribute.
- Both actual and dummy arguments have neither attribute.

You can specify an actual argument of type POINTER and a dummy argument of type POINTER. You must use an explicit interface that defines the dummy argument with the POINTER attribute in the code containing the actual argument. This ensures that the pointer is passed, rather than the array data itself.

However, if you specify an actual argument of type POINTER and do *not* specify an appropriate explicit interface (such as an interface block), it is passed as actual (target) data.

For More Information:

- On using pointers and pointer arguments, see the *Compaq Fortran Language Reference Manual*.

11.1.7 Compaq Fortran Array Descriptor Format

When using an explicit interface (by association or procedure interface block), Compaq Fortran will generate a descriptor for the following types of dummy argument data structures:

- Pointers to arrays (array pointers)
- Allocatable arrays
- Assumed-shape arrays

To allow calling between Compaq Fortran 77 and Compaq Fortran, certain data structure arguments also supported by Compaq Fortran 77 do not use a descriptor, even when an appropriate explicit interface is provided. For example, since explicit-shape and assumed-size arrays are supported by both Compaq Fortran 77 and Compaq Fortran, a descriptor is not used.

However, for cases where the called routine needs the information in the Compaq Fortran descriptor, declare the routine with an assumed-shape or pointer argument and an explicit interface.

The byte components of the Compaq Fortran descriptor follow:

- Byte 0 contains a count of the number of dimensions (rank).
- Byte 1 should always contain a 1.
- Byte 2 contains the data type of the result, as follows:
  - 1 for INTEGER (KIND=1)
  - 2 for INTEGER (KIND=2)
  - 3 for INTEGER (KIND=4)
  - 4 for INTEGER (KIND=8)
  - 5 for LOGICAL (KIND=1)
  - 6 for LOGICAL (KIND=2)
  - 7 for LOGICAL (KIND=4)
  - 8 for LOGICAL (KIND=8)
  - 9 for REAL (KIND=4)
  - 10 for REAL (KIND=8)
  - 11 for REAL (KIND=16)
  - 12 for COMPLEX (KIND=4) or COMPLEX*8

  - 13 for COMPLEX (KIND=8) or COMPLEX*16
  - 14 for CHARACTER
  - 15 for RECORD
  - 17 for COMPLEX (KIND=16) or COMPLEX*32
- Bytes 3 to 7 (inclusive) are reserved.
- Bytes 8 to 15 contain the element length for character data, in bytes.
- Bytes 16 to 23 contain the address of the first element of an array.
- Bytes 24 to 39 are reserved.
- The remaining bytes (40 to 207) contain information about each array dimension, as follows:
  - Bytes 40 to 63 contain dimension information for rank 1
  - Bytes 64 to 87 contain dimension information for rank 2
  - Bytes 88 to 111 contain dimension information for rank 3
  - Bytes 112 to 135 contain dimension information for rank 4
  - Bytes 136 to 159 contain dimension information for rank 5
  - Bytes 160 to 183 contain dimension information for rank 6
  - Bytes 184 to 207 contain dimension information for rank 7

Within the dimension information (24 bytes) for each rank:

- Bytes 0 to 7 contain the number of bytes between two successive elements in this dimension.
- Bytes 8 to 15 contain the upper bound.
- Bytes 16 to 23 contain the lower bound.

For example, consider the following declaration:

```
integer,target :: a(10,10)
integer,pointer :: p(:,:)
p => a(9:1:-2,1:9:3)
call f(p)
.
.
.
```

The descriptor for actual argument p would contain the following values:

- Byte 0 contains 2 (number of dimensions).

- Byte 1 contains 1 (always).
- Byte 2 contains 3 (data type of the result is the default INTEGER size, usually INTEGER (KIND=4)).
- Bytes 3 to 15 are reserved.
- Bytes 16 to 23 contain the address of the first element.
- Bytes 24 to 39 are reserved.
- Bytes 40 to 63 contain dimension information for rank 1, as follows:
  - Bytes 40 to 47 contain –8 (distance between elements)
  - Bytes 48 to 55 contain 5 (upper bound)
  - Bytes 56 to 63 contain 1 (lower bound)
- Bytes 64 to 87 contain dimension information for rank 2, as follows:
  - Bytes 64 to 71 contain 120 (distance between elements)
  - Bytes 72 to 80 contain 3 (upper bound)
  - Bytes 81 to 87 contain 1 (lower bound)
- Byte 87 is the last byte.

11.1.8 Argument-Passing Mechanisms and Built-In Functions

When a Compaq Fortran program needs to call a routine written in a different language (or in some cases a Fortran subprogram), there may be a need to use a form other the Compaq Fortran default passing mechanisms. For example, numeric arguments may need to be passed by immediate value instead of by reference.

To change the Compaq Fortran default mechanisms with the Compaq Fortran built-in functions, use the following functions:

- To change how an argument is passed (default passing mechanism), use the %VAL or %REF built-in functions for specific arguments.
- To compute the address of a storage element as an integer value, use the %LOC built-in function in any arithmetic expression.

The %VAL or %REF functions can only be used as unparenthesized arguments in actual argument lists.

Instead of the Compaq Fortran built-in functions, you can use the cDEC$ ATTRIBUTES directive to change the Compaq Fortran default passing mechanisms for either an entire call or for individual arguments. (See Section 11.2.2).

11.1.8.1 Passing Addresses — %LOC Function

The %LOC built-in function computes the address of a storage element as an INTEGER*8 (Alpha UNIX systems) value. You can then use this value in an arithmetic expression. It has the following form:

```
%LOC(arg)
```

The %LOC function is particularly useful for non-Fortran procedures that may require argument data structures containing the addresses of storage elements. In such cases, the data structures should be declared volatile to protect them from possible optimizations.

For More Information:

- On optimization and declaring volatile data, see Section 5.8.3.
- On the VOLATILE attribute, see the *Compaq Fortran Language Reference Manual*.

11.1.8.2 Passing Arguments by Immediate Value — %VAL Function

The %VAL function passes the argument list entry as a 64-bit immediate value on Compaq Tru64 UNIX and Linux systems. It has the following form:

```
%VAL(arg)
```

The argument-list entry generated by the compiler is the value of the argument (arg). Because argument-list entries are eight bytes long, the argument value must be an INTEGER (including INTEGER*8), LOGICAL (including LOGICAL*8), or REAL (REAL*4 and REAL*8) constant, variable, array element, or expression.

If a COMPLEX (KIND=4) or COMPLEX (KIND=8) argument is passed by value, two REAL arguments (one contains the real part; the other the imaginary part) are passed by immediate value. If a COMPLEX parameter to a routine is specified as received by value (or given the C attribute), two REAL parameters are received and stored in the real and imaginary parts of the COMPLEX parameter specified. COMPLEX*32 arguments cannot be passed by value.

If the value is a byte, word, or longword, it is sign-extended to eight bytes.

To produce a zero-extended value rather than a sign-extended value, use the ZEXT intrinsic function (see the *Compaq Fortran Language Reference Manual*).

11.1.8.3 Passing Arguments by Reference — %REF Function

The %REF function passes the argument by reference. It has the following form:

```
%REF(arg)
```

The argument-list entry the compiler generates will contain the address of the argument, (arg). The argument value can be a record name, a procedure name, or a numeric or character expression, array, character array section, or array element. In Compaq Fortran, passing by reference is the default mechanism for numeric values, so the %REF call is usually not needed. Passing character arrays with %REF caused the hidden length to be omitted.

11.1.8.4 Examples of Argument Passing Built-in Functions

The following examples show the use of the argument list built-in functions:

- The first constant is passed by reference. The second constant is passed by immediate value:

  ```
  CALL SUB(2,%VAL(2))
  ```

- The first character variable is passed by address and hidden length. The second character variable is passed by reference (no hidden length):

  ```
  CHARACTER(LEN=10) A,B
  CALL SUB(A,%REF(B))
  ```

- Both arrays are passed by reference:

  ```
  INTEGER IARY(20), JARY(20)
  CALL SUB(IARY,JARY)
  ```

For an example of passing integer data by value (using %VAL) and by reference (default) to a C function, see Section 11.3.7.

11.2 Using the cDEC$ ALIAS and cDEC$ ATTRIBUTES Directives

This section provides reference information about the following directives:

- The cDEC$ ALIAS (or !DEC$ ALIAS or *DEC$ ALIAS) directive lets you specify a name for an external subprogram that differs from the name used by the calling subprogram.
- The cDEC$ ATTRIBUTES (or !DEC$ ATTRIBUTES or *DEC$ ATTRIBUTES) directive lets you specify the properties for external data objects and procedures. This includes using C language rules, specifying how an argument is passed (passing mechanism), and specifying an alias for an external routine.

11.2.1 cDEC$ ALIAS directive

Use the cDEC$ ALIAS directive to specify that the external name of an external subprogram is different from the name by which the calling procedure references it.

The syntax is:

```
cDEC$ ALIAS internal-name, external-name
```

The *internal-name* is the name of the subprogram as used in the current program unit.

The *external-name* is either a quoted character constant (delimited by single quotation marks) or a symbolic name.

If *external-name* is a quoted character constant, the value of that constant is used as the external name for the specified internal name. The character constant is used as it appears, with no modifications for case or addition or removal of punctuation characters. The default for the Compaq Fortran compiler is to force the name into lowercase and append an underscore unless directed otherwise.

If *external-name* is a symbolic name, the symbolic name (in lowercase) is used as the external name for the specified internal name and an underscore is appended. Any other declaration of the specified symbolic name is ignored for the purposes of the ALIAS directive.

The Compaq Tru64 UNIX and Linux linker may have restrictions on what appears in an external name and whether external names are case-sensitive.

For example, assume the following program (free source form):

```
PROGRAM ALIAS_EXAMPLE
!DEC$ ALIAS ROUT1, 'ROUT1A'
!DEC$ ALIAS ROUT2, 'routine2_'
!DEC$ ALIAS ROUT3, rout3A
      CALL ROUT1
      CALL ROUT2
      CALL ROUT3
END PROGRAM ALIAS_EXAMPLE
```

The three calls are to external routines named ROUT1A, routine2_, and rout3A. Because rout3A is not in quotation marks (character constant), a trailing underscore is added (Compaq Fortran adds a trailing underscore to external names on UNIX systems unless directed otherwise) and the letter `A` becomes a lowercase `a`. For details about adding underscores on Linux systems, see Section 11.3.2.

This feature can be useful when porting code in which different routine naming conventions are in use. By adding or removing the cDEC$ ALIAS directive, you can specify an alternate routine name without recoding the application.

You can also use the DECORATE option with the ALIAS option so the external name specified in ALIAS has prefix and postfix decorations performed on it that are associated with the calling mechanism that is in effect.

The cDEC$ ALIAS and cDEC$ ATTRIBUTES ALIAS directives are synonymous.

11.2.2 cDEC$ ATTRIBUTES Directive

Use the cDEC$ ATTRIBUTES directive to specify properties for data objects and procedures. These properties let you specify how data is passed and the rules for invoking procedures. The cDEC$ ATTRIBUTES directive is intended to simplify mixed-language calls with Compaq Fortran routines written in C or Assembler. The STDCALL keyword is synonymous with the C keyword on Compaq Tru64 UNIX and Linux systems.

The cDEC$ ATTRIBUTES directive takes the following form:

```
cDEC$ ATTRIBUTES att [,att]... :: object [,object]...
```

In this form:

c
Is the letter or character (c, C, *, !) that introduces the directive (see the *Compaq Fortran Language Reference Manual*).

att
Is one of the keywords listed in the *Compaq Fortran Language Reference Manual*. For example, C, ALIAS, REFERENCE, or VALUE.

object
Is the name of a data object used as an argument or procedure. Only one object is allowed when using the C, STDCALL, or ALIAS properties.

The *Compaq Fortran Language Reference Manual* shows valid combinations of properties with the various types of objects.

The ATTRIBUTES options C, STDCALL, REFERENCE, VALUE, and VARYING all affect the calling convention of routines.

By default, Fortran passes all data by reference (except the hidden length argument of strings, which is a special case). If the C or STDCALL option is used, the default changes to passing almost all data by value except arrays.

In addition to the calling-convention options C and STDCALL, you can also specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference, regardless of the calling convention option. Arrays can only be passed by reference.

Table 11–1 summarizes the effect of the most common Fortran calling-convention directives.

Table 11–1 Calling Conventions for ATTRIBUTES Options

Arguments	Default	C or STDCALL	C or STDCALL with REFERENCE
Scalar	Reference	Value	Reference
Scalar [value]	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference
String	Reference, Len: End	String (1:1)	Reference, Len: End
String [value]	Error	String(1:1)	String(1:1)
String [reference]	Reference, No Len	Reference: No Len	Reference: No Len
Array	Reference	Reference	Reference
Array [value]	Error	Error	Error
Array [reference]	Reference	Reference	Reference
Derived type	Reference	Value, size dependent	Reference
Derived type [value]	Value, size dependent	Value, size dependent	Value, size dependent
Derived type [reference]	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor

The terms used in Table 11–1 mean the following:

Term	Description
[value]	Assigned to the VALUE property
[reference]	Assigned to the REFERENCE property
Value	The argument value is pushed on the stack. All values are padded to the next 8-byte boundary.
Reference	The 8-byte argument address is pushed on the stack.
Len: End	The length of the string is pushed (by value) on the stack after all of the other arguments.
No Len	The length of the string is not available to the called procedure.
String(1:1)	For string arguments, the first character is converted to INTEGER(KIND=8) as in ICHAR(string(1:1)) and pushed onto the stack by value.
Error	Produces a compiler error.
Descriptor	8-byte address of the array descriptor.
Size dependent	Derived-type arguments specified by value are passed as follows: • Arguments of 1 to 4 bytes are passed by value • Arguments of 5 to 8 bytes are passed by value in two registers • Arguments of more than 8 bytes provide value semantics by passing a temporary storage address by reference.

The properties are described in the following sections.

11.2.2.1 C Property

The C property provides a convenient way for code written in Compaq Fortran to interact with routines written in C.

When applied to a subprogram, the C property defines the subprogram as having a specific set of calling conventions.

Table 11–2 summarizes the differences between the calling conventions:

Table 11–2 C Property and External Names

Item	Fortran Default	C Property Specified
Trailing underscore added to procedure names	Yes	No
Case of external subprogram names	Lowercase, unless the ALIAS property is specified	Lowercase, unless the ALIAS property is specified
Argument passing	See Table 11–3	See Table 11–3

In addition to the case of external names and the trailing underscore, the C property affects how arguments are passed, as described in Table 11–3.

Table 11–3 C Property and Argument Passing

Argument Variable Type	Fortran Default	C Property Specified for Routine
Scalar (includes derived types)	Passed by reference	Passed by value
Scalar, with VALUE specified	Passed by value	Passed by value
Scalar, with REFERENCE specified	Passed by reference	Passed by reference
String	Passed by reference with hidden length	String (1:1) padded to integer length
String, with VALUE specified	Error	String (1:1) padded to integer length
String, with REFERENCE specified	Passed by reference with *no* hidden length	Passed by reference with *no* hidden length
Arrays, including pointers to arrays	Always passed by reference or descriptor	Always passed by reference or descriptor

If C is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran 95/90 conventions pass arguments by reference.

Character arguments are passed as follows:

- By Fortran default, hidden lengths are put at the end of the argument list.
- If C is specified without REFERENCE for the arguments, the first character of the string is passed by value (padded with zeros out to INTEGER*8 length).

- If C is specified with REFERENCE for the argument (or if only REFERENCE is specified), the string is passed with no length.

When the C property is specified, the case of the external name (EXTERNAL statement) is forced to lowercase, even if `-names as_is` or `-names uppercase` was specified during compilation. To allow a mixed case or uppercase name when the C property is specified, specify the ALIAS property for the same subprogram or external name.

Example 11–1 shows the Compaq Fortran code that calls the C function `pnst` (no underscore) by using the cDEC$ ATTRIBUTES C directive and C language passing conventions.

Example 11–1 Calling C Functions and Passing Integer Arguments

```
! Using !DEC$ ATTRIBUTES to pass argument to C. File: pass_int_cdec.f90

interface
   subroutine pnst(i)
     !DEC$ ATTRIBUTES C :: pnst
     integer i
   end subroutine
end interface

  integer :: i
  i = 99
  call pnst(i)             ! pass by value
  print *,"99==",i
end
```

Example 11–2 shows the C function called `pnst` (no underscore) that is called by the example program shown in Example 11–1.

Example 11–2 Calling C Functions and Passing Integer Arguments

```
/* get integer by value from Fortran. File: pass_int_cdec_c.c */

void pnst(int i) {
    printf("99==%d\n",i);
        i = 100;
}
```

The files (shown in Example 11–1 and Example 11–2) might be compiled, linked, and run as follows:

```
% cc -c  pass_int_cdec_c.c
% f90 -o pass_cdec pass_int_cdec.f90 pass_int_cdec_c.o
% pass_cdec
99==99
99==          99
```

11.2.2.2 ALIAS Property

You can specify the ALIAS property as cDEC$ ALIAS or as cDEC$ ATTRIBUTES ALIAS; they are equivalent. The ALIAS property allows you to specify that the external name of an external subprogram is different from the name by which the calling procedure references it (see Section 11.2.1).

When both ALIAS and C properties are used for a subprogram or external (EXTERNAL statement) name, the ALIAS property takes precedence over the C property. This allows you to specify case-sensitive names (the C attribute sets them to lowercase).

11.2.2.3 REFERENCE and VALUE Properties

The following cDEC$ ATTRIBUTES properties specify how a dummy argument is to be passed:

- REFERENCE specifies a dummy argument's memory location is to be passed, not the argument's value.
- VALUE specifies a dummy argument's value is to be passed, not the argument's memory location.

When a complex (KIND=4 or KIND=8) argument is passed by value, two floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value. Conversely, if a COMPLEX parameter to a routine is specified as received by value (or given the C attribute), two REAL parameters are received and stored in the real and imaginary parts of the COMPLEX parameter specified.

Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value; neither can REAL*16 and COMPLEX*32 data. When REFERENCE is specified for a character argument, the string is passed with no length.

VALUE is the default if the C property is specified in the subprogram definition.

Consider the following free-form example, which passes an integer by value:

```
interface
  subroutine foo (a)
   !DEC$ ATTRIBUTES C :: foo
      integer a
  end subroutine foo
end interface
```

This subroutine can be invoked from Compaq Fortran using the name foo (no underscore):

```
integer i
i = 1
call foo(i)

end program
```

This is the actual subroutine code:

```
subroutine foo (i)
 !DEC$ ATTRIBUTES C :: foo
    integer i
    i = i + 1
    .
    .
end subroutine foo
```

For More Information:

- On Compaq Fortran intrinsic data types, see Chapter 9.
- On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.
- On the C language, see your operating system documentation.
- On passing arguments, function return values, and the contents of registers on Compaq Tru64 UNIX systems, see the *Compaq Tru64 UNIX Calling Standard for Alpha Systems*.

11.2.2.4 EXTERN and VARYING Properties

The EXTERN property specifies that a variable is allocated in another source file. EXTERN can be used in global variable declarations, but it must not be applied to dummy arguments.

You must use EXTERN when accessing variables declared in other languages.

The VARYING directive allows a variable number of calling arguments. If VARYING is specified, the C property must also be specified.

When using the VARYING directive, either the first argument must be a number indicating how many arguments to process, or the last argument must be a special marker (such as −1) indicating it is the final argument. The sequence of the arguments, and types and kinds, must be compatible with the called procedure.

For More Information:

- See the *Compaq Fortran Language Reference Manual*.

11.3 Calling Between Compaq Fortran and C

Before creating a mixed-language program that contains procedures written in Compaq Fortran and C, you need to know how to:

- Compile and link the program
- Use the platform-specific conventions for procedure names on Compaq Tru64 UNIX systems
- Use equivalent data arguments passed between the two languages

On Linux Alpha systems, this section assumes the usage of the Compaq C compiler (`ccc` command).

11.3.1 Compiling and Linking Files

Use the `f90` or `fort` command (and not the `cc` or `ccc` command) to:

- Compile and link Compaq Fortran source files
- Link Compaq Fortran object files
- Link C object files with Compaq Fortran source or object files

The `f90` and `fort` commands pass the appropriate libraries to `ld`, including the Compaq Fortran libraries and `libc`.

You can use the `cc` and `ccc` commands with the `-c` option to compile C source files into object files.

For example, the following `f90` command compiles and links the Compaq Fortran calling main program `ex1.f90` and the called C function `uopen_.c`:

```
% f90 ex1.f90 uopen_.c
```

You can use the `cc` or, on Linux systems, the `ccc` command to compile the C program into an object file before the `f90` command:

```
% cc -c uopen_.c
% f90 ex1.f90 uopen_.o
```

The cc (ccc) and f90 (fort) commands:

1. Apply cpp to the uopen_.c file (done by cc and ccc)
2. Compile uopen_.c into the object file uopen_.o (done by cc and ccc)
3. Compile ex1.f90 into an object file (done by f90 and fort)
4. Link both resulting object files to create the file a.out file (done by ld)

When a C program calls a Compaq Fortran subprogram, specify the -nofor_main option on the f90 command line:

```
% cc -c  cmain.c
% f90 -nofor_main cmain.o fsub.f90
```

To view the preprocessor and compilers used and the libraries passed to ld, use the f90 command -v option.

For More Information:

- On the interaction of the f90 command with other components, including options passed to cc, see Section 2.2.
- On the commands used to compile a Compaq Fortran and C example program, see Section 11.3.8.

11.3.2 Procedures and External Names

When designing a program that will use Compaq Fortran and C, be aware of the following general rules and available Compaq Fortran capabilities:

- The ld linker only allows one main program. Declare either the Compaq Fortran or the C program, but not both, as the main program. When the C program is the main program, use the -nofor_main option on the f90 command line (see Section 11.3.1).

  In Compaq Fortran, you can declare a main program:

  - With the PROGRAM and END PROGRAM statements
  - With an END statement

  To create a Compaq Fortran subprogram, declare the subprogram with such statements as FUNCTION and END FUNCTION or SUBROUTINE and END SUBROUTINE.

  In C, you need to use a main( ) declaration for a main program. To create a C function (subprogram), declare the appropriate function name and omit the main( ) declaration.

- When declaring external names in C that will be called by Compaq Fortran, use lowercase.

Compaq Fortran makes external names lowercase by default, so you need to make the C function name definition and declarations with lowercase letters. Although Compaq Fortran is case-insensitive, the C compiler and `ld` linker both treat external names as case-sensitive.

- In Compaq Fortran, external names and corresponding declarations have a trailing underscore (`_`) appended to their name.

  Due to differences in the argument-passing mechanisms and the data types between C and Compaq Fortran, convention requires that a trailing underscore be appended to external names (including function declarations) in the C program. This avoids accidental calls across the two languages.

  The Compaq Fortran compiler, by default, appends the underscore to references to external procedure names as well as Compaq Fortran procedure declarations. Any problems with the naming convention are reported by the linker when it searches for external names in an object file.

  For a C program to call a Compaq Fortran subprogram, the calling C program routine must append an underscore (`_`) to the name of the Compaq Fortran function (or subroutine) (if using the Compaq Fortran defaults). For example, if a C program wants to call a Compaq Fortran function named `exponent`, the C source code must refer to it as `exponent_`.

  Similarly, for a Compaq Fortran program to call a C function, the C function must use lowercase letters and have a trailing underscore. By default, the Compaq Fortran compiler converts external names to lowercase and appends a trailing underscore character when calling C language routines from Compaq Fortran. For example, if a Compaq Fortran program calls a C function using the name `conarray` in the function reference, the C function needs to be declared as `conarray_`.

- You can consider using some of the Compaq Fortran facilities provided to simplify the Compaq Fortran and C language interface:

  - You can use the cDEC$ ALIAS and cDEC$ ATTRIBUTES directives to specify alternative names for routines and change default passing mechanisms (see Section 11.2).
  - You can consider using the `f90` option `-assume nounderscore` to prevent Compaq Fortran from appending an underscore to most external names (see Section 3.15).
  - Instead of directly calling Compaq Tru64 UNIX and Linux system and library routines, consider using the language interface “jacket” routines provided by Compaq Fortran (see Chapter 12).

- To perform I/O not supported by the Compaq Fortran run-time system, you can open a file with a C function and then use Compaq Fortran I/O statements to perform I/O (see Section 7.9). Indicate the name of the C function with the OPEN statement USEROPEN specifier, which causes the OPEN statement to call the C function to open the file.
- A C main program can obtain more control by calling the Compaq Fortran run-time initialization routine for_rtl_init_ and related Compaq Fortran library routines. (See Section 12.2, 3f Routines.)

11.3.3 Invoking a C Function from Compaq Fortran

You can use a function reference or a CALL statement to invoke a C function from a Compaq Fortran main or subprogram.

If a value will be returned, use a function reference:

C Function Declaration	Compaq Fortran Function Invocation

data-type calc_(*argument-list*)
{
...
};
	EXTERNAL CALC
	data-type :: CALC, *variable-name*
	...
	variable-name=CALC(*argument-list*)
	...

If no value is returned, use a void return value and a CALL statement:

C Function Declaration	Compaq Fortran Subroutine Invocation

void calc_(*argument-list*)
{
...
};
	EXTERNAL CALC
	...
	CALL CALC(*argument-list*)

11.3.4 Invoking a Compaq Fortran Function or Subroutine from C

A C main program or function can invoke a Compaq Fortran function or subroutine by using a function prototype declaration and invocation.

If a value is returned, use a FUNCTION declaration:

Compaq Fortran Declaration	C Invocation
FUNCTION CALC(*argument-list*) *data-type* :: CALC ... END FUNCTION CALC	extern *data-type* calc_(*argument-list*) *data-type variable-name*; *variable-name*=calc_(*argument-list*); ...

If no value is returned, use a SUBROUTINE declaration and a void return value:

Compaq Fortran Declaration	C Invocation
SUBROUTINE CALC(*argument-list*) ... END SUBROUTINE CALC	extern void calc_(*argument-list*) calc_(*argument-list*); ...

11.3.5 Equivalent Data Types for Function Return Values

Both C and Compaq Fortran pass most function return data by value, but equivalent data types must be used. Table 11–4 lists equivalent function declarations in Compaq Fortran and C. See Table 11–5 for a complete list of data declarations.

Table 11–4 Equivalent Function Declarations in C and Compaq Fortran

C Function Declaration	Compaq Fortran Function Declaration
float rfort_( )	function rfort( ) real (kind=4) :: rfort
double dfort_( )	function dfort( ) real (kind=8) :: dfort
int ifort_( )	function ifort( ) integer (kind=4) :: ifort

Because there are no corresponding data types in C, you should avoid calling Compaq Fortran functions of type REAL (KIND=16), COMPLEX, and DOUBLE COMPLEX, unless for complex data you pass a struct of two float (or double) C values (see Section 11.3.10).

The Compaq Fortran LOGICAL data types contain a zero if the value is false and a –1 if the value is true, which works with C conditional and if statements.

A character-valued Compaq Fortran function is equivalent to a C language routine with two extra initial arguments added by the Compaq Fortran compiler:

- Address of the result
- Length of the result

For More Information:

- On the language interface "jacket" routines provided by Compaq Fortran, see Chapter 12.
- On opening a file using a C USEROPEN function, see Section 7.9.
- On passing character arguments, see Section 11.3.8.
- On Compaq Fortran intrinsic data types, see Chapter 9.
- On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

11.3.6 Argument Association and Equivalent Data Types

Compaq Fortran follows the argument-passing rules described in Section 11.1.4. These rules include:

- Passing arguments by reference (address).
- Receiving arguments by reference (address).
- Compaq Tru64 UNIX convention of appending a trailing underscore to external names and passing character data by using an extra argument for the character length.
- Compaq Linux convention of appending two trailing underscores to external names.

11.3.6.1 Compaq Fortran Intrinsic Data Types

Compaq Fortran lets you specify the lengths of its intrinsic numeric data types with the following:

- The kind parameter, such as REAL (KIND=4). Intrinsic integer and logical kinds are 1, 2, 4, and 8. Intrinsic real and complex kinds are 4 (single-precision), 8 (double-precision), and 16.
- A default-length name without a kind parameter, such as REAL or INTEGER. Certain `f90` command options can change the default kind, as described in Section 3.53 (for INTEGER and LOGICAL declarations), Section 3.78 (for REAL and COMPLEX declarations), and Section 3.34 (for DOUBLE PRECISION declarations).
- The Compaq Fortran extension of appending a **n* size specifier to the default-length name, such as INTEGER*8.
- For double-precision real or complex data, the word DOUBLE followed by the default-length name without a kind parameter (specifically DOUBLE PRECISION and DOUBLE COMPLEX).

The following declarations of the integer A*n* are equivalent (unless you specified the appropriate `f90` command option to override the default integer size):

```
INTEGER (KIND=4) :: A1
INTEGER (4)      :: A2
INTEGER          :: A3
INTEGER*4        :: A4
```

Character data in Compaq Fortran is passed and received by address, using an extra hidden-length argument to contain the string length. Dummy character arguments can use assumed-length syntax for accepting character data of varying length.

Consider the following Compaq Fortran subroutine declaration:

```
SUBROUTINE H (C)
CHARACTER(LEN=*) C
```

The equivalent C function declaration is:

```
void h_(char *c, int len);
```

The Fortran subroutine can be called from C as follows:

```
.
.
.
char *chars[15];
h_(chars, 15);
```

For More Information:

- On Compaq Fortran intrinsic data types, see Chapter 9.
- On passing character data (example program), see Section 11.3.8.

11.3.6.2 Equivalent Compaq Fortran and C Data Types

The calling routine must pass the same number of arguments expected by the called routine. For each argument passed, the manner (mechanism) of passing the argument and the expected data type must match what is expected by the called routine. For instance, C usually passes data by value and Compaq Fortran typically passes argument data by reference (address and, when appropriate, length).

You must determine the appropriate data types in each language that are compatible. When you call a C routine from a Compaq Fortran main program, certain built-in functions may be useful to change the default passing mechanism, as discussed in Section 11.1.8.

If the calling routine cannot pass an argument to the called routine because of a language difference, you may need to rewrite the called routine. Another option is to create an interface jacket routine that handles the passing differences.

When a C program calls a Compaq Fortran subprogram, all arguments must be passed by reference because this is what the Compaq Fortran routine expects. To pass arguments by reference, the arguments must specify addresses rather than values. To pass constants or expressions, their contents must first be placed in variables; then the addresses of the variables are passed.

When you pass the address of the variable, the data types must correspond as shown in Table 11–5 for Compaq Tru64 UNIX systems.

Table 11–5 Compaq Fortran and C Data Types

Compaq Fortran Data Declaration	C Data Declaration
integer (kind=2) x	short int x;
integer (kind=4) x	int x;
integer (kind=8) x	long int x; __int64 x;
logical x	unsigned x;
real x	float x;
double precision x	double x;
real (kind=16) x	None[1]
complex (kind=4) x	struct { float real; float imag } x;
complex (kind=8) x	struct { double dreal; double dimag } x;
complex (kind=16) x	struct { long double dreal; long double dimag } x;[1]
character (len=5) x	char x[5]

[1]The equivalent C declaration is long double (may not support X_floating).

Be aware of the various sizes supported by Compaq Fortran for integer, logical, and real variables (see Chapter 9), and use the size consistent with that used in the C routine.

Compaq Fortran LOGICAL data types contain a zero if the value is false and a –1 if the value is true, which works with C language conditional and if statements.

When one of the arguments is character data, the Compaq Fortran compiler passes the address of the character data as an argument and adds the length of the character string to the end of the argument list (see Section 11.1.4).

When a C program calls a Compaq Fortran subprogram, the C program must explicitly specify these items in an argument list in the following order:

1. For a character function, the address of the character function result and the length of the character function result
2. For normal arguments, the addresses of arguments or functions
3. The lengths of any character string arguments, specified as integer variables (passed in the same order as the arguments)

For example, consider the following Compaq Fortran function declaration that returns character data:

```
character(len=8) function ch()
  ch = 'ABCDEFG' //CHAR(0)
  return
end
```

The following C code invokes the Compaq Fortran function as ch_, explicitly passing the address and length of the character data arguments, as follows:

```
char s[8]
ch_(&s[0],8);
```

Any character string passed by Compaq Fortran is *not* automatically null-terminated. To null-terminate a string from Compaq Fortran, use the CHAR intrinsic function (as shown in the previous example and described in the *Compaq Fortran Language Reference Manual*).

11.3.7 Example of Passing Integer Data to C Functions

Example 11–3 shows C code that declares the two functions `hln_` and `mgn_`. These functions display the arguments received. The C language function `hln_` expects the argument by value, whereas `mgn_` expects the argument by reference (address).

Example 11–3 C Functions Called by a Compaq Fortran Program

```
/* get integer by value from Fortran. File: pass_int_to_c.c */

void hln_(int i) {
    printf("99==%d\n",i);
        i = 100;
}

/* get integer by reference from Fortran */

void mgn_(int *i) {
    printf("99==%d\n",*i);
        *i = 101;
}
```

Example 11–4 shows the Compaq Fortran (main program) code that calls the two C functions `mgn_` and `hln_`.

Example 11–4 Calling C Functions and Passing Integer Arguments

```
! Using %REF and %VAL to pass argument to C. File: pass_int_to_cfuncs.f90
  integer :: i
  i = 99
  call hln(%VAL(i))     ! pass by value
  print *,"99==",i

  call mgn(%REF(i))     ! pass by reference - %REF is optional in this case
  print *,"101==",i
  i = 99
  call mgn(i)           ! pass by reference
  print *,"101==",i
```

The files (shown in Example 11–3 and Example 11–4) might be compiled, linked, and run as follows:

```
% cc -c  pass_int_to_c.c
% f90 -o pass_int_to_c pass_int_to_cfuncs.f90
pass_int_to_c.o
% pass_int_to_c
99==99
99==          99
99==99
101==         101
99==99
101==         101
```

11.3.8 Example of Passing Character Data Between Compaq Fortran and C

The following examples show a Compaq Fortran program that calls a C function that serves as an interface (jacket) routine to the setenv library routine (described in setenv(3)).

The Compaq Fortran program is named test_setenv.f.

The C program that contains the setenv_ interface function is named fort_setenv.c.

The Compaq Fortran program performs the following tasks:

1. Calls the getenv function (a Section 3f library routine) and displays the current value of the environment variable PRINTER (described in getenv(3f))
2. Calls the setenv_ interface function to set the new value for the environment variable PRINTER
3. Calls the getenv function again and displays the new value of the environment variable PRINTER

Example 11–5 shows the Compaq Fortran program test_setenv.f.

Example 11–5 Compaq Fortran Program Calling a C Function

```
!     test_setenv.f

      character(len=50) :: ename, evalue
      integer           :: overwrite, setenv, ret

!     Use 3f routine getenv to return PRINTER environment variable value

      call getenv('PRINTER',evalue)

!     Now look at current value

      write(6,*) 'Previous env. variable value of PRINTER is: ', evalue

!     Use setenv C function. Overwrite flag = non-zero means
!     overwrite any existing environment variable.
!
!     Returns -1 if there was an error in setting the environment variable

      ename  = 'PRINTER'
      evalue = 'lps40'
      overwrite = 1
      ret = setenv(ename,evalue,overwrite)
```

(continued on next page)

Example 11–5 (Cont.) Compaq Fortran Program Calling a C Function

```
      if (ret < 0) write (6,*) 'Error setting env. variable'

!     Now look at current value
      evalue = ' '
      call getenv('PRINTER',evalue)
      write(6,*)  'New env. variable value of PRINTER is: ', evalue

      end
```

Example 11–6 shows the C program fort_setenv.c.

Example 11–6 C Interface Function Called by Compaq Fortran

```
/*  fort_setenv.c */
#include <stdlib.h>

int setenv_(char *ename,char *evalue,int *overwrite,int ilen1,int ilen2)
{
     int setenv(), lnblnk_(), i1, i2, ow, rc;
     char *p1, *p2;

     /* Get string length of each input parameter */
     i1 = lnblnk_(ename,ilen1);
     i2 = lnblnk_(evalue,ilen2);

     /* Allocate temporary storage */
     p1 = malloc((unsigned) i1+1);
     if( p1 == NULL ) return(-1);
     p2 = malloc((unsigned) i2+1);
     if( p2 == NULL ) {
        free(p1);
        return(-1);
     }

     /* Copy strings, and NUL terminate */
     strncpy(p1,ename,i1);
     p1[i1] = '\0';
     strncpy(p2,evalue,i2);
     p2[i2] = '\0';

     ow = *overwrite;

     /* Call the setenv library routine to set the environment variable */
```

(continued on next page)

Example 11–6 (Cont.) C Interface Function Called by Compaq Fortran

```
    rc = setenv(p1, p2, ow);
    free(p1);
    free(p2);
    return(rc);
}
```

The setenv_ function (shown in Example 11–6) sets the environment variable PRINTER to the value lps40 (passed as arguments from the Compaq Fortran calling program) by calling the setenv library routine with the overwrite flag set to 1.

The C function (shown in Example 11–6) uses the passed length to find the last nonblank character and uses the string up to that character. The C variables ilen1 and ilen2 receive the hidden length of the character strings ename and evalue respectively. The C function allocates storage, including an extra byte for the null-terminator to each string. The extra byte is used as part of an argument when calling the setenv library routine.

The following Compaq Fortran code in Example 11–5 calls the setenv_ C function:

```
ret=setenv(ename,evalue,overwrite)
```

This function invocation passes the following arguments to the C function setenv_:

Compaq Fortran Variable	Purpose	Data Type	How Passed
ename	Environment variable name	character	by reference, *not* null-terminated
evalue	Environment variable string	character	by reference, *not* null-terminated
overwrite	Overwrite flag for setenv	integer	by reference
not declared	Hidden length of ename	integer	by value
not declared	Hidden length of evalue	integer	by value

Data passed from Compaq Fortran to C is passed by reference. When passing character data from Compaq Fortran to C, the following rules apply:

- The Compaq Fortran character argument is *not* automatically terminated by a null string. To null-terminate a string, use the CHAR intrinsic function (described in the *Compaq Fortran Language Reference Manual*).
- An additional (hidden) argument that contains the string length is passed by value.

The C function (in Example 11–6) is declared as `setenv_` and accepts the Compaq Fortran arguments with the following function declaration:

```
int setenv_(ename,evalue,overwrite,ilen1,ilen2)
char *ename, *evalue;
int  *overwrite;
int  ilen1, ilen2;
```

The status returned from the C function to the calling Compaq Fortran program is an integer passed by value. (The C function obtains this value from `setenv` library routine.)

To create the executable program, the files might be compiled with the following commands:

```
% cc -c fort_setenv.c
% f90 test_setenv.f fort_setenv.o
```

When executed, `a.out` displays:

```
% a.out
 Previous env. variable value of PRINTER is:
  lpr
 New env. variable value of PRINTER is:
  lps40
```

11.3.9 Example of Passing Complex Data to C Functions

Example 11–7 shows Compaq Fortran code that passes a COMPLEX (KIND=4) value (1.0,0.0) by immediate value to subroutine `foo`. To pass COMPLEX arguments by value, the compiler passes the real and imaginary parts of the argument as two REAL arguments by immediate value.

Example 11–7 Calling C Functions and Passing Complex Arguments

```
! Using !DEC$ATTRIBUTES to pass COMPLEX argument by value to F90 or C.
! File: cv_main.f90

interface
   subroutine foo(cplx)
     !DEC$ATTRIBUTES C :: foo
        complex cplx
   end subroutine
end interface

  complex(kind=4) c
  c = (1.0,0.0)
  call foo(c)             ! pass by value

end
```

If subroutine foo were written in Compaq Fortran, it might look similar to the following example. In this version of subroutine foo, the COMPLEX parameter is received by immediate value. To accomplish this, the compiler accepts two REAL parameters by immediate value and stores them into the real and imaginary parts, respectively, of the COMPLEX parameter cplx.

```
! File: cv_sub.f90

   subroutine foo(cplx)
     !DEC$ATTRIBUTES C :: foo
     complex cplx

     print *, 'The value of the complex number is ', cplx

   end subroutine
```

If subroutine foo were written in C, it might look similar to the following example in which the complex number is explicitly specified as two arguments of type float:

```
/* File: cv_sub.c */

#include <stdio.h>

typedef struct {float c1; float c2;} complex;

void foo(complex c)
{
    printf("The value of the complex number is (%f,%f)\n", c.c1, c.c2);
}
```

The main routine (shown in Example 11–7) might be compiled and linked to the object file created by the compilation of the Compaq Fortran subroutine and then run as follows:

```
% f90 -o cv cv_main.f90, cv_sub.f90
% cv
The value of the complex number is (1.000000,0.0000000E+00)
```

The main routine might also be compiled and linked to the object file created by the compilation of the C subroutine and then run as follows:

```
% cc -c cv_sub.c
% f90 -o cv2 cv_main.f90 cv_sub.f90
% cv2
The value of the complex number is (1.000000,0.000000)
```

11.3.10 Handling User-Defined Structures

User-defined derived types in Compaq Fortran and user-defined C structures can be passed as arguments if the following conditions are met:

- The elements of the structures use the same alignment conventions (same amount of padding bytes, if any). The default alignment for C structure members is natural alignment. You can use the `cc -member_alignment` option or pragma to alter that alignment.

  Derived-type data in Compaq Fortran is naturally aligned (the compiler adds needed padding bytes) unless you specify the `-align norecords` option (see Section 3.3).

- All elements of the structures are in the same order.

  Compaq Fortran orders elements of derived types sequentially. However, those writing standard-conforming programs should not rely on this sequential order because the standard allows elements to be in any order unless the SEQUENCE statement is specified.

- The respective elements of the structures have the same data type and length (kind), as described in Section 11.3.6.

- The structure must be passed by reference (address).

11.3.11 Handling Scalar Pointer Data

When Compaq Fortran passes scalar numeric data *with* the pointer attribute, how the scalar numeric data gets passed depends on whether or not an interface block is provided:

- If you do *not* provide an interface block to pass the actual pointer, Compaq Fortran dereferences the Compaq Fortran pointer and passes the actual data (the target of the pointer) by reference.
- If you *do* provide an interface block to pass the actual pointer, Compaq Fortran passes the Compaq Fortran pointer by reference.

When passing scalar numeric data *without* the pointer attribute, Compaq Fortran passes the actual data by reference. If the called C function declares the dummy argument for the passed data to be passed by a pointer, it accepts the actual data passed by reference (address) and handles it correctly.

Similarly, when passing scalar data from a C program to a Compaq Fortran subprogram, the C program can use pointers to pass numeric data by reference.

Example 11–8 shows a Compaq Fortran program that passes a scalar (nonarray) pointer to a C function. Variable x is a pointer to variable y.

The function call to `ifunc1_` uses a procedure interface block, whereas the function call to `ifunc2_` does not. Because `ifunc1_` uses a procedure interface block (explicit interface) and the argument is given the pointer attribute, the pointer is passed. Without an explicit interface (`ifunc2_`), the target data is passed.

Example 11–8 Calling C Functions and Passing Pointer Arguments

```
! Pass scalar pointer argument to C. File: scalar_pointer.f90

interface
  function ifunc1(a)
  integer, pointer :: a
  integer ifunc1
  end function
end interface

integer, pointer :: x
integer, target :: y
```

(continued on next page)

Example 11–8 (Cont.) Calling C Functions and Passing Pointer Arguments

```
y = 88
x => y
print *,ifunc1(x)           ! interface block visible, so pass
                            ! pointer by reference. C expects "int **"

print *,ifunc2(x)           ! no interface block visible, so pass
                            ! value of "x" by reference. C expects "int *"
print *,y
end
```

Example 11–9 shows the C function declarations that receive the Compaq Fortran pointer or target arguments from the example in Example 11–8.

Example 11–9 C Functions Receiving Pointer Arguments

```
/* C functions Fortran pointers. File: scalar_pointer.c  */

int ifunc1_(int **a) {
    printf("a=%d\n",**a);
    **a = 99;
    return 100;
    }

int ifunc2_(int *a) {
    printf("a=%d\n",*a);
    *a = 77;
    return 101;
}
```

The files (shown in Example 11–8 and Example 11–9) might be compiled, linked, and run as follows:

```
% cc -c  scalar_pointer.c
% f90 -o scalar_pointer scalar_pointer.f90 scalar_pointer.o
% scalar_pointer
a=88
         100
a=99
         101
          77
```

11.3.12 Handling Arrays

There are two major differences between the way the C and Compaq Fortran languages handle arrays:

- Compaq Fortran stores arrays with the leftmost subscript varying the fastest (column-major order). With C, the rightmost subscript varies the fastest (row-major order).
- Although the default for the lower bound of an array in Compaq Fortran is 1, you can specify an explicit lower bound of 0 (zero) or another value. With C the lower bound is 0.

Because of these two factors:

- When a C routine uses an array passed by a Compaq Fortran subprogram, the dimensions of the array and the subscripts must be interchanged and also adjusted for the lower bound of 0 instead of 1 (or a different value).
- When a Compaq Fortran program uses an array passed by a C routine, the dimensions of the array and the subscripts must be interchanged. You also need to adjust for the lower bound being 0 instead of 1, by specifying the lower bound for the Compaq Fortran array as 0.

Compaq Fortran orders arrays in column-major order. The following Compaq Fortran array declaration for a 2 by 3 array creates elements ordered as y(1,1), y(2,1), y(1,2), y(2,2), y(1,3), y(2,3):

```
integer y(2,3)
```

The Compaq Fortran declaration for a 2 by 3 array can be modified as follows to have the lowest bound 0 and not 1, resulting in elements ordered as y(0,0), y(1,0), y(0,1), y(1,1), y(0,2), y(1,2):

```
integer y(0:1,0:2)
```

The following C array declaration for a 3 by 2 array has elements in row-major order as z[0,0], z[0,1], z[1,0], z[1,1], z[2,0], z[2,1]:

```
int z[3][2]
```

To use C and Compaq Fortran array data:

- Consider using a 0 (zero) as the lower bounds in the Compaq Fortran array declaration.

  You may need to have the Compaq Fortran declaration with a lower bound 0 (not 1) for maintenance with C arrays or because of algorithm requirements.

- Reverse the dimensions in one of the array declaration statements. For example, declare a Compaq Fortran array as 2 by 3 and the C array as 3 by 2. Similarly, when passing array row and column locations between C and Compaq Fortran reverse the dimension numbers (interchange the row and column numbers in a two-dimensional array).

When passing certain array arguments, if you use an explicit interface that specifies the dummy argument as an array with the POINTER attribute or an assumed-shape array, the argument is passed by array descriptor (see Section 11.1.7).

For information about performance when using multidimensional arrays, see Section 5.5.

Example 11–10 shows a C function declaration for function `expshape_`, which prints the passed explicit-shape array.

Example 11–10 C Function That Receives an Explicit-Shape Array

```
/* Get explicit-shape arrays from Fortran  */

void expshape_(int  x[3][2]) {
    int i,j;
  for (i=0;i<3;i++)
     for (j=0;j<2;j++) printf("x[%d][%d]=%d\n",i,j,x[i][j]);
}
```

Example 11–11 shows a Compaq Fortran program that calls the C function `expshape_` (shown in Example 11–10).

Example 11–11 Compaq Fortran Program That Passes an Explicit-Shape Array

```
! Pass an explicit-shape array from Fortran to C.

integer :: x(2,3)
x = reshape( (/(i,i=1,6)/), (/2,3/) )

call expshape(x)
end
```

The files (shown in Example 11–10 and Example 11–11) might be compiled, linked, and run as follows:

```
% cc -c  exparray.c
% f90 -o exparray exparray.f90 exparray.o
% exparray
x[0][0]=1
x[0][1]=2
x[1][0]=3
x[1][1]=4
x[2][0]=5
x[2][1]=6
```

For information on the use of array arguments with Compaq Fortran, see Section 11.1.5.

11.3.13 Handling Common Blocks of Data

The following notes apply to handling common blocks of data between Compaq Fortran and C:

- In Compaq Fortran, you declare each common block with the COMMON statement. In C, you can use any global variable defined as a struct, but that global variable (an external name) must end with an underscore.
- Data types must match unless you desire implicit equivalencing. If so, you must adhere to the alignment restrictions for Compaq Fortran data types.
- If there are multiple routines that declare data with multiple COMMON statements and the common blocks are of unequal length, the largest of the sizes is used to allocate space.
- A blank common block has a name of `_BLNK__`.
- You should specify the same alignment characteristics in C and Compaq Fortran. The default alignment for C structure members is natural alignment. You can use the `cc -member_alignment` option or pragma to alter that alignment.

  To specify the alignment of common block data items, specify the `-align dcommons` or `-align commons` option when compiling Compaq Fortran procedures using the `f90` command or specify data declarations carefully (see Section 5.4).

The following examples show how C and Compaq Fortran code can access common blocks of data. The C code declares a global structure, calls the `f_calc_` Compaq Fortran function to set the values, and prints the values:

```
struct S {int j; float k;}r_;
main() {
f_calc_();
printf("%d %f\n", r_.j, r_.k);
}
```

The Compaq Fortran function then sets the data values:

```
subroutine f_calc()
common /r/j,k
real k
integer j
j = 356
k = 5.9
return
end
```

The C program then prints the structure member values 356 and 5.9 set by the Compaq Fortran function.

The previous example applies to Compaq Tru64 UNIX systems. On Compaq Linux systems, the external names with one underscore would have two trailing underscores.

11.4 Calling Between Parallel HPF and Non-Parallel HPF Code

When calling between parallel HPF and non-parallel HPF code *(TU*X only)*, the `-hpf` and `-nohpf_main` compile-time options are required in certain cases and prohibited in other cases.

12

Compaq Fortran Library Routines

This chapter contains the following topics:

- Section 12.1, Overview of Compaq Fortran Library Routines
- Section 12.2, 3f Routines
- Section 12.3, 3hpf Routines
- Section 12.4, Reference Pages for the 3f and 3hpf Routines
- Section 12.5, EXTERNAL or INTRINSIC Declarations
- Section 12.6, Example Using the 3f Library Routine shcom_connect
- Section 12.7, Example of the 3f Library Routines irand and qsort

12.1 Overview of Compaq Fortran Library Routines

Compaq Fortran library routines consist of two groups of routines, commonly referred to by their Tru64 UNIX or Linux reference page section:

- 3f routines (see Section 12.2, 3f Routines)
- 3hpf routines (see Section 12.3, 3hpf Routines)

12.2 3f Routines

The **3f routines** consist of two groups:

- Routines that serve as an interface or "jacket" to Tru64 UNIX or Linux operating system calls and library routines

  System calls (section 2 in reference pages) and library routines (section 3 in reference pages) provided with the operating system are typically written for use by the C language. The language interface 3f routines handle the various argument passing and function invocation differences between Compaq Fortran and C. This simplifies the programming effort needed to call a system call or C language library routine directly.

- Routines that perform special functions related to the Compaq Fortran run-time library (RTL) or calling between Compaq Fortran and C

Table 12–1 lists the groups of language interface ("jacket") 3f library routines.

Table 12–1 Summary of Language Interface ("Jacket") 3f Library Routines

Category	Routine Names	Standard-Conforming Alternatives
Bessel mathematical operations	besj0, besj1, besjn, bessel, besy0, besy1, besyn, dbesj0, dbesj1, dbesjn, dbesy0, dbesy1, dbesyn	None.
Bit manipulation	and, bit, lshift, not, or, rshift, xor	Consider using the Compaq Fortran intrinsics with the same name instead.
Directories and files	access, chdir, chmod, fstat, flush, fsync, isatty, link, lstat, rename, stat, symlnk, ttynam, umask, unlink	None.
Error handling	gerror, ierrno, perror	Use error-handling specifiers to handle Compaq Fortran errors, such as ERR and IOSTAT. Use these routines to handle Tru64 UNIX and Linux errors.
I/O	fgetc, fputc, fseek, ftell, getc, putc	Consider using Compaq Fortran nonadvancing I/O instead of fgetc, fputc, getc, putc.
Miscellaneous	index, len, lnblnk, loc, long, qsort, short, system	Instead of index and len, use the Compaq Fortran intrinsic functions INDEX and LEN.
Random numbers	drandm, irand, irandm, rand, random, srand	Consider using the Compaq Fortran intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED.

(continued on next page)

Table 12–1 (Cont.) Summary of Language Interface ("Jacket") 3f Library Routines

Category	Routine Names	Standard-Conforming Alternatives
Return date and time	ctime, dtime, etime, fdate, gmtime, idate, itime, ltime, time	Consider using the Compaq Fortran intrinsic subroutine DATE_AND_TIME or, if you need a subset of the information returned by DATE_AND_ TIME, the intrinsic subroutines (Compaq extensions) DATE, IDATE, and TIME.
Return error function	erf, derf, erfc, derfc	None.
Return process, system, or command-line information	getarg, getcwd, getenv, getgid, getlog, getpid, getuid, iargc	None.
Signals and processes	abort, alarm, fork, kill, signal, sleep, wait	Instead of abort, consider using the STOP statement.
Virtual memory allocation	falloc, free, malloc	For arrays and pointers, consider using the standard Fortran 95/90 ALLOCATABLE attribute or the ALLOCATE and DEALLOCATE statements.

Table 12–2 describes the 3f routines that provide special functions allowing Compaq Fortran and C language programs to work together.

Table 12–2 Summary of 3f Library Routines Providing Special Functions

Routine Name	Function and Comments
for_rtl_init_	Allows a C main language program to use the Compaq Fortran run-time library (RTL) environment by initializing the environment, including associated signal handlers; see for_rtl_init_ in Table 12–3.
for_rtl_finish_	Allows a C main language program to terminate use of the Compaq Fortran run-time library (RTL) environment; see for_rtl_finish_ in Table 12–3.
for_get_fpe	Returns information on the floating-point exception handling established for the current program unit; see for_get_fpe in Table 12–3.
for_set_fpe	Sets the floating-point exception handling established for the current program unit; see for_set_fpe in Table 12–3.
for_set_reentrancy	Sets reentrancy protection for the Fortran RTL.
getfd	Returns the file descriptor associated with a unit number, after the Compaq Fortran run-time library (RTL) environment has opened the file; see getfd in Table 12–3.
omp_* *(TU*X only)*	Various OpenMP Fortran API run-time routines related to parallel processing. See omp_* in Table 12–3 and Section D.1, OpenMP Fortran API Run-Time Library Routines.
ots* *(TU*X only)*	Various Compaq Fortran run-time routines related to parallel processing. See ots* in Table 12–3 and Section D.2, Other Parallel Threads Routines.
shcom_connect *(TU*X only)*	Allows multiple processes to access common block data in a shared library (uses memory mapping). See Section 12.6, Example Using the 3f Library Routine shcom_connect.

Table 12–3 describes each Compaq Fortran 3f library routine and lists the appropriate reference page.

For those 3f library routines that serve as interface routines to a system call or a different library routine, the required related routine is listed. Most routines are invoked as functions, with the exception of those noted in the table as subroutines.

Table 12–3 3f Functions and Subroutines

Name	Reference Page	Description
abort	abort(3f)	Terminates the program abnormally and may cause a core dump. Use as a subroutine.
access	access(3f)	Determines the accessibility of a file.
alarm	alarm(3f)	Executes a subroutine after a specified time.
and	bit(3f)	Returns the bitwise AND of two operands. Use as an intrinsic function.
besj0	bessel(3f)	Returns single-precision (REAL*4) bessel function value (first kind, zero order).
besj1	bessel(3f)	Returns single-precision (REAL*4) bessel function value (first kind, first order).
besjn	bessel(3f)	Returns single-precision (REAL*4) bessel function value (first kind, *n*th order).
bessel	bessel(3f)	Returns bessel functions.
besy0	bessel(3f)	Returns single-precision (REAL*4) bessel function value (second kind, zero order).
besy1	bessel(3f)	Returns single-precision (REAL*4) bessel function value (second kind, first order).
besyn	bessel(3f)	Returns single-precision (REAL*4) bessel function value (second kind, *n*th order).
bit	bit(3f)	Returns bitwise functions.
chdir	chdir(3f)	Changes the default directory.
chmod	chmod(3f)	Changes the mode of a file.
ctime	time(3f)	Returns the system time as a 24-character ASCII string.
dbesj0	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (first kind, zero order).

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
dbesj1	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (first kind, first order).
dbesjn	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (first kind, *n*th order).
dbesy0	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (second kind, zero order).
dbesy1	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (second kind, first order).
dbesyn	bessel(3f)	Returns a double-precision (REAL*8) bessel function value (second kind, *n*th order).
derf	erf(3f)	Returns a double-precision error function.
derfc	erf(3f)	Returns a double-precision error function (complementary form).
dffrac	flmin(3f)	Returns the fractional accuracy of a double-precision floating-point (REAL*8) number.
dflmax	flmin(3f)	Returns maximum positive double-precision floating-point (REAL*8) value.
dflmin	flmin(3f)	Returns minimum positive double-precision floating-point (REAL*8) value.
dlgamma	lgamma(3f)	Returns the REAL*8 log of the gamma function.
drand	rand(3f)	Generates a random number. Use drandm instead.
drandm	random(3f)	Generates a double-precision (REAL*8) random number.
dtime	etime(3f)	Returns the elapsed (delta) execution time.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
erf	erf(3f)	Returns a single-precision error function.
erfc	erf(3f)	Returns a single-precision error function (complementary form).
etime	etime(3f)	Returns the actual execution time of a process.
falloc	malloc(3f)	Allocates space for an array in virtual memory. Use with malloc and free. Consider using the ALLOCATABLE attribute or the ALLOCATE and DEALLOCATE statements.
fdate	fdate(3f)	Returns the date and time in ASCII string. Use as a subroutine.
ffrac	flmin(3f)	Returns the fractional accuracy of single-precision floating-point (REAL*4) numbers.
fgetc	fgetc(3f)	Returns a character from a specified logical unit.
flmax	flmin(3f)	Returns the maximum positive single-precision floating-point (REAL*4) value.
flmin	flmin(3f)	Returns the minimum positive single-precision floating-point (REAL*4) value.
flush	flush(3f)	Writes (flushes) the output in a user buffer to system buffer. Discards read-ahead data in user buffer.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
for_get_fpe	for_get_fpe(3f)	Returns the status of the floating-point exception (fpe) handling currently set for the program. Usually used with for_set_fpe. To use for_get_fpe from a C program, you must first call for_rtl_init_, described in this table. For more information, see Section 3.44, -fpen — Control Arithmetic Exception Handling and Reporting and Chapter 14.
for_rtl_init_	for_rtl_init_(3f)	Initializes the Compaq Fortran run-time library (RTL) environment for a C program. Use this subroutine from a main program written in C that calls Compaq Fortran subprograms. Calling this subroutine from the main C program initializes Compaq Fortran RTL data. It also establishes Compaq Fortran RTL signal handlers and floating-point exception handling so that the Compaq Fortran subprograms behave as if they were the main program. The trailing underscore (_) is required, and this routine must be called from a C program.
for_rtl_finish_	for_rtl_init_(3f)	Cleans up the Compaq Fortran run-time library (RTL) environment for a C main program that previously called for_rtl_init_. A status value is returned by this function. The trailing underscore (_) is required, and this routine must be called from a C program. Also see for_rtl_init_.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
for_set_fpe	for_get_fpe(3f)	Changes the floating-point exception (fpe) handling currently set for the program and also returns the status of the previous floating-point exception (fpe) handling setting. Can be used with for_get_fpe. To use for_set_fpe from a C program, you must first call for_rtl_init_, described in this table. For more information, see Section 3.44 and Chapter 14.
for_set_reentrancy	for_set_reentrancy(3f)	Sets reentrancy protection for the Fortran RTL.
fork	fork(3f)	Creates a copy of the calling process.
fortran	intro(3f)	Lists Fortran library routines.
fputc	putc(3f)	Writes a character to a specified logical unit.
free	malloc(3f)	Frees the memory allocated by falloc or malloc. Consider using the ALLOCATABLE attribute or the ALLOCATE and DEALLOCATE statements.
fseek	fseek(3f)	Repositions a file on a logical unit.
fstat	stat(3f)	Returns information about file status. The file is specified as a Fortran 95/90 logical unit number.
fsync	fsync(3f)	Writes the output in buffer to permanent storage.
ftell	fseek(3f)	Repositions a file on a logical unit.
gerror	perror(3f)	Writes system error messages.
getarg	getarg(3f)	Returns the command line arguments.
getc	getc(3f)	Returns a character from a logical unit.
getcwd	getcwd(3f)	Returns the directory path of the current directory.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
getenv	getenv(3f)	Returns the value of environment variables.
getfd	getfd(3f)	For a file already opened by the Compaq Fortran run-time library (RTL), returns the file descriptor associated with a particular unit number.
getgid	getuid(3f)	Returns the group's id of the caller.
getlog	getlog(3f)	Returns the user's login name.
getpid	getuid(3f)	Returns the process id.
getuid	getuid(3f)	Returns the user's or group's id of the caller.
gmtime	time(3f)	Returns the system time in month, day, and so forth in G.M.T. (Greenwich Mean Time).
iargc	getarg(3f)	Returns the command-line arguments.
idate	idate(3f)	Returns the date or time in numeric form. Also available as an intrinsic subroutine (Compaq extension) described in the *Compaq Fortran Language Reference Manual*.
ierrno	perror(3f)	Returns a system error message number.
index	index(3f)	Returns the index of a substring within the string. Consider using INDEX intrinsic function described in the *Compaq Fortran Language Reference Manual*.
inmax	flmin(3f)	Returns the maximum positive integer value.
irand	rand(3f)	Generates random values. For an example program that uses irand, see Section 12.7.
irandm	random(3f)	Generates a positive integer random number.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
isatty	ttynam(3f)	Returns whether the specified unit is a terminal port. Use as a subroutine.
itime	idate(3f)	Returns the date or time in numeric form.
kill	kill(3f)	Sends a signal to a process.
len	index(3f)	Returns the length of a string. Use the LEN intrinsic function described in the *Compaq Fortran Language Reference Manual*.
lgamma	lgamma(3f)	Returns the REAL*4 log of the gamma function.
link	link(3f)	Makes a directory link to an existing file.
lnblnk	index(3f)	Returns the index of the last nonblank character in a string.
loc	loc(3f)	Returns the address of an object. Similar to the %LOC built-in function described in the *Compaq Fortran Language Reference Manual*.
long	long(3f)	Converts INTEGER*2 to INTEGER*4.
lshift	bit(3f)	Shifts a word left by *n* bits. Use as an intrinsic function.
lstat	stat(3f)	Returns information about a file or a symbolic link.
ltime	time(3f)	Returns the system time in month, day, hour, minute, and seconds for the time zone.
malloc	malloc(3f)	Returns the address of a block of virtual memory. See also free. Consider using the ALLOCATABLE attribute or the ALLOCATE and DEALLOCATE statements.
not	bit(3f)	Returns the bitwise NOT (complement) of the operand. Use as an intrinsic function.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
omp_destroy_lock	omp_destroy_lock(3f)	*(TU*X only)* Disassociates a lock variable from any locks. See Section D.1.2.1, omp_destroy_lock.
omp_get_dynamic	omp_get_dynamic(3f)	*(TU*X only)* Informs if dynamic thread adjustment is enabled. See Section D.1.1.1, omp_get_dynamic.
omp_get_max_threads	omp_get_max_threads(3f)	*(TU*X only)* Gets the maximum value that can be returned by calls to the omp_get_num_threads( ) function. See Section D.1.1.2, omp_get_max_threads.
omp_get_nested	omp_get_nested(3f)	*(TU*X only)* Informs if nested parallelism is enabled. See Section D.1.1.3, omp_get_nested.
omp_get_num_procs	omp_get_num_procs(3f)	*(TU*X only)* Gets the number of processors that are available to the program. See Section D.1.1.4, omp_get_num_procs.
omp_get_num_threads	omp_get_num_threads(3f)	*(TU*X only)* Gets the number of threads currently in the team executing the parallel region from which the routine is called. See Section D.1.1.5, omp_get_num_threads.
omp_get_thread_num	omp_get_thread_num(3f)	*(TU*X only)* Gets the thread number, within the team, in the range from zero to omp_get_num_threads( ) minus 1. See Section D.1.1.6, omp_get_thread_num.
omp_in_parallel	omp_in_parallel(3f)	*(TU*X only)* Informs whether or not a region is executing in parallel. See Section D.1.1.7, omp_in_parallel.
omp_init_lock	omp_init_lock(3f)	*(TU*X only)* Initializes a lock to be used in subsequent calls. See Section D.1.2.2, omp_init_lock.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
omp_set_dynamic	omp_set_dynamic(3f)	*(TU*X only)* Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. See Section D.1.1.8, omp_set_dynamic.
omp_set_lock	omp_set_lock(3f)	*(TU*X only)* Makes the executing thread wait until the specified lock is available. See Section D.1.2.3, omp_set_lock.
omp_set_nested	omp_set_nested(3f)	*(TU*X only)* Enables or disables nested parallelism. See Section D.1.1.9, omp_set_nested.
omp_set_num_threads	omp_set_num_threads(3f)	*(TU*X only)* Sets the number of threads to use for the next parallel region. See Section D.1.1.10, omp_set_num_threads.
omp_test_lock	omp_test_lock(3f)	*(TU*X only)* Try to set the lock associated with a lock variable. See Section D.1.2.4, omp_test_lock.
omp_unset_lock	omp_unset_lock(3f)	*(TU*X only)* Releases the executing thread from ownership of the lock. See Section D.1.2.5, omp_unset_lock.
or	bit(3f)	Returns the bitwise OR of two operands. Use as an intrinsic function.
_OtsGetMaxThreads	otsgetmaxthreads(3f)	*(TU*X only)* Returns the number of threads that are normally used for parallel processing in the current environment. See Section D.2.1, _OtsGetMaxThreads or mpc_maxnumthreads.
_OtsGetNumThreads	otsgetnumthreads(3f)	*(TU*X only)* Returns the number of threads being used or the number of created threads. See Section D.2.2, _OtsGetNumThreads or mpc_numthreads.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
_OtsGetThreadNum	otsgetthreadnum(3f)	*(TU*X only)* Returns a number that identifies the current thread. See Section D.2.3, _OtsGetThreadNum or mpc_my_threadnum.
_OtsInitParallel	otsinitparallel(3f)	*(TU*X only)* Starts slave threads for parallel processing. See Section D.2.4, _OtsInitParallel.
_OtsInParallel	otsinparallel(3f)	*(TU*X only)* Informs whether you are currently within a parallel region. See Section D.2.5, _OtsInParallel or mpc_in_parallel_region.
_OtsSetNumThreads	otssetnumthreads(3f)	*(TU*X only)* Sets the number of threads to use for the next parallel region. See Section D.2.6, _OtsSetNumThreads .
_OtsStopWorkers	otsstopworkers(3f)	*(TU*X only)* Stops any slave threads created by parallel library support. See Section D.2.7, _OtsStopWorkers or mpc_destroy.
perror	perror(3f)	Writes system error messages.
putc	putc(3f)	Writes a character to a Fortran 95/90 logical unit.
qsort	qsort(3f)	Performs a quick sort of array elements. For an example program that uses qsort, see Section 12.7.
rand	rand(3f)	Generates random values.
random	random(3f)	Generates a single-precision (REAL) random number.
rename	rename(3f)	Renames a file.
rshift	bit(3f)	Shifts a word right by *n* bits. Use as an intrinsic function.
shcom_connect	shcom_connect(3f)	*(TU*X only)* Allows multiple processes to access common block data in a shared library (uses memory mapping).
short	long(3f)	Converts INTEGER*4 to INTEGER*2.
signal	signal(3f)	Changes the action of a signal.

(continued on next page)

Table 12–3 (Cont.) 3f Functions and Subroutines

Name	Reference Page	Description
sleep	sleep(3f)	Suspends execution for an interval.
srand	rand(3f)	Initializes the seed for subsequent use of rand and irand.
stat	stat(3f)	Returns information about the status of specified file.
symlnk	link(3f)	Makes a symbolic directory link to an existing file.
system	system(3f)	Executes a shell command.
time	time(3f)	Returns the system time in number of seconds from 00:00:00 G.M.T. January 1, 1970. Also available as an intrinsic subroutine (Compaq extension) described in the *Compaq Fortran Language Reference Manual*.
ttynam	ttynam(3f)	Returns the name of a terminal port or returns blanks if specified unit is not a terminal port.
umask	umask(3f)	Sets the file mode creation mask (protection).
unlink	unlink(3f)	Removes a directory entry. See also link.
wait	wait(3f)	Waits for the process to terminate.
xor	bit(3f)	Returns the bitwise exclusive OR of two operands. Use as an intrinsic function.

For More Information:

- On an example program that uses the qsort and irand routines, see Section 12.7.

12.3 3hpf Routines

The **3hpf routines** *(TU*X only)* provide functions for using High Performance Fortran (HPF) constructs to control parallel execution characteristics. Although usually used in programs that will execute in parallel, you can also use these routines for nonparallel programs.

The 3hpf routines are contained in the HPF_LOCAL_LIBRARY library (described in this chapter) and in the HPF_LIBRARY library (described in the *Compaq Fortran Language Reference Manual*).

HPF programs execute with the support of Compaq MPI (Message Passing Interface).

For more information about Compaq MPI, see:

`http://www.compaq.com/hpc/software/dmpi.html`

Table 12–4 describes each of the Compaq Fortran 3hpf routines contained in the HPF_LOCAL_LIBRARY library.

Table 12–4 Compaq Fortran 3hpf HPF_LOCAL_LIBRARY Library Routines

Name	Reference Page	Description
`abstract_to_physical`	`abstract_to_physical`(3hpf)	Returns processor identification for the physical processor associated with a specified abstract processor relative to a global actual argument array.
`get_hpf_my_node`	`get_hpf_my_node`(3hpf)	Returns an integer value in the range 0 to *number of processors - 1*. This is the unique process number on which an instance of an extrinsic(hpf_local) is executing; on a scalar computer system the value is the constant 0.
`get_hpf_numnodes`	`get_hpf_numnodes`(3hpf)	Returns the number of peers on which the program is executing.
`global_alignment`	`global_alignment`(3hpf)	Returns information about the global HPF array actual argument associated with an array local dummy argument; has the same behavior and interface as the subroutine HPF_ALIGNMENT.
`global_bounds`	`global_bounds`(3hpf)	Returns the upper and lower bounds of the actual argument associated with an assumed shape local dummy array.

(continued on next page)

Table 12–4 (Cont.) Compaq Fortran 3hpf HPF_LOCAL_LIBRARY Library Routines

Name	Reference Page	Description
global_distribution	global_distribution (3hpf)	Returns information about the global HPF array actual argument associated with an array local dummy argument; has the same behavior and interface as the subroutine HPF_DISTRIBUTION.
global_template	global_template(3hpf)	Returns information about the global HPF array actual argument associated with an array local dummy argument; has the same behavior and interface as the subroutine HPF_TEMPLATE.
global_to_local	global_to_local(3hpf)	Converts a set of global coordinates within a global HPF actual argument array to an equivalent set of local coordinates within the associated local dummy array.
global_to_physical	global_to_physical (3hpf)	Converts a global reference to an array element in the HPF global actual array argument. The argument must be associated with an assumed-shape local dummy array into the number of the process to which the element is mapped and the subscripts to access that element using the associated assumed shape array dummy argument on that process.
hpf_synch	hpf_synch(3hpf)	Synchronizes execution of all peers.
local_to_global	local_to_global(3hpf)	Converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates within the associated global HPF actual argument array.

(continued on next page)

Table 12–4 (Cont.) Compaq Fortran 3hpf HPF_LOCAL_LIBRARY Library Routines

Name	Reference Page	Description
physical_to_abstract	physical_to_abstract(3hpf)	Returns coordinates for an abstract processor, relative to a global actual argument array, corresponding to a specified physical processor.

For More Information:

- On the 3hpf routines in the HPF_LIBRARY library, see the *Compaq Fortran Language Reference Manual* and intro(3hpf).

12.4 Reference Pages for the 3f and 3hpf Routines

The reference page intro(3f) describes the 3f routines. Individual 3f functions also have their own reference pages, as shown in Table 12–3.

The reference page intro(3hpf) describes the 3hpf routines. Individual 3hpf functions also have their own reference pages, as shown in Table 12–4. And each routine is summarized in the *Compaq Fortran Language Reference Manual*.

You can use the man command to view online reference page information in the following ways:

- When the topic exists for multiple section numbers, specify the desired section number before the topic to view a specific reference page. For example, the following command returns the reference page for the 3f access routine, not its section 2 (system call) counterpart:

  ```
  % man 3f access
  ```

  The following man command returns the 3hpf intro reference page on Tru64 UNIX systems:

  ```
  % man 3hpf intro
  ```

- Use the -f option to view a one-line summary of the specified topic. This is useful to determine whether the same topic is associated with multiple section numbers, such as most 3f routines. The following man command lists all occurrences of the access topic:

  ```
  % man -f access
  ```

12.5 EXTERNAL or INTRINSIC Declarations

Certain Compaq Fortran library routines have the same names as intrinsic functions or subroutines (subprograms). You need to make sure that the correct routine or intrinsic subprogram is used, as follows:

- To use an intrinsic subprogram rather than a 3f library routine, you should *not* declare the function or subroutine as external in an EXTERNAL statement.
- To use a 3f library routine rather than an intrinsic subprogram, you should specify the subprogram as external in an EXTERNAL statement.

The following 3f routines have names that match similar intrinsic subprograms:

```
and
idate
index
len
lshift
not
or
rshift
time
xor
```

For portability reasons, you should consider using the intrinsic routines instead of the equivalent 3f external routine.

For More Information:

- On the EXTERNAL and INTRINSIC statements, see *Compaq Fortran Language Reference Manual*.

12.6 Example Using the 3f Library Routine shcom_connect

Example 12–1 shows the use of the 3f routine `shcom_connect` *(UN*X only)*).

Example 12–1 Using the 3f Routine shcom_connect

```
C
C       FILE: shared_data.f - Example of initialized common data.
C
        BLOCK DATA shared_block_data

        INTEGER*8 init_data
        REAL operand, result
        COMMON /shared_data/ init_data, operand, result

        DATA    init_data/42/, operand/0.0/, result/0.0/

        END

C
C       FILE: compute_agent.f - Example Fortran program
C
        PROGRAM compute_agent
        INCLUDE '../include/shcom.f'

        INTEGER*8 init_data
        REAL operand, result
        COMMON /shared_data/ init_data, operand, result

        INTEGER shcom_connenct, stat
        EXTERNAL shcom_connenct

        stat = shcom_connect(init_data, '/tmp/shcom_demo')

        IF (stat .EQ. SHCOM_SUCCESS) THEN
            result = SQRT(operand)
        ELSE
            TYPE *, 'shcom_connect() failed, error = ', stat
        ENDIF

        STOP
        END
/*
 * FILE ui_agent.c - Example of initialized common data written in C.
 */

#include "shcom.h"

typedef struct {
    long init_data;
    float operand;
    float result;
    } demo_t;

    extern demo_t shared_data_;

    main() {
 int stat;
```

(continued on next page)

Example 12–1 (Cont.) Using the 3f Routine shcom_connect

```
printf("shared_data_.init_data = %d\n", shared_data_.init_data);

stat = shcom_connect(&shared_data_, "/tmp/shcom_demo");

if (stat == SHCOM_SUCCESS) {
    shared_data_.operand = 2.0;
    shared_data_.result = 0.0;
    system("compute_agent");
    printf("shared_data_.result = %f\n", shared_data_.result);
} else {
    printf("shcom_connect() failed, error = %d\n", stat);
}
   }

   #
   #       FILE: Makefile  - Builds shared common example.
   #
   FC = f90
   CFLAGS = -g -I../include
   FFLAGS = -g
   LDFLAGS = -g
   LIBS = ../lib/libshcom.a

   all:    ui_agent compute_agent

   ui_agent:       ui_agent.o shared_data.so
    cc ${CFLAGS} -o ui_agent ui_agent.o shared_data.so ${LIBS}

   compute_agent:  compute_agent.o shared_data.so
    f90 ${FFLAGS} -o compute_agent compute_agent.o shared_data.so
   ${LIBS}

   shared_data.so: shared_data.o
    ld ${LDFLAGS} -shared -o shared_data.so shared_data.o -lc

   test:   shared_data.so ui_agent compute_agent
    LD_LIBRARY_PATH=.; export LD_LIBRARY_PATH; ui_agent

   clean:
    -rm -f ui_agent compute_agent
    -rm -f *.o shared_data.so
    -rm -f /tmp/shcom_demo so_locations
```

To compile and link this program, enter:

```
% make all
```

To run this program, enter:

```
% make test
```

The output from the program is as follows:

```
shared_data_.init_data = 42
shared_data_.result = 1.414214
```

12.7 Example of the 3f Library Routines irand and qsort

Example 12–2 shows the use of the 3f routines irand and qsort.

Example 12–2 Using the 3f Routines irand and qsort

```
      PROGRAM EXAMPLE
!
!     This is an example of calling the IRAND(3F) and QSORT(3F) entries.
!
        EXTERNAL IRAND, QSORT, SUBSORT
        INTEGER (KIND=4) :: IRAND, SUBSORT
        INTEGER (KIND=4) :: IARRAY(10), I

        WRITE(6,100)
!
!     Initialize the array using the IRAND(3F) routine.
!
        DO I=1,10
          IARRAY(I) = IRAND(0)
        END DO
        WRITE(6,120) 'IRAND(3F)', IARRAY
!
!     Now to sort the array using QSORT(3F)
!
        CALL QSORT( IARRAY, 10, 4, SUBSORT)
        WRITE(6,120) 'QSORT(3F)', IARRAY
!
!     Define FORMAT statements
!
100     FORMAT ('0Start of EXAMPLE'//)
120     FORMAT ('0Array contents after ',A,' call:'/, 10(/T20,I) //)
        STOP
      END PROGRAM EXAMPLE
!
!   Subroutine called by QSORT(3F)
!
      INTEGER (KIND=4) FUNCTION SUBSORT (A,B)
        INTEGER (KIND=4) :: A,B
        SUBSORT = 1
        IF ( A == B ) SUBSORT = 0
        IF ( A < B ) SUBSORT = -1
        RETURN
```

(continued on next page)

Example 12–2 (Cont.) Using the 3f Routines irand and qsort

```
      END FUNCTION SUBSORT
```

To compile and link this program (named `example`), enter:

```
% f90 -o example example.for
```

To run this program and redirect its output from `stdout` to `example.out`, enter:

```
% example > example.out
```

The output from the program `example` contains the contents of the array before and after sorting, as follows:

```
0Start of EXAMPLE

0Array contents after IRAND(3F) call:

                        16838
                         5758
                        10113
                        17515
                        31051
                         5627
                        23010
                         7419
                        16212
                         4086

0Array contents after QSORT(3F) call:

                         4086
                         5627
                         5758
                         7419
                        10113
                        16212
                        16838
                        17515
                        23010
                        31051
```

13

Using the Compaq Extended Math Library (CXML)

This chapter contains the following topics:

- Section 13.1, What Is CXML?
- Section 13.2, CXML Routine Groups
- Section 13.3, Using CXML from Fortran
- Section 13.4, CXML Program Example
- Section 13.5, CXML Documentation

13.1 What Is CXML?

The Compaq Extended Math Library (CXML) provides a comprehensive set of mathematical library routines callable from Fortran and other languages. CXML contains a set of over 1500 high-performance mathematical subprograms designed for use in many different types of scientific and engineering applications.

CXML is included with Compaq Fortran for Tru64 UNIX Systems and can be installed using the instructions in the *Compaq Fortran Installation Guide for Tru64 UNIX Systems*.

CXML kits for Tru64 UNIX are also available from the Compaq Math Libraries Web site, which always has the latest version:

`http://www.compaq.com/math`

Since CXML might have been updated since the Compaq Fortran kit was released, you should check the Web site to make sure you have the latest version.

CXML is available as a separate download item for Linux Alpha systems. For more information, see the Compaq Math Libraries Web site.

CXML documentation is also available at the Web site. See Section 13.5, CXML Documentation.

13.2 CXML Routine Groups

CXML routines include those for basic linear algebra (BLAS), signal processing, sparse linear system solution, linear algebra (LAPACK), and utilities related to random numbers, vector math, and sorting. The routines are described in Table 13–1.

Table 13–1 CXML Routine Groups

Name	Description
Basic Linear Algebra	The Basic Linear Algebra Subprograms (BLAS) library includes the industry-standard Basic Linear Algebra Subprograms for Level 1 (vector-vector, BLAS1), Level 2 (matrix-vector, BLAS2), and Level 3 (matrix-matrix, BLAS3). Also included are subprograms for BLAS Level 1 Extensions, and Sparse BLAS Level 1.
Signal Processing	The Signal Processing library provides a basic set of signal processing functions. Included are one-, two-, and three-dimensional Fast Fourier Transforms (FFT), group FFTs, Cosine/Sine Transforms (FCT/FST), Convolution, Correlation, and Digital Filters.
Sparse Linear System	The Sparse Linear System library provides both direct and iterative sparse linear system solvers. The direct solver package supports both symmetric and nonsymmetric sparse matrices stored using the skyline storage scheme. The iterative solver package contains a basic set of storage schemes, preconditioners, and iterative solvers.
LAPACK	LAPACK is an industry-standard subprogram package offering an extensive set of linear system and eigenproblem solvers. LAPACK uses blocked algorithms that are better suited to most modern architectures, particularly ones with memory hierarchies.
Utility subprograms	Utility subprograms include random number generation, vector math functions, and sorting subprograms.

Where appropriate, each subprogram has a version to support each combination of real or complex and single or double precision arithmetic. In addition, selected key CXML routines are available in parallel form as well as serial form on Compaq Tru64 UNIX systems.

13.3 Using CXML from Fortran

To use CXML, you need to make the CXML routines and their interfaces available to your program and specify the appropriate libraries when linking. To specify the CXML routines library when linking, use the `-lcxml` option. To compile and link a Fortran program that contains calls to CXML routines on Tru64 UNIX or Linux Alpha systems, use one of the following commands:

Operating System	Command
Tru64 UNIX	`f90 my_prog.f90 -lcxml`
Linux	`fort my_prog.f -lcxml`

For example, to link a Fortran 90 program with the serial CXML library on a Tru64 UNIX system, you would give this command:

```
% f90 my_prog.f90 -lcxml
```

On Tru64 UNIX systems, selected key CXML routines have been parallelized using OpenMP. To link with the parallel version of the CXML library, use `-lcxmlp` instead of `-lcxml`.

13.4 CXML Program Example

Example 13–1, Fortran Example Program Using CXML invokes the function SAXPY from the BLAS portion of the CXML Libraries. The SAXPY function computes `a*x+y`.

13.5 CXML Documentation

For more information, see the following CXML documentation, available at the Math Libraries Web site described in Section 13.1, What Is CXML?:

- README file and Release Notes
- Reference Manual in HTML, PDF, and PS format

When CXML is installed on Tru64 UNIX systems, subsets containing reference pages in both traditional (“man page”) and HTML format can be installed.

When CXML is installed on Linux systems, the reference pages are available in HTML format and are placed in a `/usr/doc` subdirectory.

Example 13–1 Fortran Example Program Using CXML

```
   PROGRAM example
!
!  This free-form example demonstrates how to call
!  CXML routines from Fortran.
!
   REAL(KIND=4)    :: a(10)
   REAL(KIND=4)    :: x(10)
   REAL(KIND=4)    :: alpha
   INTEGER(KIND=4) :: n
   INTEGER(KIND=4) :: incx
   INTEGER(KIND=4) :: incy
   n = 5 ; incx = 1 ; incy = 1 ; alpha = 3.0
   DO i = 1,n
     a(i) = FLOAT(i)
     x(i) = FLOAT(2*i)
   ENDDO
   PRINT 98, (a(i),i=1,n)
   PRINT 98, (x(i),i=1,n)
98 FORMAT(' Input = ',10F7.3)
   CALL saxpy( n, alpha, a, incx, x, incy )
   PRINT 99, (x(i),I=1,n)
99 FORMAT(/,' Result = ',10F7.3)
   STOP
   END PROGRAM example
```

14

Controlling Floating-Point Exceptions

Note

This chapter applies only to Compaq Fortran Tru64 UNIX systems.

This chapter contains the following topics:

- Section 14.1, Overview of Controlling Floating-Point Exceptions
- Section 14.2, Using the for_fpe_flags.f File
- Section 14.3, Calling the for_get_fpe and for_set_fpe Functions
- Section 14.4, File fordef.f and Its Usage

14.1 Overview of Controlling Floating-Point Exceptions

This chapter contains information about controlling exceptions that can occur during the run-time processing of floating-point numbers. These exceptions are **underflow**, **division by zero**, **overflow**, and **invalid operation** (such as the square root of a negative number).

See Section 9.4, Native IEEE Floating-Point Representations and Exceptional Values for information about the internal representation of floating-point numbers including exceptional values (such as plus infinity).

You can use these command-line options to direct the processing of Compaq Fortran floating-point numbers at run time:

- `-fpe`*n* option to control exception handling and reporting (see Section 3.44)
- `-check underflow` option to control checking for floating-point underflow at run time (see Section 3.29)
- `-synchronous_exceptions` option to control reporting of exceptions at run time (see Section 3.86)

- `-fprm` *keyword* option to control rounding of floating-point operations (see Section 3.46)

14.2 Using the for_fpe_flags.f File

The `-fpe`*n* option, as explained in Section 3.44, lets you control floating-point exceptions according to the value of *n*. Table 3–3 contains the values of *n* and the corresponding results at run time when floating-point exceptions occur. These exceptions are underflow, division by zero, overflow, and invalid operation (such as the square root of a negative number).

Normally the `-fpe`*n* option, along with the `-check underflow` and `-synchronous_exceptions` options, gives you adequate control over floating-point exception handling and reporting.

However, if any combination of these three options does not give you adequate control, then you can use the following:

- File `/usr/include/for_fpe_flags.f`, which contains definitions of the bits in an INTEGER*4 variable. (See Section 14.2.1, Bit Definitions in File for_fpe_flags.f.) This file is a file of Fortran statements with flags that relate to floating-point exceptions.
- Functions `for_get_fpe` and `for_set_fpe`. (See Section 14.3, Calling the for_get_fpe and for_set_fpe Functions.)
- Option `-fpe3`.

For example, suppose you want to read a denormalized number from an unformatted file and perform calculations on it without generating an exception about an invalid operation. Suppose you also want to have divide-by-zero exceptions trapped and reported. While the `-fpe3` option meets the first requirement, it also results in no trapping and reporting of divide-by-zero exceptions.

The general steps of the solution to this example are:

1. Include, in your source program, file `/usr/include/for_fpe_flags.f`.
2. Include statements that select, from the included file, bits corresponding to the phrases “trap divide-by-zero exceptions” and “report divide-by-zero exceptions.” Include them in the INTEGER*4 argument to function `for_set_fpe`.

3. Compile the program with the -fpe3 option. As long as the reference to function for_set_fpe occurs at the beginning of the program, division-by-zero exceptions will be trapped and reported. (When function for_set_fpe executes, it overrides the initial effect of the -fpe3 option: to disable trapping and reporting of divide-by-zero exceptions.)

This example reappears later, in program fpe_div0_msg.f90 in Section 14.3.2.

Note

Normally the -fpe*n*, -check underflow, -synchronous_exceptions, and -fprm *keyword* options give you sufficient control over the processing of floating-point exceptions. Using file /usr/include/for_fpe_flags.f and function for_set_fpe can yield unanticipated results.

14.2.1 Bit Definitions in File for_fpe_flags.f

Table 14–1 explains the bit definitions in file /usr/include/for_fpe_flags.f. The compiler automatically generates a call to for_set_fpe( ), which is expected when the program begins execution. The argument to for_set_fpe( ) has the appropriate bits set (as defined in for_fpe_flags.f) to achieve the documented behavior for the -fpe*n* option specified at compile time. You have the option of changing this bit setting according to the individual bit definitions and their corresponding effects in Table 14–1. The function for_set_fpe works with the bit definitions to make the change.

Note

For the default -fpe0 option, the compiler does not generate the call to for_set_fpe( ). The run-time library automatically initializes floating-point exception behavior in this case at run-time initialization.

Table 14–1 Bit Definitions in File for_fpe_flags.f

Bit Name	Effect When Set
FPE_M_TRAP_UND	Requests delivery of underflow traps to the current signal handler.
FPE_M_TRAP_OVF	Requests delivery of overflow traps to the current signal handler.

(continued on next page)

Table 14–1 (Cont.) Bit Definitions in File for_fpe_flags.f

Bit Name	Effect When Set
FPE_M_TRAP_DIV0	Requests delivery of division-by-zero traps to the current signal handler.
FPE_M_TRAP_INV	Requests delivery of invalid operation traps to the current signal handler.
FPE_M_MSG_UND	If the current signal handler is the default handler provided with the Fortran run-time library and bit FPE_M_TRAP_UND is also set, then the default handler sends a message to `stderr` for each of the first two occurrences of underflow traps. The default handler also sends a total count of underflow traps to `stderr` when the program terminates. If the current signal handler is not the default handler, then bit FPE_M_MSG_UND has no effect.
FPE_M_MSG_OVF	If the current signal handler is the default handler provided with the Fortran run-time library and bit FPE_M_TRAP_OVF is also set, then the default handler sends a message to `stderr` for each of the first two occurrences of overflow traps. The default handler also sends a total count of overflow traps to `stderr` when the program terminates. If the current signal handler is not the default handler, then bit FPE_M_MSG_OVF has no effect.
FPE_M_MSG_DIV0	If the current signal handler is the default handler provided with the Fortran run-time library and bit FPE_M_TRAP_DIV0 is also set, then the default handler sends a message to `stderr` for each of the first two occurrences of divide-by-zero traps. The default handler also sends a total count of divide-by-zero traps to `stderr` when the program terminates. If the current signal handler is not the default handler, then bit FPE_M_MSG_DIV0 has no effect.
FPE_M_MSG_INV	If the current signal handler is the default handler provided with the Fortran run-time library and bit FPE_M_TRAP_INV is also set, then the default handler sends a message to `stderr` for each of the first two occurrences of invalid operation traps. The default handler also sends a total count of invalid operation traps to `stderr` when the program terminates. If the current signal handler is not the default handler, then bit FPE_M_MSG_INV has no effect.
FPE_M_ABRUPT_UND	Replaces denormalized numbers obtained as results of calculations with zeroes.
FPE_M_ABRUPT_OVF	Reserved.
FPE_M_ABRUPT_DIV0	Reserved.
FPE_M_ABRUPT_INV	Reserved.

(continued on next page)

Table 14–1 (Cont.) Bit Definitions in File for_fpe_flags.f

Bit Name	Effect When Set
FPE_M_ABRUPT_DMZ	Replaces denormalized numbers used as input operands to floating-point instructions with zeroes. This does not have anything to do with reading data from unformatted files.

14.3 Calling the for_get_fpe and for_set_fpe Functions

Table 14–1 contains a list of bit definitions (in an INTEGER*4 variable) and their run-time effects when floating-point exceptions occur. Functions `for_get_fpe` and `for_set_fpe` read and set the bits, respectively, after a program begins execution. (The `-fpe`*n* option also sets these bits when a program begins execution.)

14.3.1 Calling for_get_fpe

Function `for_get_fpe` has no argument. It returns an INTEGER*4 variable whose bits specify how the run-time library currently handles floating-point exceptions. The bits of the variable are defined in Table 14–1. `for_get_fpe` must be declared INTEGER(4).

For example, consider the following program. When compiled with the default `-fpe0` option, this program displays the default settings of the bits in the INTEGER*4 variable:

```
      PROGRAM TESTFGPE_FPE0
!
!     This program returns floating-point exception flags
!        using function for_get_fpe().
!
      integer*4 fpe_flags
      integer*4 for_get_fpe
      external for_get_fpe
!
      PRINT *, ''
      PRINT *, 'Start of program'
      fpe_flags = for_get_fpe()
      PRINT *, ''
      PRINT *, 'for_get_fpe() has returned, with option fpe0:'
      WRITE (*, 200) fpe_flags
200   FORMAT (' ', 'In B32 format: ', B32)
      PRINT *, ''
      PRINT *, 'End of program'

      END PROGRAM TESTFGPE_FPE0
```

The compilation and execution commands are:

```
% f90 -fpe0 testfgpe_fpe0.f90
% a.out
```

The output from this program, with spaces added to the 32-character representation of variable *fpe_flags*, is:

```
Start of program

for_get_fpe() has returned, with option fpe0:
In B32 format:                1 0000 0000 0000 1110

End of program
```

Compiling this program with a different value of the `-fpe`*n* option gives different output. For example, if you compile with `-fpe3`, variable *fpe_flags* contains zero.

For More Information:

- On the `for_get_fpe` function, see `for_set_fpe`(3f).

14.3.2 Calling for_set_fpe

Function `for_set_fpe` has a single argument: a floating-point exception behavior mask. This is an INTEGER*4 variable whose bits specify how the run-time library will handle floating-point exceptions. Function `for_set_fpe` must be declared INTEGER(4).

The bits of the mask variable are defined in Table 14–1. When the function executes, it both returns the previous value of the mask and changes the mask to the value of the argument. Recall that the `-fpe`*n* option determines the value of the mask when the program begins to execute. Function `for_set_fpe` allows you to change the mask, and the handling of floating-point exceptions, after the program begins to execute.

Recall in Section 14.2 the example where the requirements are to read a denormalized number from an unformatted file and perform calculations on it without generating an invalid operation exception. Also, divide-by-zero exceptions must be trapped and reported. The `-fpe3` option meets the first requirement. File `/usr/include/for_fpe_flags.f` and function `for_set_fpe` work together to meet the second requirement.

Consider program `fpe_div0_msg.f90`:

```
      program fpe_div0_msg
!     Trap when division by zero occurs and display a message.
!     Compile with the command   f90 -fpe3 fpe_div0_msg.f90
      external for_set_fpe      ! External function

      integer*4 for_set_fpe

!     Include the file with the definitions of the
!        floating-point exceptions.
      include '/usr/include/for_fpe_flags.f'
      real*4 a,b,c
      integer*4 old_fpe_flags, new_fpe_flags
!     Set the bits, of the argument to for_set_fpe(), to trap
!        when division by zero occurs and to display a message.
      new_fpe_flags = FPE_M_TRAP_DIV0 + FPE_M_MSG_DIV0
      old_fpe_flags = for_set_fpe(new_fpe_flags)
      a = 5.0
      print *, ''
      print *, 'Give me 0.0'
      read *, b
      print *, 'Division by zero is next'
      c = a / b
      print *, 'The result of division by zero is', c
      print *, ''
      end
```

In this program, at run time, division by zero occurs. This floating-point exception results in:

1. A trap and its delivery to the current signal handler
2. Reporting by the default signal handler provided with the Compaq Fortran run-time library
3. Continuation of the program

The compilation and execution commands are:

```
% f90 -fpe3 fpe_div0_msg.f90
% a.out
```

The output from this program and user input (0.0) are:

```
 Give me 0.0
0.0
 Division by zero is next
forrtl: error (73): floating divide by zero
 The result of division by zero is Infinity

forrtl: info (299): 1 floating divide-by-zero traps
```

Setting bit FPE_M_TRAP_DIV0 causes a trap when division by zero occurs and delivery of a signal to the current signal handler. Setting bit FPE_M_MSG_DIV0 causes the Compaq Fortran run-time routines (the current and default signal handler) to display a message about the trapped division-by-zero floating-point exception. Because of compilation with -fpe3, function for_set_fpe returns zero to variable *old_fpe_flags*.

For More Information:

- On the for_set_fpe function, see for_set_fpe(3f).

14.4 File fordef.f and Its Usage

The parameter file /usr/include/fordef.f contains symbols and INTEGER*4 values corresponding to the classes of floating-point representations. Some of these classes are exceptional ones such as bit patterns that represent positive denormalized numbers. See Section 9.4.8, Exceptional Floating-Point Representations.

With this file of symbols and with the FP_CLASS intrinsic function, you have the flexibility of identifying exceptional numbers so that, for example, you can replace positive and negative denormalized numbers with true zero.

The following is a simple example of identifying floating-point bit representations:

```
include '/usr/include/fordef.f'
real*4 a
integer*4 class_of_bits
a = 57.0   ! Bit pattern is an Alpha finite number
class_of_bits = fp_class(a)
if ( class_of_bits .eq. for_k_fp_pos_norm  .or.   &
     class_of_bits .eq. for_k_fp_neg_norm       ) then
   print *, a, ' is a non-zero and non-exceptional value'
else
   print *, a, ' is zero or an exceptional value'
end if
end
```

In this example, the symbol for_k_fp_pos_norm in file /usr/include/fordef.f plus the REAL*4 value 57.0 to the FP_CLASS intrinsic function results in the execution of the first print statement.

Table 14–2 explains the symbols in file /usr/include/fordef.f and their corresponding floating-point representations. Section 9.4.8, Exceptional Floating-Point Representations explains each representation.

Table 14–2 Symbols in File fordef.f

Symbol Name	Class of Floating-Point Bit Representation
FOR_K_FP_SNAN	Signaling NaN
FOR_K_FP_QNAN	Quiet NaN
FOR_K_FP_POS_INF	Positive infinity
FOR_K_FP_NEG_INF	Negative infinity
FOR_K_FP_POS_NORM	Positive normalized finite number
FOR_K_FP_NEG_NORM	Negative normalized finite number
FOR_K_FP_POS_DENORM	Positive denormalized number
FOR_K_FP_NEG_DENORM	Negative denormalized number
FOR_K_FP_POS_ZERO	Positive zero
FOR_K_FP_NEG_ZERO	Negative zero

Another example of using file `fordef.f` and intrinsic function FP_CLASS follows. The goals of this program are to quickly read any 32-bit pattern into a REAL*4 number from an unformatted file with no exception reporting and to replace denormalized numbers with true zero:

```
      include '/usr/include/fordef.f'
      real*4 a(100)
      integer*4 class_of_bits
!     open an unformatted file as unit 1
!     ...
      read (1) a
      do i = 1, 100
         class_of_bits = fp_class(a(i))
         if ( class_of_bits .eq. for_k_fp_pos_denorm  .or.   &
              class_of_bits .eq. for_k_fp_neg_denorm       ) then
            a(i) = 0.0
         end if
      end do
      close (1)
      end
```

You can compile this program with any value of `-fpe`*n*. Intrinsic function FP_CLASS helps to find and replace denormalized numbers with zeroes before the program can attempt to perform calculations on the denormalized numbers. On the other hand, if this program did not replace denormalized numbers read from unit 1 with zeroes and the program was compiled with `-fpe0`, then the first attempted calculation on a denormalized number would result in a floating-point exception.

File fordef.f and intrinsic function FP_CLASS can work together to identify NaNs. A variation of the previous example would contain the symbols for_k_fp_snan and for_k_fp_qnan in the IF statement. A faster way to do this is based on the intrinsic ISNAN function. One modification of the previous example, using ISNAN, follows:

```
!     The ISNAN function does not need file /usr/include/fordef.f
      real*4 a(100)
!     open an unformatted file as unit 1
!     ...
      read (1) a
      do i = 1, 100
         if ( isnan (a(i)) ) then
            print *, 'Element ', i, ' contains a NaN'
         end if
      end do
      close (1)
      end
```

You can compile this program with any value of -fpe*n*.

A

Compatibility: Compaq Fortran 77 and Compaq Fortran on Multiple Platforms

This appendix provides compatibility information for those porting Compaq Fortran 77 and Compaq Fortran applications from other Compaq systems and for those designing applications for portability to multiple platforms.

This appendix contains the following topics:

- Section A.1, Compaq Fortran and Compaq Fortran 77 Compatibility on Various Platforms
- Section A.2, Compatibility with Compaq Fortran 77 for Compaq Tru64 UNIX Systems
- Section A.3, Language Compatibility with Compaq Visual Fortran
- Section A.4, Compatibility with Compaq Fortran 77 and Compaq Fortran for OpenVMS Systems
- Section A.5, Calling Between Compaq Fortran 77 and Compaq Fortran

A.1 Compaq Fortran and Compaq Fortran 77 Compatibility on Various Platforms

Table A–1 summarizes the compatibility of Compaq Fortran for Compaq Tru64 UNIX and Linux Alpha systems with Compaq Fortran on OpenVMS Alpha Systems, Compaq Fortran 77 on other platforms (architecture/operating system pairs), and Compaq Visual Fortran for Windows systems.

Table A–1 Summary of Language Compatibility

Language Feature	Compaq Fortran 77 (CF77) or Compaq Fortran (CF95) for . . . Systems						
	CF95 UNIX Alpha	CF95 Linux Alpha	CF77 UNIX Alpha	CF95 Windows	CF95 OpenVMS Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Linking against static and shared libraries	X	X	X	X	X	X	X
Create code for shared libraries	X	X	X	X	X	X	X
Recursive code support	X	X	X	X	X	X	X
AUTOMATIC and STATIC statements	X	X	X	X	X	X	X
STRUCTURE and RECORD declarations	X	X	X	X	X	X	X
INTEGER*1, *2, *4	X	X	X	X	X	X	X
LOGICAL*1, *2, *4	X	X	X	X	X	X	X
INTEGER*8 and LOGICAL*8	X	X	X	X[1]	X	X	
REAL*4, *8	X	X	X	X	X	X	X
REAL*16[2]	X	X	X		X	X	X
COMPLEX*8, *16	X	X	X	X	X	X	X
COMPLEX*32[3]	X	X			X		
POINTER (CRAY-style)	X	X	X	X	X	X	X
INCLUDE statements	X	X	X	X	X	X	X
IMPLICIT NONE statements	X	X	X	X	X	X	X
Data initialization in type declarations	X	X	X	X	X	X	X
Automatic arrays	X	X	X	X	X	X	
VOLATILE statements	X	X	X	X	X	X	X
NAMELIST-directed I/O	X	X	X	X	X	X	X
31-character names including $ and _	X	X	X	X	X	X	X
Source listing with machine code	X	X	X	X	X	X	X
Debug statements in source	X	X	X	X	X	X	X

[1]Alpha systems only.

[2]For REAL*16 data, OpenVMS VAX systems use H_float format, and Alpha systems use IEEE style X_float format.

[3]For COMPLEX*32 data, Alpha systems use IEEE style X_float format for both REAL*16 parts.

(continued on next page)

Table A–1 (Cont.) Summary of Language Compatibility

Language Feature	Compaq Fortran 77 (CF77) or Compaq Fortran (CF95) for . . . Systems						
	CF95 UNIX Alpha	CF95 Linux Alpha	CF77 UNIX Alpha	CF95 Windows	CF95 OpenVMS Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Bit constants to initialize data and use in arithmetic	X	X	X	X	X	X	X
DO WHILE and END DO statements	X	X	X	X	X	X	X
Built-in functions %LOC, %REF, %VAL	X	X	X	X	X	X	X
SELECT CASE construct	X	X	X	X	X	X	
EXIT and CYCLE statements	X	X	X	X	X	X	
Variable FORMAT expressions (VFEs)	X	X	X	X	X	X	X
! marks end-of-line comment	X	X	X	X	X	X	X
Optional run-time bounds checking for arrays and substrings	X	X	X	X	X	X	X
Binary (unformatted) I/O in IEEE big endian, IEEE little endian, VAX, IBM, and CRAY floating-point formats	X	X	X	X	X	X	X
Fortran 95/90 standards checking	X	X		X	X		
FORTRAN-77 standards checking			X	X		X	X
IEEE exception handling	X	X	X	X	X	X	
VAX floating data type in memory					X	X	X
IEEE floating data type in memory	X	X	X	X	X	X	
CDD/Repository DICTIONARY support						X	X
KEYED access and INDEXED files					X	X	X
Parallel decomposition	X[5]		[5]	[5]	[5]	[5]	X
OpenMP parallel directives	X						
Conditional compilation using IF . . . DEF constructs	X	X		X	X		
Vector code support							X

[5]For parallel processing, you can also use the optional KAP performance preprocessor for a shared memory multiprocessor system.

(continued on next page)

Table A–1 (Cont.) Summary of Language Compatibility

Language Feature	Compaq Fortran 77 (CF77) or Compaq Fortran (CF95) for . . . Systems						
	CF95 UNIX Alpha	CF95 Linux Alpha	CF77 UNIX Alpha	CF95 Windows	CF95 OpenVMS Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Direct inlining of Basic Linear Algebra Subroutines (BLAS)	[6]	[6]	[6]	[6]	[6]	[6]	X
DATE_AND_TIME returns 4-digit year	X	X	X	X	X	X	X
FORALL statement and construct	X	X		X	X		
Automatic deallocation of ALLOCATABLE arrays	X	X		X	X		
Dim argument to MAXLOC and MINLOC	X	X		X	X		
PURE user-defined subprograms	X	X		X	X		
ELEMENTAL user-defined subprograms	X	X		X	X		
Pointer initialization (initial value)	X	X		X	X		
The NULL intrinsic to nullify a pointer	X	X		X	X		
Derived-type structure initialization	X	X		X	X		
CPU_TIME intrinsic subroutine	X	X		X	X		
Kind argument to CEILING and FLOOR intrinsics	X	X		X	X		
Nested WHERE constructs, masked ELSEWHERE statement, and named WHERE constructs	X	X		X	X		
Comments allowed in namelist input	X	X		X	X		
Generic identifier in END INTERFACE statements	X	X		X	X		
Minimal FORMAT edit descriptor field width	X	X		X	X		
Detection of Obsolescent and/or Deleted features [7]	X	X		X	X		

[6]BLAS and other routines are available with the Compaq Extended Mathematical Library (CXML) product on Alpha systems.

[7]Compaq Fortran flags these deleted and obsolescent features, but fully supports them.

A.2 Compatibility with Compaq Fortran 77 for Compaq Tru64 UNIX Systems

This section provides compatibility information for those porting Compaq Fortran 77 applications from Compaq Tru64 UNIX systems. It discusses the following topics:

- Major language features for compatibility with Compaq Fortran 77 for Compaq Tru64 UNIX systems (Section A.2.1)
- Language differences between Compaq Fortran and Compaq Fortran 77, including Compaq Fortran 77 extensions on Compaq Tru64 UNIX Systems that are not supported by this version of Compaq Fortran on Compaq Tru64 UNIX Systems (Section A.2.2)
- Language features detected during compilation differently by Compaq Fortran than Compaq Fortran 77 for Compaq Tru64 UNIX Systems (Section A.2.3)

A.2.1 Major Language Features for Compatibility with Compaq Fortran 77 for Compaq Tru64 UNIX Systems

To simplify porting applications from Compaq Fortran 77 to Compaq Fortran on Tru64 UNIX systems, Compaq Fortran supports the following Compaq Fortran 77 extensions that are not part of the Fortran 95/90 standards:

- Record structures (STRUCTURE and RECORD statements)
- I/O statements, including PRINT, ACCEPT, TYPE, DELETE, and UNLOCK
- I/O statement specifiers, such as the INQUIRE statement specifiers CARRIAGECONTROL, CONVERT, ORGANIZATION, and RECORDTYPE
- Certain data types, including 8-byte INTEGER and LOGICAL variables and 16-byte REAL variables (available on Alpha systems)
- Size specifiers for data declaration statements, such as INTEGER*4, in addition to the KIND type parameter
- IEEE floating-point data type in memory
- The POINTER statement and its associated data type (CRAY pointers).
- The typeless PARAMETER statement
- The VOLATILE statement
- The AUTOMATIC and STATIC statements

- Built-in functions used in argument lists, such as %VAL and %LOC
- Hollerith constants
- Variable-format expressions (VFEs)
- Certain intrinsic functions
- The tab source form (resembles fixed-source form)
- I/O formatting descriptors
- USEROPEN routines for user-defined open routines
- Additional language features, including the DEFINE FILE, ENCODE, DECODE, and VIRTUAL statements

In addition to language extensions, Compaq Fortran also supports the following Compaq Fortran 77 features:

- Compaq Fortran 77 compilation control statements and directives (see the *Compaq Fortran Language Reference Manual*), including:
  - INCLUDE statement forms using /LIST and /NOLIST (requires compiling with `-vms`)
  - OPTIONS statement to override or set compiler command-line options
  - General cDEC$ directives, including:

    cDEC$ ALIAS
    cDEC$ IDENT
    cDEC$ OPTIONS
    cDEC$ PSECT
    cDEC$ TITLE
    cDEC$ SUBTITLE
- A nearly identical set of command-line options and their associated features (see Section A.2.4).
- The ability to call between Compaq Fortran 77 and Compaq Fortran routines and a common run-time environment. For example, a Compaq Fortran 77 procedure and a Compaq Fortran procedure can perform I/O to the same unit number (see Section A.5).
- `foriosdef.for` symbolic parameter definitions for use with run-time (IOSTAT) error handling (see Chapter 8).

For More Information:

On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.2.2 Language Features Provided Only by Compaq Fortran 77 for Compaq Tru64 UNIX Systems

Compaq Fortran conforms to the Fortran 95/90 standard, which is a superset of the FORTRAN-77 standard. Compaq Fortran provides many but not all of the FORTRAN-77 extensions provided by Compaq Fortran 77.

The following list shows FORTRAN-77 extensions provided by Compaq Fortran 77 on Compaq Tru64 UNIX systems are *not* provided by Compaq Fortran. Where appropriate, this list indicates equivalent Compaq Fortran language features:

- Octal notation for integer constants is not part of the Compaq Fortran Language. Compaq Fortran 77 (`f77` command) only supports this feature when the `-vms` option is specified. For example:

```
I = "0014         ! Assigns 12 to I, not supported by Compaq Fortran
```

- The Compaq Fortran compiler discards leading zeros for `"disp"` in the STOP statement. For example:

```
STOP 001   ! Prints 1 instead of 001
```

- When a single-precision constant is assigned to a double-precision variable, Compaq Fortran 77 evaluates the constant in double precision. The Fortran 95/90 standards require that the constant be evaluated in single precision.

  When a single-precision constant is assigned to a double-precision variable with Compaq Fortran, it is evaluated in single precision. You can, however, specify the `f90 -fpconstant` option to request that a single-precision constant assigned to a double-precision variable be evaluated in double precision.

  In the example below, Compaq Fortran 77 assigns identical values to D1 and D2, whereas Compaq Fortran obeys the standard and assigns a less precise value to D1.

  For example:

```
REAL*8 D1,D2
DATA D1 /2.71828182846182/    ! Incorrect - only REAL*4 value
DATA D2 /2.71828182846182D0/  ! Correct - REAL*8 value
```

- The names of intrinsics introduced by Compaq Fortran may conflict with the names of existing external procedures if the procedures were not specified in an EXTERNAL declaration. For example:

```
EXTERNAL SUM
REAL A(10),B(10)
S = SUM(A)           ! Correct - invokes external function
T = DOT_PRODUCT(A,B) ! Incorrect - invokes intrinsic function
```

- When writing namelist external records, Compaq Fortran uses the syntax for namelist external records specified by the Fortran 95/90 standards, rather than the Compaq Fortran 77 syntax (an extension to the FORTRAN-77 and Fortran 95/90 standards).

  Consider the following program:

```
% cat test.f

INTEGER I
NAMELIST /N/ I
I = 5
PRINT N
END
```

  When this program is compiled by the `f90` command and run, the following output appears:

```
% f90 test.f
% a.out
&N
I     =      5
/
```

  When this program is compiled by the `f77` command and run, the following output appears:

```
% f77 test.f
% a.out
$N
I     =      5
$END
```

  Use the `-f77rtl` option to tell Compaq Fortran to generate NAMELIST output in Compaq Fortran 77 format.

  Compaq Fortran accepts Fortran 95/90 namelist syntax and Compaq Fortran 77 namelist syntax for reading records.

- The Compaq Fortran language does not include C-style escape sequences in standard char constants. For example:

```
CHARACTER NL
NL = '\n'              ! Incorrect
NL = CHAR(10)          ! Correct
```

The Compaq Fortran extension C-string allows certain C-style escape sequences in char constants that end with a C. For example:

```
CHAR NL
NL = '\n'C
```

- Compaq Fortran inserts a leading blank when doing list-directed I/O to an internal file. For example:

```
CHARACTER*10 C
WRITE(C,*) 'FOO'    ! C = ' FOO'
```

- Compaq Fortran 77 and Compaq Fortran produce different output a real value whose data magnitude is 0 with a G field descriptor. For example:

```
     X = 0.0
     WRITE(*,100) X     ! Compaq Fortran 77 prints 0.0000E+00
100  FORMAT(G12.4)      ! Compaq Fortran prints 0.000
```

- Compaq Fortran does not allow certain intrinsics (such as SQRT) in constant expressions for array bounds. For example:

```
REAL A(SQRT(31.5))
END
```

- Compaq Fortran 77 returns UNKNOWN while Compaq Fortran returns UNDEFINED when the ACCESS, BLANK, and FORM characteristics can not be determined. For example:

```
INQUIRE(20,ACCESS=acc,BLANK=blk,FORM=form)
```

- Compaq Fortran does not allow an extraneous parenthesis in I/O lists. For example:

```
write(*,*) ((i,i=1,1),(j,j=1,2))
```

- Compaq Fortran does not allow control characters within quoted strings. For example, the assignment statement in the following program is incorrect because it contains the character Ctrl/C.

```
character*5 c
c = 'ab^cef'
end
```

- Compaq Fortran does not recognize certain hexadecimal and octal constants in DATA statements, such as those used in the following program:

```
INTEGER I, J
DATA I/O20101/, J/Z20/
TYPE *, I, J
END
```

- Compaq Fortran, like Compaq Fortran 77, supports the use of character literal constants (such as 'ABC' or "ABC") in numeric contexts, where they are treated as Hollerith constants.

  Compaq Fortran 77 also allows character PARAMETER constants (typed and untyped) and character constant expressions (using the // operator) in numeric constants as an undocumented extension.

  Compaq Fortran does allow character PARAMETER constants in numeric contexts, but does not allow character expressions. For example, the following is valid for Compaq Fortran 77, but will result in an error message from Compaq Fortran:

  ```
  REAL*8 R
  R = 'abc' // 'def'
  WRITE (5,*) R
  END
  ```

  Compaq Fortran does allow PARAMETER constants:

  ```
  PARAMETER abcdef = 'abc' // 'def'
  REAL*8 R
  R = abcdef
  WRITE (5,*) R
  END
  ```

- Compaq Fortran 77 namelist output formats character data delimited with apostrophes. For example, consider:

  ```
  CHARACTER CHAR4*4
  NAMELIST /CN100/ CHAR4

  CHAR4 = 'ABCD'
  WRITE(20,CN100)
  CLOSE (20)
  ```

  This produces the following output file:

  ```
  $CN100
  CHAR4   = 'ABCD'
  $END
  ```

  This file is read by:

  ```
  READ (20, CN100)
  ```

  In contrast, Compaq Fortran produces the following output file by default:

  ```
  &CN100
  CHAR4   = ABCD
  /
  ```

When read, this generates a `syntax error in NAMELIST input` error. To produce delimited strings from namelist output that can be read by namelist input, use `DELIM="'"` in the OPEN statement of a Compaq Fortran program.

For More Information:

- On argument passing between Compaq Fortran and Compaq Fortran 77 for Compaq Tru64 UNIX systems, see Section A.5.
- On compatibility between Compaq Fortran for Compaq Tru64 UNIX or Linux Alpha systems and Compaq Fortran on OpenVMS systems, see Section A.4.
- On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.2.3 Improved Compaq Fortran Compiler Diagnostic Detection

The following language features are detected or interpreted differently by Compaq Fortran and Compaq Fortran 77:

- The Compaq Fortran compiler enforces the constraint that the "nlist" in an EQUIVALENCE statement must contain at least two variables. For example:

```
EQUIVALENCE (X)     ! Incorrect
EQUIVALENCE (Y,Z)   ! Correct
```

- The Compaq Fortran compiler enforces the constraint that entry points in a SUBROUTINE must not be typed. For example:

```
SUBROUTINE ABCXYZ(I)
  REAL ABC
  I = I + 1
  RETURN
  ENTRY ABC       ! Incorrect
  BAR = I + 1
  RETURN
  ENTRY XYZ       ! Correct
  I = I + 2
  RETURN
END SUBROUTINE
```

- The Compaq Fortran compiler enforces the constraint that a type must appear before each list in an IMPLICIT statement. For example:

```
IMPLICIT REAL (A-C), (D-H)        ! Incorrect
IMPLICIT REAL (O-S), REAL (T-Z)   ! Correct
```

- The Compaq Fortran language disallows passing mismatched actual arguments to intrinsics with corresponding integer formal arguments. For example:

```
R = REAL(.TRUE.)    ! Incorrect
R = REAL(1)         ! Correct
```

- The Compaq Fortran compiler enforces the constraint that a simple list element in an I/O list must be a variable or an expression. For example:

```
READ (10,100) (I,J,K)   ! Incorrect
READ (10,100) I,J,K     ! Correct
```

- The Compaq Fortran compiler enforces the constraint that if two operators are consecutive, the second operator must be a plus or a minus. For example:

```
I = J -.NOT.K           ! Incorrect
I = J - (.NOT.K)        ! Correct
```

- The Compaq Fortran compiler enforces the constraint that character entities with a length greater than 1 cannot be initialized with a bit constant in a DATA statement. For example:

```
CHARACTER*1 C1
CHARACTER*4 C4
DATA C1/'FF'X/            ! Correct
DATA C4/'FFFFFFFF'X/      ! Incorrect
```

- The Compaq Fortran compiler enforces the requirement that edit descriptors in the FORMAT statement must be followed by a comma or slash separator. For example:

```
1  FORMAT (SSF4.1)       ! Incorrect
2  FORMAT (SS,F4.1)      ! Correct
```

- The Compaq Fortran compiler enforces the constraint that the number and types of actual and formal statement function arguments must match (such as incorrect number of arguments). For example:

```
CHARACTER*4 C,C4,FUNC
FUNC()=C4
C=FUNC(1)               ! Incorrect
C=FUNC()                ! Correct
```

- The Compaq Fortran compiler detects the use of a format of the form Ew.dE0 at compile time. For example:

```
1   format(e16.8e0)   ! Compaq Fortran detects error at compile time
    write(*,1) 5.0    ! Compaq Fortran 77 compiles but an output
                      !    conversion error occurs at run time
```

- Compaq Fortran detects passing of a statement function to a routine. For example:

```
foo(x) = x * 2
call bar(foo)
end
```

- The Compaq Fortran compiler enforces the constraint that a branch to a statement shared by more than one DO statements must occur from within the innermost loop. For example:

```
   DO 10 I = 1,10
      IF (L1) GO TO 10      ! Incorrect
      DO 10 J = 1,10
         IF (L2) GO TO 10    ! Correct
10 CONTINUE
```

- The Compaq Fortran compiler enforces the constraint that a file must contain at least one program unit. For example, a source file containing only comment lines results in an error at the last line (end-of-file).

  The Compaq Fortran 77 compiler compiles files containing less than one program unit.

- The Compaq Fortran compiler correctly detects misspellings of the ASSOCIATEVARIABLE keyword to the OPEN statement. For example:

```
OPEN(1,ASSOCIATEVARIABLE = I)     ! Correct
OPEN(2,ASSOCIATEDVARIABLE = J)    ! Incorrect (extra D)
```

- The Compaq Fortran language enforces the constraint that the result of an operation is determined by the data types of its operands. For example:

```
INTEGER*8 I8
I8 = 2147483647 + 1      ! Incorrect. Produces less accurate
                         ! INTEGER*4 result
I8 = 2147483647_8 + 1_8  ! Correct
```

- The Compaq Fortran compiler enforces the constraint that an object can be typed only once. Compaq Fortran 77 issues a warning message and uses the first type. For example:

```
LOGICAL B,B             ! Incorrect (B multiply declared)
```

- The Compaq Fortran compiler enforces the constraint that certain intrinsic procedures defined by the Fortran 95/90 standards cannot be passed as actual arguments. For example, Compaq Fortran 77 allows most intrinsic procedures to be passed as actual arguments, but the Compaq Fortran compiler only allows those defined by the Fortran 95/90 standards (issues an error message).

Consider the following program:

```
program tstifx

intrinsic ifix,int,sin

call a(ifix)
call a(int)
call a(sin)
stop
end

subroutine a(f)
external f
integer f
print *, f(4.9)
return
end
```

The IFIX and INT intrinsic procedures cannot be passed as actual arguments (the compiler issues an error message). However, the SIN intrinsic is allowed to be passed as an actual argument by the Fortran 95/90 standards.

- Compaq Fortran reports character truncation with an error-level message, not as a warning.

  The following program produces an error message during compilation with Compaq Fortran, whereas Compaq Fortran 77 produces a warning message:

```
    INIT5 = 'ABCDE'
    INIT4 = 'ABCD'
    INITLONG = 'ABCDEFGHIJKLMNOP'
    PRINT 10, INIT5, INIT4, INITLONG
10  FORMAT (' ALL 3 VALUES SHOULD BE THE SAME: ' 3I)
    END
```

- If your code invokes Compaq Fortran intrinsic procedures with the wrong number of arguments or an incorrect argument type, Compaq Fortran reports this with an error-level message, not with a warning. Possible causes include:

  – A Compaq Fortran intrinsic has been added with the same name as a user-defined subprogram and the user-defined subprogram needs to be declared as EXTERNAL.

  – An intrinsic that is an extension to an older Fortran standard is incompatible with a newer standard-conforming intrinsic (for example, the older RAN function that accepted two arguments).

The following program produces an error message during compilation with Compaq Fortran, whereas Compaq Fortran 77 produces a warning message:

```
      INTEGER ANOTHERCOUNT
      ICOUNT=0
100   write(6,105) (ANOTHERCOUNT(ICOUNT), INT1=1,10)
105   FORMAT(' correct if print integer values 1 through 10' /10I7)
      Q = 1.
      R = .23
      S = SIN(Q,R)
      WRITE (6,110) S
110   FORMAT(' CORRECT = 1.23   RESULT = ',f8.2)
      END
!
      INTEGER FUNCTION ANOTHERCOUNT(ICOUNT)
      ICOUNT=ICOUNT+1
      ANOTHERCOUNT=ICOUNT
      RETURN
      END

      REAL FUNCTION SIN(FIRST, SECOND)
      SIN = FIRST + SECOND
      RETURN
      END
```

- Compaq Fortran reports missing commas in FORMAT descriptors with an error-level message, not as a warning.

  The following program produces an error message during compilation with Compaq Fortran, whereas Compaq Fortran 77 produces a warning message:

```
    LOGICAL LOG/111/
    TYPE 1,LOG
1   FORMAT(' '23X,'LOG='O12)
    END
```

  In the preceding example, the correct coding (adding the missing comma) for the FORMAT statement is:

```
1   FORMAT(' ',23X,'LOG='O12)
```

- Compaq Fortran generates an error when it encounters a 1-character source line containing a Ctrl/Z character, whereas Compaq Fortran 77 allows such a line (which is treated as a blank line).

- Compaq Fortran detects the use of a character variable within parentheses in an I/O statement. For example:

```
CHARACTER*10 CH/'(I5)'/
INTEGER I

READ CH,I    ! Acceptable

READ (CH),I  ! Generates error message, interpreted as an internal READ

END
```

- Compaq Fortran evaluates the exponentiation operator at compile time only if the exponent has an integer data type. Compaq Fortran 77 evaluates the exponentiation operator even when the exponent does not have an integer data type. For example:

```
PARAMETER ( X = 4.0 ** 1.1)
```

- Compaq Fortran detects an error when evaluating constants expressions that result in an NaN or Infinity exceptional value, while Compaq Fortran 77 allows such expressions. For example:

```
PARAMETER ( X = 4.0 / 0.0 )
```

- Compaq Fortran reports a warning error message when the same variable is initialized more than once. Compaq Fortran 77 allows multiple initializations of the same variable without a warning. For example:

```
integer i
data i /1/
data i /2/
write (*,*) i
stop
end
```

For More Information:

- On passing arguments and returning function values between Compaq Fortran and Compaq Fortran 77, see Section A.5.
- On Compaq Fortran procedure calling and argument passing, see Section 11.1.
- On compatibility between Compaq Fortran for Compaq Tru64 UNIX systems and Compaq Fortran 77 on OpenVMS systems, see Section A.4.
- On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.2.4 Compiler Command-Line Differences

Compaq Fortran 77 (`f77` command) and Compaq Fortran (`f90` command) share most of the same command-line options. The following options are provided only by Compaq Fortran 77 (not by Compaq Fortran):

- `-assume backslash`
- `-f77`
- `-ident`
- `-show xref` (same as `-cross_reference`)
- `-stand` *keyword*
- `-warn informational`
- `-warn nounreachable`

The following options are provided only by Compaq Fortran (not by Compaq Fortran 77):

- `-align recNbyte`
- `-annotations`
- `-assume buffered_io`
- `-assume gfullpath`
- `-assume minus0`
- `-f77rtl`
- `-fixed`
- `-free`
- `-fpconstant`
- `-fuse_xref` *(TU*X only)*
- `-hpf` *(TU*X only)* and associated HPF parallel options, including `-assume bigarrays`, `-assume nozsize`, `-nearest_neighbor`, `-nohpf_main`, `-show hpf`, and `-warn hpf`
- `-intconstant`
- `-ladebug`
- `-module`
- `-mp` *(TU*X only)*

- `-omp` *(TU*X only)* and assorted OpenMP parallel options including `-assume pthreads_lock` and `-check omp_bindings`
- `-std` (performs Fortran 95/90 standards checking, whereas the Compaq Fortran 77 `-stand` *keyword* performs FORTRAN 77 and NTT MIA standards checking)
- `-warn granularity`

The `f77` command by default executes the Compaq Fortran 90 compiler and uses the various `DECF90_` environment variables. To execute the Compaq Fortran 77 compiler, use the `f77` command with the `-old_f77` option. This option must be the first text on the command line after `f77`.

A.3 Language Compatibility with Compaq Visual Fortran

The following language features found in Compaq Visual Fortran (and Microsoft Fortran Powerstation Version 4) are now supported by Compaq Fortran:

- # Constants. Constants using a base other than 10.
- C Strings. NULL terminated strings contain C-style escape sequences.
- Conditional Compilation And Metacommand Expressions ($define, $undefine, $if, $elseif, $else, $endif).
- $FREEFORM, $NOFREEFORM, $FIXEDFORM. Source file format.
- $INTEGER, $REAL. Selects size.
- $FIXEDFORMLINESIZE. Line length for fixed form source.
- $STRICT, $NOSTRICT. F90 conformance.
- $PACK. Structure packing.
- $ATTRIBUTES ALIAS. External name for a subprogram or common block.
- $ATTRIBUTES C, STDCALL. Calling and naming conventions.
- $ATTRIBUTES VALUE, REFERENCE. Calling conventions.
- \ Descriptor. Prevents writing an end-of-record mark.
- Ew.dDe and Gw.dDe Edit Descriptors. Similar to Ew.dEe and Gw.dEe.
- 7200 Character Statement Length.
- Free form infinite line length.
- $DECLARE and $NODECLARE == IMPLICIT NONE.

- $ATTRIBUTES EXTERN. Variable allocated in another source file.
- $ATTRIBUTES VARYING. Variable number of arguments.
- $ATTRIBUTES ALLOCATABLE. Allocatable array.
- Mixing Subroutines/Functions in Generic Interfaces.
- $MESSAGE. Output message during compilation.
- $LINE == C's #line.
- INT1. Converts to one byte integer by truncating.
- INT2. Converts to two byte integer by truncating.
- INT4. Converts to four byte integer by truncating.
- COTAN. Returns cotangent.
- DCOTAN. Returns double precision cotangent.
- IMAG. Returns the imaginary part of complex number.
- IBCHNG. Reverses value of bit.
- ISHA. Shifts arithmetically left or right.
- ISHC. Performs a circular shift.
- ISHL. Shifts logically left or right.

A.4 Compatibility with Compaq Fortran 77 and Compaq Fortran for OpenVMS Systems

This section provides compatibility information for those who:

- Port Compaq Fortran 77 and Compaq Fortran applications from OpenVMS systems to Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha Systems
- Design Compaq Fortran applications to run on multiple platforms, including OpenVMS and Compaq Tru64 UNIX or Linux Alpha systems

If your primary concern is the design and development of Compaq Fortran applications for only Compaq Tru64 UNIX (or other U*X) systems, consider skipping this section.

This section discusses the following topics:

- Compaq Fortran 77 extensions for OpenVMS systems that are not supported by this version of Compaq Fortran 77 or Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha Systems (Section A.4.1)
- Porting Compaq Fortran data files from an OpenVMS system to a Compaq Tru64 UNIX or Linux Alpha system (Section A.4.2)
- Nonnative VAX floating-point representations, provided for those converting unformatted OpenVMS floating-point data (Section A.4.3)

A.4.1 Language Features Specific to Compaq Fortran 77 and Compaq Fortran for OpenVMS Systems

Some extensions to the FORTRAN-77 standard provided by Compaq Fortran 77 and Compaq Fortran for OpenVMS Systems are specific to the OpenVMS operating system, VAX architecture, or certain products on OpenVMS systems. Such extensions are not included in Compaq Fortran 77 or Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha Systems.

For information on language compatibility between Compaq Fortran and Compaq Fortran 77 without regard to operating system or architecture differences, see Section A.2.

Compaq Fortran 77 and Compaq Fortran products for OpenVMS systems include:

- Compaq Fortran Version 7.4 for OpenVMS Alpha Systems
- Compaq Fortran 77 Version 7.4 for OpenVMS Alpha Systems
- Compaq Fortran 77 Version 6.6 for OpenVMS VAX Systems (previously called VAX FORTRAN)

Unless otherwise noted, the following list describes the Compaq Fortran 77 extensions in Compaq Fortran 77 and/or Compaq Fortran for OpenVMS systems that are *not* supported by Compaq Fortran for Compaq Tru64 UNIX Systems:

- DICTIONARY statement

  The DICTIONARY and related support for the CDD/Repository (common data dictionary) product are not provided by Compaq Fortran or Compaq Fortran 77 for Compaq Tru64 UNIX Systems.
- Support for indexed sequential files

I/O statement specifiers for indexed file (keyed access) record I/O using OpenVMS OPEN and INQUIRE statement specifiers are not provided by Compaq Fortran or Compaq Fortran 77 for Compaq Tru64 UNIX or Linux Systems, as follows:

ACCESS='KEYED'	EXTENDSIZE
INITIALSIZE	KEY
NOSPANBLOCKS	ORGANIZATION='INDEXED'
SHARED	

- FORSYSDEF symbol definitions for OpenVMS systems

  The parameter definitions of run-time messages found in FORSYSDEF.TLB library module FORIOSDEF on OpenVMS systems are provided in the file `/usr/include/foriosdef.f` (see Section 8.2.2) on Compaq Tru64 UNIX Systems. On Compaq Tru64 UNIX and Linux Alpha systems, Compaq Fortran and Compaq Fortran 77 provides jacket routines to simplify calling system calls and library routines (see Chapter 12).

- The INCLUDE statement option of including text from text libraries.

  On Compaq Tru64 UNIX and Linux Alpha systems, OpenVMS text libraries are not supported.

- The %DESCR built-in function (for OpenVMS character descriptors).

  On Compaq Tru64 UNIX and Linux Alpha systems, character data is passed by address and hidden length. For information about calling or being called by procedures written in other languages, see Chapter 11.

- Run-time default I/O units spelled as FOR0nn.dat, SYS$INPUT, and so on

  In Compaq Fortran and Compaq Fortran 77 on Compaq Tru64 UNIX and Linux Alpha systems, these are environment variables FORT*n*, `stdin`, `stdout`, and so forth (see Section 7.5).

- VAX floating-point formats and related selection of the floating-point format in memory

  Only IEEE floating-point formats are supported in memory on Compaq Tru64 UNIX Alpha systems. (Compaq proprietary VAX floating-point formats are not supported in memory.) You can request conversion of unformatted files containing VAX floating-point formats into the appropriate IEEE memory format during record I/O (see Chapter 10).

  On OpenVMS VAX systems, you specify the floating-point format to be used in memory with either the option [NO]G_FLOATING in the OPTIONS statement or the qualifier /[NO]G_FLOATING on the FORTRAN command line.

On OpenVMS Alpha systems, you specify the floating-point format to be used in memory using the /FLOAT qualifier on the FORTRAN command line.

- Stream record format differences

  With Compaq Fortran 77 and Compaq Fortran for OpenVMS systems, the Stream record type is delimited by CR-LF character sequence (carriage control and line feed characters). In Compaq Fortran for Compaq Tru64 UNIX systems, the Stream record type uses no delimiters.

  For more information on compatible record types, see Section A.4.2.

- Other differences related to the OpenVMS operating system and the Compaq Tru64 UNIX and Linux operating systems

  When parsing file specifications for the OPEN, INQUIRE, and INCLUDE statements, keep in mind that file names are case-sensitive on Compaq Tru64 UNIX and Linux systems and that OpenVMS file specification syntax differs from pathname syntax.

  For the INCLUDE statement, the network node names (terminated by "::"), logical names (usually terminated by ":"), and other OpenVMS file specification components are not recognized. Instead, the INCLUDE statement should specify a pathname, possibly with an absolute directory path.

- The OpenVMS operating system provides various system services (SYS$ prefix) and run-time library routines (LIB$, SMG$, and other prefixes) that are not supported on Compaq Tru64 UNIX and Linux systems. Compaq Tru64 UNIX systems support system calls and library routines with similar functions (but different names).

  To make programs more portable to other operating systems, wherever possible you should use standard-conforming Compaq Fortran intrinsic routines in place of routines specific to a particular operating system.

  For more information on specifying files, see Section 7.5.3.

The following language and VAX architecture features are associated only with Compaq Fortran 77 on OpenVMS VAX Systems (previously called VAX FORTRAN) and are not supported by Compaq Fortran for Compaq Tru64 UNIX and Compaq Fortran for Linux Alpha systems:

- Directed decomposition features and CPAR$ directives for parallel processing

  CPAR$ directives are treated as comments (ignored). Parallel processing capabilities (appropriate `f90` options, OpenMP, Compaq Fortran parallel, and HPF data mapping directives) are provided by Compaq Fortran.

- OPTIONS statement options /BLAS, /NOBLAS, /CHECK=ALIGNMENT, /CHECK=NOALIGNMENT, /CHECK=ASSERTION, /CHECK=NOASSERTION, /G_FLOAT, and /NOG_FLOAT

  You can specify some of these options by using the corresponding `f90` command-line options. The OPTIONS statement is treated as a comment (ignored).

- CDEC$ performance directives ASSERT and NOVECTOR are treated as comments (ignored).

- The REAL*16 floating-point data type

  On VAX systems, REAL*16 data is in H_float format. On Alpha systems, REAL*16 data is in the native IEEE style X_float; so are both halves (real and imaginary) of COMPLEX*32 data.

- The following subroutines for PDP-11 compatibility:

ASSIGN	ERRTST	RAD50
CLOSE	FDBSET	R50ASC
ERRSET	IRAD50	USEREX

- Radix-50 constants and character set

- The BLAS routines

  Similar basic linear algebra routines are provided in the Compaq Extended Mathematical Library (CXML) product (see Section 5.1.1).

The following language and VAX architecture features are interpretation differences between Compaq Fortran and Compaq Fortran 77 on Alpha systems and Compaq Fortran 77 on OpenVMS VAX Systems (previously called VAX FORTRAN):

- Random number generator (RAN)

  The RAN function (one argument) generates a different pattern of numbers in Compaq Fortran than in Compaq Fortran 77 on OpenVMS VAX Systems for the same random seed. Compaq Fortran and Compaq Fortran 77 use the same random seed. (The RAN and RANDU functions are provided for Compaq Fortran 77 on OpenVMS VAX Systems compatibility. See *Compaq Fortran Language Reference Manual*.)

- Hollerith constants in formatted I/O statements

  Compaq Fortran 77 on OpenVMS VAX Systems and Compaq Fortran behave differently if either of the following occurs:

  – Two different I/O statements refer to the same CHARACTER PARAMETER constant as their format specifier. For example:

```
CHARACTER*(*) FMT2
PARAMETER (FMT2='(10Habcdefghij)')
READ (5, FMT2)
WRITE (6, FMT2)
```

– Two different I/O statements use the identical character constant as their format specifier. For example:

```
READ (5, '(10Habcdefghij)')
WRITE (6, '(10Habcdefghij)')
```

In Compaq Fortran 77 for OpenVMS VAX Systems, the parameter value obtained by the READ statement is modified. The parameter value modified by the READ statement is used as the output of the WRITE statement (FMT2 is ignored). However, in Compaq Fortran, the parameter value is *not* modified (the parameter value read by the READ statement has no effect on the parameter value written by the WRITE statement.)

For More Information:

- On language compatibility information about Compaq Fortran for Compaq Tru64 UNIX systems and Compaq Fortran 77, see Section A.2.2.
- On language interpretation differences between Compaq Fortran for Compaq Tru64 UNIX systems and Compaq Fortran 77, see Section A.2.3.
- About the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.4.2 OpenVMS Data Porting Considerations

When porting data between systems running the Compaq Tru64 UNIX and Linux Alpha operating systems and systems running the OpenVMS operating system, the file formats and the floating-point representations may differ.

The file and record formats of Compaq Fortran 77 on Compaq Tru64 UNIX systems are compatible with Compaq Fortran on Compaq Tru64 UNIX and Linux Alpha systems; they share the same language run-time I/O environment (see Chapter 7).

OpenVMS Fortran[1] files containing only character, integer, or logical data do not need field-by-field conversion, but the record types must match. The segmented record type is the same on OpenVMS Fortran systems and Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha systems. Certain other record types, such as variable-length records, differ between OpenVMS systems and Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha systems.

[1] OpenVMS Fortran refers collectively to VAX FORTRAN, Compaq Fortran 77 for OpenVMS Alpha Systems, and Compaq Fortran 77 for OpenVMS VAX Systems

Table A–2 summarizes the OpenVMS Fortran record types and their equivalent record types in Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha systems.

Table A–2 Equivalent Record Types for OpenVMS Fortran and Compaq Fortran on Compaq Tru64 UNIX or Linux Alpha Systems

OpenVMS Fortran Record Type	Compaq Tru64 UNIX Fortran Record Type	Comments
Fixed-length	None	Equivalent (must be copied correctly) if you use sequential access and you specify the `-vms` option when compiling the Compaq Fortran file. Otherwise, convert the file to a different record type.
Variable-length	None	Not equivalent. Convert the file to a different record type.
Segmented	Segmented	Equivalent (must be copied correctly). Segmented data files can contain formatted or unformatted data.
Stream	None	Not equivalent. Convert the file to a different record type.
Stream_CR	Stream_CR	Equivalent (must be copied correctly).
Stream_LF	Stream_LF	Equivalent (must be copied correctly).

A.4.2.1 Matching Record Types

To match record types, there are several options:

- For the Segmented, Stream_CR, and Stream_LF record types, you do not need to convert the files.
- For fixed-length records where you will only use sequential access, use the `-vms` option when compiling the Compaq Fortran program that will access the OpenVMS Fortran files. For fixed-length records where you will use direct access, convert the files to a different record format.
- For incompatible record types, convert the files by writing a OpenVMS Fortran or C conversion program or by using the ANALYZE/RMS/FDL and CONVERT/FDL (or EXCHANGE/FDL) commands for an appropriate file. For instance, convert the OpenVMS Fortran file to the segmented record type.

A.4.2.2 Copying Files

Equivalent record types must be copied carefully to preserve control information and record characteristics. For example:

- Do not use the ASCII transfer mode for binary files.
- Segmented files must be copied in a manner that preserves record length information.

To transfer (copy) the files, choose one of the following methods:

- From an NFS mounted disk, use the `cp` command (see `cp`(1)).
- Perform a DECnet copy from a Compaq Tru64 UNIX or Linux Alpha system running the appropriate optional network software using `dcp` (see `dcp`(8)). Use the `dcp -i` option when you want to preserve record format information.
- Perform a copy from a Compaq Tru64 UNIX or Linux Alpha system with `rcp`, possibly by using an intermediate node running the appropriate optional network software when using a version of the OpenVMS operating system that does not support a compatible network protocol (optional product).
- Use `ftp` from a Compaq Tru64 UNIX or Linux Alpha system to copy a file between a Compaq Tru64 UNIX or Linux Alpha system and an OpenVMS system. Use the `binary` or `ascii` command to set the mode before you copy (get or put) the file. For example, use the ftp `binary` command before copying an unformatted file (such as the segmented record type).
- Perform a DECnet copy from an OpenVMS system with the EXCHANGE command with the /NETWORK and /TRANSFER=BLOCK qualifiers with a Compaq Tru64 UNIX system. To convert the file to Stream_LF format during the copy operation, use /TRANSFER=(BLOCK,RECORD_SEPARATOR=LF) instead of /TRANSFER=BLOCK, or specify the /FDL qualifier to the EXCHANGE command to specify the record type.

In addition to using the correct record type and carefully transferring the files, the data inside unformatted records may need to be converted. OpenVMS Fortran data files that contain VAX binary floating-point data must be converted before they can be accessed by a Compaq Fortran program. There are several methods:

- On an OpenVMS system, a Fortran program can convert files containing unformatted data to files containing formatted data. Once the files contain formatted data, they can be read by the appropriate Compaq Fortran programs. However, converting unformatted data to formatted data may result in a loss of accuracy for unformatted floating-point data.

- On an OpenVMS VAX system, a Compaq Fortran 77 program can read and write files containing unformatted data by using the Compaq Fortran conversion capabilities described in *DEC Fortran User Manual for OpenVMS VAX Systems*.
- On an OpenVMS Alpha system, a Fortran program can read and write files containing unformatted data by using the Compaq Fortran conversion capabilities described in the *Compaq Fortran User Manual for OpenVMS Alpha Systems*.

  A Compaq Fortran 77 for OpenVMS Alpha Systems program can also use the CVT$CONVERT_FLOAT routine to convert individual floating-point fields.
- On a Compaq Tru64 UNIX or Linux Alpha system, a Compaq Fortran program can read and write files containing unformatted data using the Compaq Fortran conversion capabilities described in Section 10.3. A program using the Compaq Fortran conversion capabilities can also convert such data to other formats.

If you need to convert unformatted floating-point data, keep in mind that Compaq Fortran 77 for OpenVMS VAX programs (VAX hardware) store the following:

- REAL*4 or COMPLEX*8 data in VAX F_float format
- REAL*8 or COMPLEX*16 data in either VAX D_float or G_float format
- REAL*16 data in VAX H_float format

In contrast, Compaq Fortran programs running on the Compaq Tru64 UNIX or Linux Alpha operating system on Alpha hardware store the following:

- REAL*4 or COMPLEX*8 data in IEEE S_float format
- REAL*8 or COMPLEX*16 data in IEEE T_float format
- REAL*16 data or COMPLEX*32 data in native (IEEE style) X_float format

Compaq Fortran 77 and Compaq Fortran for OpenVMS Alpha programs store floating-point data in the format specified by the /FLOAT qualifier:

- REAL*4 or COMPLEX*8 data in VAX F_float or IEEE S_float format
- REAL*8 or COMPLEX*16 data in VAX D_float, VAX G_float, or IEEE T_float format
- REAL*16 data or COMPLEX*32 data in native (IEEE style) X_float format

For information on Compaq Fortran data types, see Chapter 9.

For More Information:

- On Compaq Fortran I/O, see Chapter 7.
- About the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.4.3 Nonnative VAX Floating-Point Representations

This section provides information about VAX floating-point data formats. You can convert unformatted files from OpenVMS systems by using the methods described in Chapter 10.

On OpenVMS VAX systems, single-precision data (such as REAL*4) is stored in VAX F_float format and double-precision data (such as REAL*8) data can be stored in either VAX D_float or VAX G_float formats, depending on whether the /G_FLOATING qualifier was specified on the FORTRAN command line (see the *DEC Fortran User Manual for OpenVMS VAX Systems*).

On OpenVMS Alpha systems, you can specify the floating-point format in memory by using the /FLOAT qualifier (see the *DEC Fortran User Manual for OpenVMS AXP Systems*). Single-precision data on OpenVMS Alpha systems is stored in either VAX F_float or IEEE S_float formats; double-precision data can be stored in VAX D_float, VAX G_float, or IEEE T_float formats.

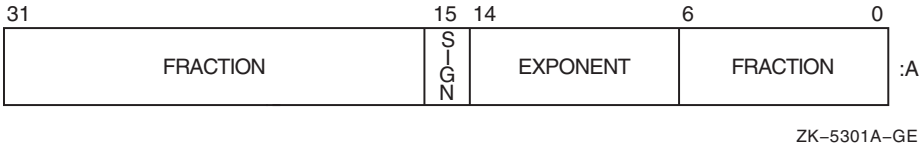
REAL*16 (extended precision) data is always stored in IEEE style X_float format on Alpha systems.

With VAX floating-point data types, the binary radix point is to the left of the most-significant bit.

A.4.3.1 VAX F_float REAL (KIND=4) or REAL*4

Intrinsic REAL (KIND=4) or REAL*4 F_float data occupies four contiguous bytes. Bits are labeled from the right, 0 through 31, as shown in Figure A–1.

Figure A–1 VAX F_float REAL (KIND=4) or REAL*4 Representation



The form of REAL (KIND=4) or REAL*4 F_float data is sign magnitude, where:

- Bit 15 is the sign bit (0 for positive numbers, 1 for negative numbers).

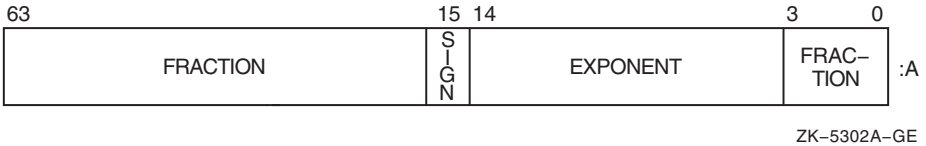
- Bits 14:7 are a binary exponent in excess 128 notation (binary exponents from –127 to 127 are represented by binary 1 to 255).
- Bits 6:0 and 31:16 are a normalized 24-bit fraction with the redundant most significant fraction bit not represented.

When converting unformatted F_float data from an OpenVMS system, the approximate range is 0.293873588E–38 to 1.7014117E38. The precision is approximately one part in 2**23, typically seven decimal digits.

A.4.3.2 VAX G_float REAL (KIND=8) or REAL*8

Intrinsic REAL (KIND=8) or REAL*8 (same as DOUBLE PRECISION) G_float data occupies eight contiguous bytes. The bits are labeled from the right, 0 through 63, as shown in Figure A–2.

Figure A–2 VAX G_float REAL (KIND=8) or REAL*8 Representation



The form of REAL (KIND=8) or REAL*8 G_float data is sign magnitude, where:

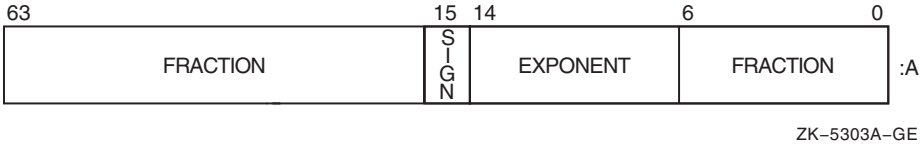
- Bit 15 is the sign bit (0 for positive numbers, 1 for negative numbers).
- Bits 14:4 are a binary exponent in excess 1024 notation (binary exponents from –1023 to 1023 are represented by the binary 1 to 2047).
- Bits 3:0 and 63:16 are a normalized 53-bit fraction with the redundant most significant fraction bit not represented.

When converting unformatted G_float data from an OpenVMS system, the approximate range is 0.5562684646268004D–308 to 0.8988465674311578540 7D308. The precision of G_float data is approximately one part in 2**52, typically 15 decimal digits.

A.4.3.3 VAX D_float REAL (KIND=8) or REAL*8

Intrinsic REAL (KIND=8) or REAL*8 (same as DOUBLE PRECISION) D_float data occupies eight contiguous bytes. Bits are labeled from the right, 0 through 63, as shown in Figure A–3.

Figure A–3 VAX D_float REAL (KIND=8) or REAL*8 Representation



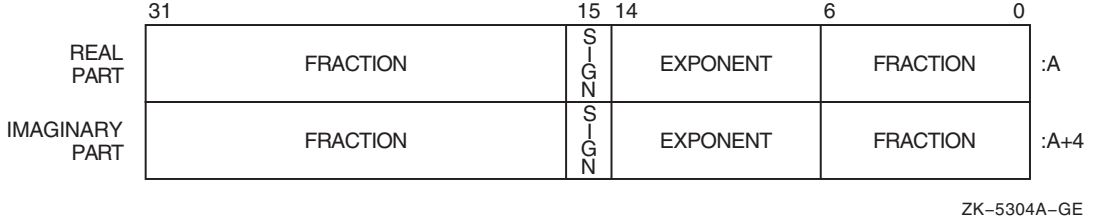
The form of REAL (KIND=8) or REAL*8 D_float data is identical to an F_float real number, except for an additional 32 low-significance fraction bits. The exponent conventions and approximate range of values are the similar to those for F_float.

When converting unformatted D_float data from an OpenVMS system, the approximate range is 0.2938735877055719D–38 to 1.70141183460469229D38. The precision is approximately one part in 2**55, typically 16 decimal digits.

A.4.3.4 VAX F_float COMPLEX (KIND=4) or COMPLEX*8

Intrinsic COMPLEX (KIND=4) or COMPLEX*8 (single-precision COMPLEX) data in VAX F_float format occupies eight contiguous bytes containing a pair of REAL*4 values. The low-order four bytes contain REAL*4 data that represents the real part of the complex number. The high-order four bytes contain REAL (KIND=4) or REAL*4 data that represents the imaginary part of the complex number. Figure A–4 shows a COMPLEX*8 number in F_float format.

Figure A–4 VAX F_float COMPLEX (KIND=4) or COMPLEX*8 Representation



The limits for REAL (KIND=4) or REAL*4 apply to the two separate real and imaginary parts of a COMPLEX (KIND=4) or COMPLEX*8 number. Like REAL (KIND=4) or REAL*4 numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

A.4.3.5 VAX G_float and D_float COMPLEX (KIND=8) or COMPLEX*16

Intrinsic COMPLEX (KIND=8) or COMPLEX*16 (same as DOUBLE COMPLEX) data occupies 16 contiguous bytes containing a pair of REAL*8 or REAL (KIND=8) values. COMPLEX (KIND=8) or COMPLEX*16 data from an OpenVMS system is in one of the following REAL*8 or REAL (KIND=8) formats:

- VAX G_float format
- VAX D_float format

The low-order eight bytes contain REAL (KIND=8) or REAL*8 data that represents the real part of the complex data. The high-order eight bytes contain REAL (KIND=8) or REAL*8 data that represents the imaginary part of the complex data, as shown in Figure A–5 (for G_float) and Figure A–6 (for D_float).

Figure A–5 VAX G_float COMPLEX (KIND=8) or COMPLEX*16 Representation

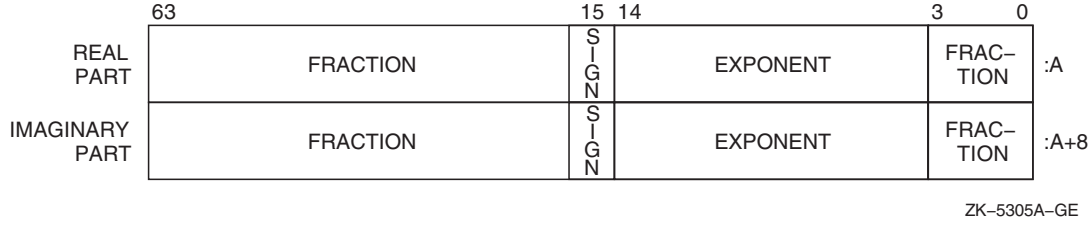
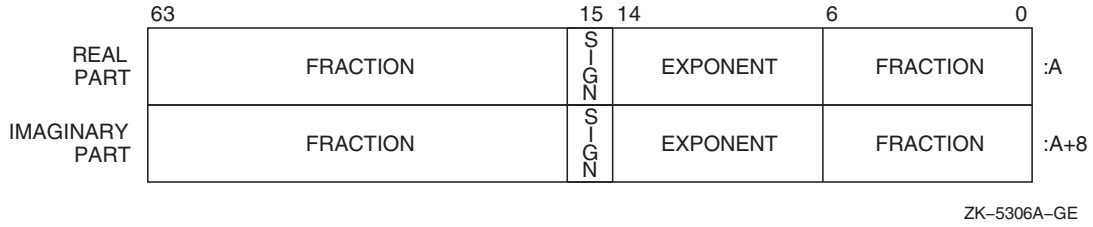


Figure A–6 VAX D_float COMPLEX (KIND=8) or COMPLEX*16 Representation



The limits for REAL (KIND=8) or REAL*8 apply to the two separate real and imaginary parts of a COMPLEX (KIND=8) or COMPLEX*16 number. Like REAL (KIND=8) or REAL*8 numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

A.4.3.6 VAX H_float Representation

The REAL (KIND=16) or REAL*16 VAX H_float data format is used only on OpenVMS VAX systems. On Alpha systems, REAL (KIND=16) extended precision data is always stored in Alpha X_float format.

With VAX floating-point data types, the binary radix point is to the left of the most-significant bit.

As shown in Figure A–7, REAL*16 H_float data is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right, 0 through 127.

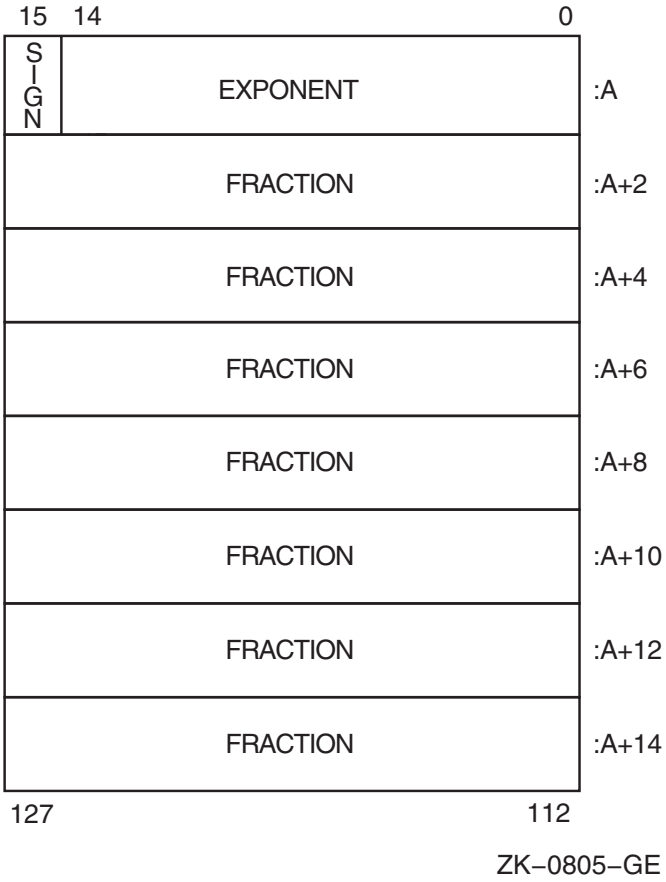
The form of an H_float REAL*16 data is sign magnitude with bit 15 the sign bit, bits 14:0 an excess 16384 binary exponent, and bits 127:16 a normalized 113-bit fraction with the redundant most significant fraction bit not represented.

The value of H_float data is in the approximate range 0.84*10**–4932 through 0.59*10**4932. The precision of H_float data is approximately one part in 2**112 or typically 33 decimal digits.

For More Information:

- On converting unformatted data files, see Chapter 10.
- On native floating-point ranges, see Table 9–1.

Figure A–7 VAX H_float REAL*16 Representation (VAX Systems)



A.5 Calling Between Compaq Fortran 77 and Compaq Fortran

On Compaq Tru64 UNIX systems, you can call a Compaq Fortran 77 subprogram from Compaq Fortran or call a Compaq Fortran subprogram from Compaq Fortran 77 (with a few exceptions). A Compaq Fortran 77 procedure and a Compaq Fortran procedure can also perform I/O to the same unit number.

A.5.1 Argument Passing and Function Return Values

The recommended rules for passing arguments and function return values between Compaq Fortran 77 and Compaq Fortran procedures are as follows:

- If possible, express the following Compaq Fortran features with the Compaq Fortran 77 language:
  - Function references
  - CALL statements
  - Function definitions
  - Subroutine definitions

  Avoid using Compaq Fortran language features not available in Compaq Fortran 77. Since Compaq Fortran is a superset of Compaq Fortran 77, specifying the procedure interface using the Compaq Fortran 77 language helps ensure that calls between the two languages will succeed.
- Not all data types in Compaq Fortran have equivalent Compaq Fortran 77 data types. The following Compaq Fortran features should not be used between Compaq Fortran and Compaq Fortran 77 procedures, because they are not supported by Compaq Fortran 77:
  - COMPLEX*32 data
  - Derived-type (user-defined) data, which has no equivalent in Compaq Fortran 77.
  - Compaq Fortran data with the POINTER attribute, which has no equivalent in Compaq Fortran 77. The pointer data type supported by Compaq Fortran 77 is not equivalent to Compaq Fortran pointer data.

    Because Compaq Fortran supports the pointer data type supported by Compaq Fortran 77, you can use Compaq Fortran 77 pointer data types in both Compaq Fortran and Compaq Fortran 77. (In some cases, you can create Compaq Fortran 77 pointer data in a Compaq Fortran procedure using the %LOC function.)

    Compaq Fortran arrays with the POINTER attribute are passed by array descriptor. A program written in Compaq Fortran 77 needs to interpret the array descriptor format generated by a Compaq Fortran 90 array with the POINTER attribute (see Section 11.1.7).
  - Compaq Fortran assumed-shape arrays.

Compaq Fortran assumed-shape arrays are passed by array descriptor. A program written in Compaq Fortran 77 needs to interpret the array descriptor format generated by a Compaq Fortran assumed-shape array (see Section 11.1.7).

You can use Compaq Fortran record structures, which are supported by Compaq Fortran 77 and Compaq Fortran as an extension to the Fortran 95/90 standards.

For more information on how Compaq Fortran handles arguments and function return values, see Section 11.1.4.

- Make sure the sizes of INTEGER, LOGICAL, REAL, and COMPLEX declarations match.

  For example, Compaq Fortran declarations of REAL (KIND=4) and INTEGER (KIND=4) match Compaq Fortran 77 declarations of REAL*4 and INTEGER*4. For COMPLEX values, a Compaq Fortran declaration of COMPLEX (KIND=4) matches a Compaq Fortran 77 declaration of COMPLEX*8; COMPLEX (KIND=8) matches COMPLEX*16. Compaq Fortran 77 does not have COMPLEX*32 declarations.

  Your source programs may contain INTEGER, LOGICAL, REAL, or COMPLEX declarations without a kind parameter (or size specifier). In this case, when compiling the Compaq Fortran procedures (`f90` command) and Compaq Fortran 77 procedures (`f77` command), either omit the options or specify the equivalent options for controlling the sizes of these declarations.

  For more information on these options (the same for `f90` and `f77`), see Section 3.53 for INTEGER and LOGICAL declarations, Section 3.78 for REAL and COMPLEX declarations, and Section 3.34 for DOUBLE PRECISION declarations.

- Compaq Fortran uses the same argument-passing conventions as Compaq Fortran 77 on Compaq Tru64 UNIX systems (see Section 11.1.4).

- You can return nearly all function return values from a Compaq Fortran function to a calling Compaq Fortran 77 routine, with the following exceptions:

  - You cannot return Compaq Fortran pointer data from Compaq Fortran to a Compaq Fortran 77 calling routine.

  - You cannot return Compaq Fortran user-defined data types from a Compaq Fortran function to a Compaq Fortran 77 calling routine.

Example A–1 and Example A–2 show passing an array from a Compaq Fortran program to a Compaq Fortran 77 subroutine that prints its value.

Example A–1 shows the Compaq Fortran program (file array_to_f77.f90). It passes the same argument as a target and a pointer. In both cases, it is received by reference by the Compaq Fortran 77 subroutine as a target (regular) argument. The interface block in Example A–1 is not needed, but does allow data type checking.

Example A–1 Compaq Fortran Program Calling a Compaq Fortran 77 Subroutine

```
! Pass arrays to f77 routine. File: array_to_f77.f90

! this interface block is not required, but must agree
! with actual procedure. It can be used for type checking.

interface                    ! Procedure interface block
  subroutine meg(a)
  integer :: a(3)
  end subroutine
end interface

integer, target :: x(3)
integer, pointer :: xp(:)

x = (/ 1,2,3 /)
xp => x

call meg(x)                  ! Call f77 subroutine twice.
call meg(xp)
end
```

Example A–2 shows the Compaq Fortran 77 subprogram called by the Compaq Fortran program (file array_f77.f).

Example A–2 Compaq Fortran 77 Subroutine Called by a Compaq Fortran Program

```
! Get array argument from F90. File: array_f77.f

 subroutine meg(a)
 integer a(3)
 print *,a
 end
```

These files (shown in Example A–1 and Example A–2) might be compiled, linked, and run as follows:

```
% f77 -c array_f77.f
% f90 -o array_to_f77 array_to_f77.f90 array_f77.o
% array_to_f77
           1           2           3
           1           2           3
```

In Example A–1, because array a is not defined as a pointer in the interface block, the Compaq Fortran pointer variable xp is passed as target data by reference (address of the target data).

However, if the interface to the dummy argument had the POINTER attribute, the variable xp would be passed by descriptor. This descriptor would not work with the Compaq Fortran 77 program shown in Example A–2.

For More Information:

- On how Compaq Fortran handles arguments and function return values, see Section 11.1.4.
- On explicit interfaces, see the *Compaq Fortran Language Reference Manual*.
- On compatibility between the Compaq Fortran and Compaq Fortran 77 languages, see Appendix A.
- On other aspects of the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.

A.5.2 Using Data Items in Common Blocks

To make global data available across Compaq Fortran and Compaq Fortran 77 procedures, use common blocks.

Common blocks are supported by both Compaq Fortran 77 and Compaq Fortran, but modules are not supported by Compaq Fortran 77. Some suggestions about using common blocks follow:

- Use the *same* COMMON statement to ensure that the data items match in order, type, and size.

  If multiple Compaq Fortran procedures will use the same common block, declare the data in a module and reference that module with a USE statement where needed.

  If Compaq Fortran 77 procedures use the same common block as the Compaq Fortran procedures and the common block is declared in a module, consider modifying the Compaq Fortran 77 source code as follows:

  – Replace the common block declaration with the appropriate USE statement.

- Recompile the Compaq Fortran 77 source code with the `f90` command.

- Specify the same alignment characteristics with the `-align` option when compiling both Compaq Fortran procedures (`f90` command) and Compaq Fortran 77 procedures (`f77` command).

  When compiling the source files with more than one `f90` or `f77` command, consistently use the `-align dcommons` or `-align commons` option. This naturally aligns data items in a common block and ensures consistent format of the common block.

- Make sure the sizes of INTEGER, LOGICAL, REAL, and COMPLEX declarations match.

  For example, Compaq Fortran declarations of REAL (KIND=4) and INTEGER (KIND=4) match Compaq Fortran 77 declarations of REAL*4 and INTEGER*4. For COMPLEX values, a Compaq Fortran declaration of COMPLEX (KIND=4) matches a Compaq Fortran 77 declaration of COMPLEX*8; COMPLEX (KIND=8) matches COMPLEX*16. Fortran 77 does not have COMPLEX*32 data.

  Your source programs may contain INTEGER, LOGICAL, REAL, or COMPLEX declarations without a kind parameter or size specifier. In this case, either omit or specify the same options that control the sizes of these declarations when compiling the procedures with multiple commands (same rules as Section A.5.1).

A.5.3 I/O to the Same Unit Number

Compaq Fortran and Compaq Fortran 77 share the same run-time system, so you can perform I/O to the same unit number with Compaq Fortran and Compaq Fortran 77 procedures. For instance, a Compaq Fortran main program can open the file, a Compaq Fortran 77 function can issue READ or WRITE statements to the same unit, and the Compaq Fortran main program can close the file.

For More Information:

- On the Compaq Fortran language, see the *Compaq Fortran Language Reference Manual*.
- On passing arguments, function return values, and the contents of registers on Compaq Tru64 UNIX systems, see the *Compaq Tru64 UNIX Calling Standard for Alpha Systems*.
- On Compaq Fortran intrinsic data types, see Chapter 9.
- On Compaq Fortran I/O, see Chapter 7.

B

Compaq Fortran Environment Variables

This appendix contains the following topics:

- Section B.1, Commands for Setting and Unsetting Environment Variables
- Section B.2, Compile-Time Environment Variables
- Section B.3, Run-Time Environment Variables

In addition to the environment variables recognized by Compaq Fortran, the Compaq Tru64 UNIX operating system recognizes other environment variables. For example, you can use the PROFDIR environment variable to request a profile data file name different from mon.out during pixie command execution on a Compaq Tru64 UNIX system and you can set the LD_LIBRARY_PATH environment variable to install a private shared library.

B.1 Commands for Setting and Unsetting Environment Variables

The commands used to set and unset environment variables vary with the shell in use.

To view the previously set environment variables, use the printenv command. (See printenv(1).)

B.1.1 Bourne Shell (sh) and Bourne Again Shell (bash) and Korn Shell (ksh) Commands

With the Bourne shell (sh), Bourne Again Shell (bash) *(L*X only)*, and Korn shell (ksh), use an export and an assignment statement to set an environment variable:

```
$ export environment-variable-name
$ environment-variable-name=value
```

For example, to associate the environment variable TMPDIR with the directory /usr/users/smith/, enter:

```
$ export TMPDIR
$ TMPDIR=/usr/users/smith/
```

To remove the association of an environment variable and its value with the Bourne or Korn shell or bash shell *(L*X only)*, use the unset command:

```
$ unset environment-variable-name
```

B.1.2 C Shell (csh) Commands

With the C shell (csh), use the setenv command to set an environment variable value:

```
% setenv environment-variable-name value
```

For example, to associate the environment variable FORT8 with the file located at /usr/users/smith/test.dat, enter:

```
% setenv FORT8 /usr/users/smith/test.dat
```

To remove the association of an environment variable and its value with the C shell, use the unsetenv command:

```
% unsetenv environment-variable-name
```

B.2 Compile-Time Environment Variables

Table B–1 describes environment variables that Compaq Fortran recognizes at compile time.

For more information, see Section 7.5.7, Using Environment Variables.

Table B–1 Compile-Time Environment Variables

Environment Variable	Description
DECF90	Location of the f90 compiler to invoke.
DECF90_CC	Location of the cc command.
DECF90_DIR	Path for the f90 command to use to find the f90 compiler and for_main.o file. DECF90_DIR supersedes DECF90 and is superseded by DECF90_LIB_DIR.

(continued on next page)

Table B–1 (Cont.) Compile-Time Environment Variables

Environment Variable	Description
DECF90_FPP	Contains the full file name of the preprocessor to be run. If -cpp is specified on the command line, this preprocessor is run instead of cpp. See Section 3.31. If -fpp is specified on the command line, this preprocessor is run instead of fpp. See Section 3.45.
DECF90_GMPILIB *(TU*X only)*	Variable for the f90 (or f95) command. When the -hpf_target gmpi option is specified, this variable is used to specify a path to the desired generic Message Passing Interface (MPI) library to link with. If this variable is not used, then you must specify the desired MPI library to link against on the command line. See Section 3.50, -hpf, -hpf num, and Related Options — Compile HPF Programs for Parallel Execution.
DECF90_HPF_TARGET *(TU*X only)*	Variable for the f90 (or f95) command. If this variable is set, it must have one of the values of the -hpf_target option. See Section 3.50, -hpf, -hpf num, and Related Options — Compile HPF Programs for Parallel Execution. DECF90_WSF_TARGET is a nonpreferred synonym for DECF90_HPF_TARGET.
DECF90_INIT	Initial options for the f90 (or f95) command. If this variable is defined, its value must have the form: [[*pre*] [:: [*post*]] The items enclosed in square brackets ([]) are optional and can be empty. The *pre* and *post* variables are strings to be added to the command line: • *pre* is added to the front of the command line, before any characters the user enters. • *post* is added to the end of the command line.
DECF90_LIB_DIR	Path for the f90 compiler to use to find the Fortran run-time libraries.
DECFORT *(TU*X only)*	Location of the Fortran 77 compiler to invoke.
DECFORT_CC *(TU*X only)*	Location of the cc command for the Fortran 77 compiler.

(continued on next page)

Table B–1 (Cont.) Compile-Time Environment Variables

Environment Variable	Description
DECFORT_DIR *(TU*X only)*	Path for the f77 command to use to find the f77 compiler and for_main.o file. DECFORT_DIR supersedes DECFORT and is superseded by DECFORT_LIB_DIR.
DECFORT_FPP *(TU*X only)*	Requests that the Fortran preprocessor fpp be run before compiling Fortran 77 source files when -cpp has been specified. For more information about the Fortran preprocessor and the related -fpp and -nofpp options, see Section 3.45.
DECFORT_INIT *(TU*X only)*	Initial options for the f77 command. If this variable is defined, its value must have the form: [[*pre*] [:: [*post*]] The items enclosed in square brackets ([]) are optional and can be empty. The *pre* and *post* variables are strings to be added to the command line: • *pre* is added to the front of the command line, before any characters the user enters. • *post* is added to the end of the command line.
DECFORT_LIB_DIR *(TU*X only)*	Path for the f77 compiler to use to find the Fortran run-time libraries.
LD_LIBRARY_PATH *(TU*X only)*	Path on which to find shared libraries for an executable program.
TMPDIR	Specifies an alternate working directory where temporary files are created during preprocessing or compilation. To specify an alternate working directory for temporary files, set the TMPDIR environment variable to the desired directory name. If TMPDIR is not set, temporary files created during preprocessing or compilation reside in the /tmp directory. For large applications, you might set this variable to balance disk I/O during compilation. For performance reasons, use a local disk (rather than using an NFS-mounted disk) to contain the temporary files.

B.3 Run-Time Environment Variables

Table B–2 describes the environment variables Compaq Fortran recognizes at run time.

Environment variables used with OpenMP Fortran API (multithreaded parallel processing) are described in Section 6.4, Environment Variables for Adjusting the Run-Time Environment.)

For more information, see Section 7.5.7, Using Environment Variables.

Table B–2 Run-Time Environment Variables

Environment Variable	Description
`decfort_dump_flag`	Requests that a core dump (`core` file) be created when any severe Compaq Fortran run-time error occurs. Most severe Compaq Fortran run-time errors do not result in a core dump, unless accompanied by certain operating system messages. To request that a `core` file be created for all severe Compaq Fortran run-time errors, set the environment variable `decfort_dump_flag` to the character `Y` or `y` and then run the erroneous program for which you need a `core` file created. For more information, see Section 8.1.4.
`FOR_ACCEPT`	For programs compiled with the `f90` command `-vms` option, specifies the name of a file to receive input from an ACCEPT statement instead of `stdin`.
`FOR_DISABLE_STACK_TRACE` *(TU*X only)*	Disables outputting of stack trace information as part of the run-time error message for a severe error. The normal run-time error message is produced instead. See Section 8.1.1, Run-Time Message Format.
`FOR_PRINT`	For programs compiled with the `f90` command `-vms` option, specifies the name of a file to receive output from a PRINT statement instead of `stdout`.
`FOR_READ`	For programs compiled with the `f90` command `-vms` option, specifies the name of a file to receive input from a READ statement instead of `stdin`.
`FOR_TYPE`	For programs compiled with the `f90` command `-vms` option, specifies the name of a file to receive output from a TYPE statement instead of `stdout`.

(continued on next page)

Table B–2 (Cont.) Run-Time Environment Variables

Environment Variable	Description
FORT*n*	Allows the user to specify the directory and file name at run time for a logical unit (*n*) for which the OPEN statement does not specify a file name. If the appropriate environment variable is not set and the OPEN statement does not specify a file name for that logical unit, a default file name of `fort.`*n* is used.
FORT_BUFFERED	Specifies that buffered I/O will be used for sequential output to all I/O units, except those whose output is to the terminal. This provides a run-time mechanism to support the behavior enabled by the `-assume buffered_io` option. For more information, see Section 3.6 and Section 5.6.7, Efficient Use of Record Buffers and Disk I/O.
FORT_CONVERT*n*	For an unformatted file, specifies the nonnative numeric format of the data at run time for a logical unit (*n*). Otherwise, the nonnative numeric format of the unformatted data must be specified at compile-time by using the `f90` command `-convert` *type* option. For more information, see Section 10.5.1.
FORT_CONVERT.*ext*	For an unformatted file, specifies the nonnative numeric format of the data at run time for a file whose suffix is *ext*. Otherwise, the nonnative numeric format of the unformatted data must be specified at compile-time by using the `f90` command `-convert` *type* option. For more information, see Section 10.5.2.
MP_* *(TU*X only)*	Compaq Fortran parallel compiler environment variables, used with directed parallel processing. For more information, see Section 6.4, Environment Variables for Adjusting the Run-Time Environment.
NLSPATH	If the run-time message catalog file cannot be located, the Compaq Fortran run-time system attempts to open the message catalog file at the location indicated by the `NLSPATH` environment variable. For more information, see Section 8.1.2.
OMP_* *(TU*X only)*	OpenMP Fortran API environment variables, used with directed parallel processing. For more information, see Section 6.4, Environment Variables for Adjusting the Run-Time Environment.

(continued on next page)

Table B–2 (Cont.) Run-Time Environment Variables

Environment Variable	Description
TMPDIR	Specifies an alternate working directory where scratch files are created. To specify an alternate working directory for scratch files, set the TMPDIR environment variable to the desired directory name. For performance reasons, use a local disk (rather than an NFS-mounted disk) to contain the scratch files. If TMPDIR is not set, scratch files are created in the directory specified in the OPEN statement DEFAULTFILE (if specified).

C
Compiler Output Listings

This appendix describes the three sections of an output listing produced by the Compaq Fortran compiler:

- Section C.1, Source-Code Section of the Output Listing
- Section C.2, Machine-Code Section of the Output Listing
- Section C.3, Compilation Summary of the Output Listing

To request a listing file, use the `-V` option. For example:

```
% f90 -V peak.f90
```

C.1 Source-Code Section of the Output Listing

The source-code section of a compiler output listing displays the source program as it appears in the input file, with the addition of sequential line numbers generated by the compiler. Example C–1 shows a sample of a source-code section of a compiler output listing.

Example C–1 Sample Source Code Listing

```
RELAX2          Source Listing    14-Nov-2001 09:59:31  Compaq Fortran V5.5-1843
                                  13-Feb-1998 11:00:18  listing.f90

       1    SUBROUTINE RELAX2(EPS)
       2      INTEGER, PARAMETER :: M=40
       3      INTEGER, PARAMETER :: N=60
       4      COMMON X (M,N)
       5      LOGICAL DONE
       6  1    DONE = .TRUE.
       7       DO J=1,N-1
       8         DO I=1,M-1
       9           XNEW = (X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))/4
      10           IF (ABS(XNEW-X(I,J)) > EPS) DONE = .FALSE.
      11           X(I,J) = XNEW
      12         END DO
      13       END DO
      14       IF (.NOT. DONE) GO TO 1
      15       RETURN
      16     END SUBROUTINE
```

The first heading line contains "Source Listing," the date and time the listing file was created, and the version of Compaq Fortran.

The second line contains the creation date of the source file and its file name.

Compiler-generated line numbers appear in the left margin.

Compile-time error messages that contain line numbers refer to these compiler-generated line numbers. See Section 2.3, Compiler Limits, Diagnostic Messages, and Error Conditions.

C.2 Machine-Code Section of the Output Listing

The machine-code section of a compiler output listing provides a symbolic representation of the compiler-generated object code. The representation of the generated code and data is similar to that of assembler language.

The machine-code section is optional. To create a listing file with a machine-code section, specify both the `-V` and `-show code` options. For example:

```
% f90 -c -V -show code peak.f90
```

Example C–2 shows a sample of a machine-code section of a compiler output listing for an Alpha system.

Note

The machine code listing is for reference purposes only. Such code is not intended to be assembled and run.

Example C–2 Sample Machine-Code Listing

```
RELAX2    Machine Code Listing   14-Nov-2001 09:59:31  Compaq Fortran V5.5-1843
                                 13-Feb-1998 11:00:18  listing.f90

                                .text
                                .globl  relax2_
                                .ent    relax2_
                                .eflag 16
                    0000    relax2_:
27BB0001            0000            ldah    gp, relax2_         # gp, (r27)
2FFE0000            0004            unop
23BD8420            0008            lda     gp, relax2_         # gp, (gp)
2FFE0000            000C            unop
                    0010    L$1:
23DEFFD0            0010            lda     sp, -48(sp)
88100000            0014            lds     f0, (r16)                       # 000010
B75E0000            0018            stq     r26, (sp)
A79D8010            001C            ldq     r28, (gp)                       # 000009
A47D8018            0020            ldq     r3, var$0004        # r3, 8(gp)
9C5E0008            0024            stt     f2, 8(sp)                       # 000001
9C7E0010            0028            stt     f3, 16(sp)
9C9E0018            002C            stt     f4, 24(sp)
20630000            0030            lda     r3, var$0004        # r3, (r3)  # 000009
9CBE0020            0034            stt     f5, 32(sp)                      # 000001
9CDE0028            0038            stt     f6, 40(sp)
                                .mask 0x04000000,-48
                                .fmask 0x0000007C,-40
                                .frame  $sp, 48, $26
                                .prologue 1
883C0000            003C            lds     f1, (r28)                       # 000009
                    0040    .1:                                             # 000006
209FFFFF            0040            mov     -1, DONE            # -1, r4
47E77405            0044            mov     59, var$0002        # 59, r5    # 000007
47F41406            0048            mov     160, r6                         # 000011
2FFE0000            004C            unop
                    0050    lab$0004:                                       # 000007
40660407            0050            addq    r3, r6, r7                      # 000009
47E4F408            0054            mov     39, var$0003        # 39, r8    # 000008
20E7FF5C            0058            lda     r7, -164(r7)                    # 000009
2FFE0000            005C            unop
                    0060    lab$0008:                                       # 000008
89470008            0060            lds     f10, 8(r7)                      # 000009
8987FF64            0064            lds     f12, -156(r7)
47FF041B            0068            clr     r27                             # 000010
89670000            006C            lds     f11, (r7)                       # 000009
 .
 .
 .
Routine Size: 1012 bytes,    Routine Base: $CODE$ + 0000
```

(continued on next page)

Example C–2 (Cont.) Sample Machine-Code Listing

```
                        .rdata
                $$1:
00000000     0000       .quad   .literal
00000000     0008       .quad   _BLNK__
                        .rconst
                $$2:
3E800000     0000       .long 0x3E800000 # .float 0.2500000
                        .data
                $$3:
             0000       ; Procedure descriptor for relax2_
                        ; flags :short
                        ; rsa_offset          : 0
                        ; imask               : 0x00
                        ; fmask               : 0x1F
                        ; frame_size          : 6
                        ; sp_set              : 4
                        ; entry_length        : 15
                        .data
                $$4:
             0000       ; Code range descriptor for relax2_
                        beginaddress     :     0000
                        rpd_offset     :     0000
                        flags               : Standard
                        .comm   _BLNK__ 9600
```

C.2.1 How Generated Code and Data are Represented in Machine-Code Listings

Like a source listing, the first heading line contains the name of the program, subroutine, or function; the date and time the listing file was created; and the version of Compaq Fortran.

The second heading line contains "Machine Code Listing," the creation date of the source file, and the name of the source file.

The third heading line contains a `.section` assembler directive, indicating the attributes of the machine-code program unit. The `$CODE$` shown in Example C–2 indicates a code section.

The lines following each data program section provide information such as the contents of storage initialized for FORMAT statements, DATA statements, constants, and subprogram argument call lists.

The lines following `$CODE$` show the machine instructions represented in the form of Alpha assembler mnemonics and syntax. Each line contains compiler-generated object code starting at the left margin, followed by the hexadecimal byte offset (four hexadecimal digits), followed by the actual assembler code.

C.2.2 Assembler Code Represented in Machine-Code Listings

General registers (0 through 31) are represented by r0 through r31 and floating-point registers are similarly represented by `f`*n*.

Variables and arrays defined in the source program are shown as they were defined in the program. Offsets from variables and arrays are shown in decimal. Optimization frequently places variables in registers, so variable names may be missing.

Fortran source labels referenced in the source program are shown with a period (`.`) prefix. For example, if the source program refers to label 300, the label appears in the machine-code listing as .300. Labels that appear in the source program, but are not referenced or are deleted during compiler optimization, are ignored. They do not appear in the machine-code listing unless you specified `-O0`.

The compiler may generate labels for its own use. These labels appear as `L$`*n* or `lab$000`*n*, where the value of *n* is unique for each such label in a program unit.

Integer constants are shown as signed integer values.

Addresses are represented by the program section name plus the hexadecimal offset within that program section. Changes from one program section to another are indicated by lines.

C.3 Compilation Summary of the Output Listing

The final entries on the compiler listing are the compiler options and compiler statistics.

The options shown include the ones specified on the command line, both `f90` and `fort`, and the ones in effect as defaults during the compilation. The compiler statistics are the machine resources used by the compiler.

Example C–3 shows how compiler options and `f90` (on Tru64 UNIX systems) command-line options and compilation statistics appear on the listing.

A summary of compilation statistics appears at the end of the listing file.

Example C–3 Sample Compilation Summary on Tru64 UNIX Systems

(continued on next page)

Example C–3 (Cont.) Sample Compilation Summary on Tru64 UNIX Systems

```
COMPILER OPTIONS BEING USED

  no   -align commons                    no   -align dcommons
       -align records                    no   -align sequence
  no   -align rec1byte                   no   -align rec2byte
  no   -align rec4byte                   no   -align rec8byte
       -altparam                         no   -annotations code
  no   -annotations detail               no   -annotations feedback
  no   -annotations inlining             no   -annotations loop_transforms
  no   -annotations loop_unrolling       no   -annotations prefetching
  no   -annotations shrinkwrapping       no   -annotations software_pipelining
  no   -annotations tail_calls           no   -annotations tail_recursion
       -arch generic                          -assume accuracy_sensitive
  no   -assume bigarrays                 no   -assume buffered_io
  no   -assume byterecl                  no   -assume cc_omp
  no   -assume dummy_aliases             no   -assume gfullpath
  no   -assume minus0                         -assume protect_constants
  no   -assume pthreads_lock                  -assume source_include
       -assume underscore                no   -assume 2underscores
       -assume zsize                     no   -automatic
  no   -bintext                               -call_shared
       -ccdefault default                no   -check arg_temp_created
  no   -check bounds                     no   -check format
  no   -check omp_bindings               no   -check output_conversion
  no   -check overflow                        -check power
  no   -check underflow                       -convert native
  no   -D                                     -double_size 64
  no   -d_lines                               -error_limit 30
  no   -extend_source                    no   -f66
  no   -f77rtl                           no   -fast
  no   -fpscomp filesfromcmd             no   -fpscomp general
  no   -fpscomp ioformat                 no   -fpscomp ldio_spacing
  no   -fpscomp logicals                 no   -fpconstant
       -fpe0                                 -fprm nearest
       -free                                 -g1
       -granularity quadword                 -Gt0
  no   -hpf_matmul                       no   -hpf
       -iface                            no   -intconstant
       -integer_size 32                  no   -ladebug
       -machine_code                         -math_library accurate
  no   -mixed_str_len_arg                no   -module
       -names lowercase                      -nearest_neighbor
  no   -nohpf_main                       no   -non_shared
  no   -noinclude                             -numnodes 0
  no   -numa                                  -numa_memories 0
       -numa_tpm 0                           -O4
       -inline speed                     no   -transform_loops
       -pipeline                             -speculate none
       -tune generic                         -unroll 0
  no   -pad_source                       no   -pg
       -real_size 32                     no   -recursive
       -reentrancy none                      -shadow_width 0
```

(continued on next page)

Example C–3 (Cont.) Sample Compilation Summary on Tru64 UNIX Systems

```
  no   -shared                          no   -show hpf_all
  no   -show hpf_comm                   no   -show hpf_default
  no   -show hpf_dev                    no   -show hpf_indep
  no   -show hpf_nearest                no   -show hpf_punt
  no   -show hpf_temps                  no   -show include
       -show map                        no   -show wsfinfo
  no   -std                             no   -synchronous_exceptions
  no   -syntax_only                     no   -vms
       -warn alignments                 no   -warn argument_checking
  no   -warn declarations                    -warn general
       -warn granularity                no   -warn hpf
  no   -warn ignore_loc                 no   -warn truncated_source
       -warn uncalled                        -warn uninitialized
  no   -warn unused                          -warn usage
       -warning_severity warning        no   -fuse_xref

       -I    path : /usr/lib/cmplrs/hpfrtl/,/usr/include/
       -V    filename : listing.l
       -o    filename : listing.o

COMPILER: Compaq Fortran V5.5-1843-48BB1
```

Example C–4 shows how compiler options and `fort` (on Linux systems) command-line options and compilation statistics appear on the listing.

A summary of compilation statistics appears at the end of the listing file. Note that some options that are available on Tru64 UNIX systems are not available on Linux Alpha systems.

Example C–4 Sample Compilation Summary on Linux Systems

```
COMPILER OPTIONS BEING USED

  no   -align commons                    no   -align dcommons
       -align records                    no   -align sequence
  no   -align rec1byte                   no   -align rec2byte
  no   -align rec4byte                   no   -align rec8byte
       -altparam                         no   -annotations code
  no   -annotations detail               no   -annotations feedback
  no   -annotations inlining             no   -annotations loop_transforms
  no   -annotations loop_unrolling       no   -annotations prefetching
  no   -annotations shrinkwrapping       no   -annotations software_pipelining
  no   -annotations tail_calls           no   -annotations tail_recursion
       -arch generic                          -assume accuracy_sensitive
  no   -assume bigarrays                 no   -assume buffered_io
  no   -assume byterecl                  no   -assume cc_omp
  no   -assume dummy_aliases             no   -assume gfullpath
  no   -assume minus0                         -assume protect_constants
  no   -assume pthreads_lock                  -assume source_include
       -assume underscore                     -assume 2underscores
       -assume zsize                     no   -automatic
  no   -bintext                               -call_shared
       -ccdefault default                no   -check arg_temp_created
  no   -check bounds                     no   -check format
  no   -check omp_bindings               no   -check output_conversion
  no   -check overflow                        -check power
  no   -check underflow                       -convert native
  no   -D                                     -double_size 64
  no   -d_lines                               -error_limit 30
  no   -extend_source                    no   -f66
  no   -f77rtl                           no   -fast
  no   -fpscomp filesfromcmd             no   -fpscomp general
  no   -fpscomp ioformat                 no   -fpscomp ldio_spacing
  no   -fpscomp logicals                      -fixed
  no   -fpconstant                            -fpe0
       -fprm nearest                          -g1
       -granularity quadword                  -Gt0
  no   -hpf_matmul                       no   -hpf
       -iface                            no   -intconstant
       -integer_size 32                  no   -ladebug
  no   -machine_code                          -math_library accurate
  no   -mixed_str_len_arg                no   -module
       -names lowercase                       -nearest_neighbor
  no   -nohpf_main                       no   -non_shared
  no   -noinclude                             -numnodes 0
  no   -numa                                  -numa_memories 0
       -numa_tpm 0                            -O4
       -inline speed                     no   -transform_loops
       -pipeline                              -speculate none
       -tune generic                          -unroll 0
  no   -pad_source                       no   -pg
       -real_size 32                     no   -recursive
       -reentrancy none                       -shadow_width 0
```

(continued on next page)

Example C–4 (Cont.) Sample Compilation Summary on Linux Systems

```
  no   -shared                          no   -show hpf_all
  no   -show hpf_comm                   no   -show hpf_default
  no   -show hpf_dev                    no   -show hpf_indep
  no   -show hpf_nearest                no   -show hpf_punt
  no   -show hpf_temps                  no   -show include
       -show map                        no   -show wsfinfo
  no   -std                             no   -synchronous_exceptions
  no   -syntax_only                     no   -vms
       -warn alignments                 no   -warn argument_checking
  no   -warn declarations                    -warn general
       -warn granularity                no   -warn hpf
  no   -warn ignore_loc                 no   -warn truncated_source
       -warn uncalled                        -warn uninitialized
  no   -warn unused                          -warn usage
       -warning_severity warning        no   -fuse_xref

       -I    path : /usr/include/
       -V    filename : end.l
       -o    filename : /tmp/forlYrc7r.o

COMPILER: Compaq Fortran V1.2.0-1843-48BB1
```

D

Parallel Library Routines

Note

This appendix applies only to Compaq Fortran on Tru64 UNIX systems.

This appendix contains the following sections:

- Section D.1, OpenMP Fortran API Run-Time Library Routines
- Section D.2, Other Parallel Threads Routines

This appendix summarizes the library routines available for use with directed parallel decomposition requested by the `-mp` and `-omp` compiler options.

Where applicable, new applications should call run-time parallel library routines using the OpenMP Fortran API format. (See Section D.1, OpenMP Fortran API Run-Time Library Routines.) For compatibility with existing programs, the Compaq Fortran compiler recognizes equivalent routines of the formats described in Section D.2. Thus, for example, if your program calls `_OtsGetNumThreads`, the Compaq Fortran compiler interprets that as a call to `omp_get_num_threads`.

D.1 OpenMP Fortran API Run-Time Library Routines

This section describes:

- Library routines that control and query the parallel execution environment
- General-purpose lock routines supported by Compaq Fortran.

Table D–1 lists the supported OpenMP Fortran API run-time library routines. These routines are all external procedures.

Table D–1 OpenMP Fortran API Run-Time Library Routines

Routine Name	Usage
Library Routines That Control and Query the Parallel Execution Environment	
omp_get_dynamic	Inform if dynamic thread adjustment is enabled. See Section D.1.1.1, omp_get_dynamic.
omp_get_max_threads	Get the maximum value that can be returned by calls to the omp_get_num_threads( ) function. See Section D.1.1.2, omp_get_max_threads.
omp_get_nested	Inform if nested parallelism is enabled. See Section D.1.1.3, omp_get_nested.
omp_get_num_procs	Get the number of processors that are available to the program. See Section D.1.1.4, omp_get_num_procs.
omp_get_num_threads	Get the number of threads currently in the team executing the parallel region from which the routine is called. See Section D.1.1.5, omp_get_num_threads.
omp_get_thread_num	Get the thread number, within the team, in the range from zero to omp_get_num_threads( )–1. See Section D.1.1.6, omp_get_thread_num.
omp_in_parallel	Inform whether or not a region is executing in parallel. See Section D.1.1.7, omp_in_parallel.
omp_set_dynamic	Enable or disable dynamic adjustment of the number of threads available for execution of parallel regions. See Section D.1.1.8, omp_set_dynamic.
omp_set_nested	Enable or disable nested parallelism. See Section D.1.1.9, omp_set_nested.
omp_set_num_threads	Set the number of threads to use for the next parallel region. See Section D.1.1.10, omp_set_num_threads.
General-Purpose Lock Routines	
omp_destroy_lock	Disassociate a lock variable from any locks. See Section D.1.2.1.
omp_init_lock	Initialize a lock to be used in subsequent calls. See Section D.1.2.2.
omp_set_lock	Make the executing thread wait until the specified lock is available. See Section D.1.2.3.
omp_test_lock	Try to set the lock associated with a lock variable. See Section D.1.2.4.
omp_unset_lock	Release the executing thread from ownership of a lock. See Section D.1.2.5.

D.1.1 Library Routines That Control and Query the Parallel Execution Environment

These routines are described in detail in the following sections.

D.1.1.1 omp_get_dynamic

Determines the status of dynamic thread adjustment.

Syntax:

```
INTERFACE
    LOGICAL FUNCTION omp_get_dynamic ()
    END FUNCTION omp_get_dynamic
END INTERFACE
LOGICAL result
result = omp_get_dynamic ()
```

Return Values:

This function returns `TRUE` if dynamic thread adjustment is enabled; otherwise it returns `FALSE`. The function always returns `FALSE` if dynamic adjustment of the number of threads is not implemented.

See Also:

Section D.1.1.8, omp_set_dynamic

D.1.1.2 omp_get_max_threads

Returns the maximum value that can be returned by calls to the `omp_get_num_threads( )` function.

Syntax:

```
INTERFACE
    INTEGER FUNCTION omp_get_max_threads ()
    END FUNCTION omp_get_max_threads
END INTERFACE
INTEGER result
result = omp_get_max_threads ()
```

Description:

If your program uses `omp_set_num_threads( )` to change the number of threads, subsequent calls to `omp_get_max_threads( )` will return the new value. When the `omp_set_dynamic( )` routine is set to `TRUE`, you can use `omp_get_max_threads( )` to allocate data structures that are maximally sized for each thread.

This function has global scope.

Return Values:

This function returns the maximum value whether executing from a serial region or from a parallel region.

If your program used omp_set_num_threads to change the number of threads, subsequent calls to omp_get_max_threads will return the new value.

See Also:

Section D.1.1.10, omp_set_num_threads
Section D.1.1.8, omp_set_dynamic

D.1.1.3 omp_get_nested

Determines the status of nested parallelism.

Syntax:

```
INTERFACE
    LOGICAL FUNCTION omp_get_nested ()
    END FUNCTION omp_get_nested
END INTERFACE
LOGICAL result
result = omp_get_nested ()
```

Description:

This function returns `TRUE` if nested parallelism is enabled. If nested parallelism is disabled it returns `FALSE`. The function always returns `FALSE` if nested parallelism is not implemented.

See Also:

Section D.1.1.9, omp_set_nested

D.1.1.4 omp_get_num_procs

Returns the number of processors that are available to the program.

Syntax:

```
INTERFACE
    INTEGER FUNCTION omp_get_num_procs ()
    END FUNCTION omp_get_num_procs
END INTERFACE
INTEGER result
result = omp_get_num_procs ()
```

Return Values:

This function returns an integer value indicating the number of processors your program has available.

D.1.1.5 omp_get_num_threads

Returns the number of threads currently in the team executing the parallel region from which it is called.

Syntax:

```
INTERFACE
    INTEGER FUNCTION omp_get_num_threads ()
    END FUNCTION omp_get_num_threads
END INTERFACE
INTEGER result
result = omp_get_num_threads ()
```

Description:

This function interacts with the omp_set_num_threads call and the OMP_NUM_THREADS environment variable that control the number of threads in a team. If the number of threads has not been explicitly set by the user, the default is implementation dependent.

The omp_get_num_threads function binds to the closest enclosing PARALLEL directive (see Chapter 6, Parallel Compiler Directives and Their Programming Environment). It returns 1 if the call is made from the serial portion of a program, or from a nested parallel region that is serialized.

See Also:

Section D.1.1.10, omp_set_num_threads
OMP_NUM_THREADS environment variable in Table 6–4, OpenMP Fortran API Environment Variables

D.1.1.6 omp_get_thread_num

Returns the thread number, within the team.

Syntax:

```
INTERFACE
    INTEGER FUNCTION omp_get_thread_num ()
    END FUNCTION omp_get_thread_num
END INTERFACE
INTEGER result
result = omp_get_thread_num ()
```

Description:

This function binds to the closest enclosing PARALLEL directive (see Chapter 6, Parallel Compiler Directives and Their Programming Environment). The master thread of the team is thread zero.

Return Values:

The value returned ranges from zero to `omp_get_num_threads( )` - 1. The function returns zero when called from a serial region or from within a nested parallel region that is serialized.

See Also:

Section D.1.1.5, omp_get_num_threads
Section D.1.1.10, omp_set_num_threads

D.1.1.7 omp_in_parallel

Returns whether or not a region is executing in parallel.

Syntax:

```
INTERFACE
    LOGICAL FUNCTION omp_in_parallel ()
    END FUNCTION omp_in_parallel
END INTERFACE
LOGICAL result
result = omp_in_parallel()
```

Description:

This function has global scope.

Return Values:

This function returns `TRUE` if it is called from the dynamic extent of a region executing in parallel, even if nested regions exist that may be serialized; otherwise it returns `FALSE`. A parallel region that is serialized is not considered to be a region executing in parallel.

D.1.1.8 omp_set_dynamic

Enables or disables dynamic adjustment of the number of threads available for execution in a parallel region.

Syntax:

```
INTERFACE
    SUBROUTINE omp_set_dynamic (enable)
    LOGICAL enable
    END SUBROUTINE omp_set_dynamic
END INTERFACE
LOGICAL scalar_local_expression
CALL omp_set_dynamic (scalar_logical_expression)
```

Description:

To obtain the best use of system resources, certain run-time environments automatically adjust the number of threads that are used for executing subsequent parallel regions. This adjustment is enabled only if the value of the scalar logical expression to `omp_set_dynamic` is `TRUE`. Dynamic adjustment is disabled if the value of the scalar logical expression is `FALSE`.

When dynamic adjustment is enabled, the number of threads specified by the user becomes the maximum thread count. The number of threads remains fixed throughout each parallel region and is reported by the `omp_get_num_threads( )` function.

A call to `omp_set_dynamic` overrides the `OMP_DYNAMIC` environment variable.

The default for dynamic thread adjustment is implementation dependent. A user code that depends on a specific number of threads for correct execution should explicitly disable dynamic threads. Implementations are not required to provide the ability to dynamically adjust the number of threads, but they are required to provide the interface in order to support portability across platforms.

See Also:

Section D.1.1.1, omp_get_dynamic
Section D.1.1.5, omp_get_num_threads
`OMP_DYNAMIC` environment variable in Table 6–4, OpenMP Fortran API Environment Variables

D.1.1.9 omp_set_nested

Enables or disables nested parallelism.

Syntax:

```
INTERFACE
    SUBROUTINE omp_set_nested (enable)
    LOGICAL enable
    END SUBROUTINE omp_set_nested
END INTERFACE
LOGICAL scalar_logical_expression
CALL omp_set_nested (scalar_logical_expression)
END INTERFACE
```

Description:

If the value of the scalar logical expression is FALSE, nested parallelism is disabled, and nested parallel regions are serialized and executed by the current thread. This is the default. If the value of the scalar logical expression is set to TRUE, nested parallelism is enabled, and parallel regions that are nested can deploy additional threads to form the team.

A call to omp_set_nested overrides the OMP_NESTED environment variable.

When nested parallelism is enabled, the number of threads used to execute the nested parallel regions is implementation dependent. This allows implementations that comply with the OpenMP standard to serialize nested parallel regions, even when nested parallelism is enabled.

See Also:

Section D.1.1.3, omp_get_nested
OMP_NESTED environment variable in Table 6–4, OpenMP Fortran API Environment Variables

D.1.1.10 omp_set_num_threads

Sets the number of threads to use for the next parallel region.

Syntax:

```
INTERFACE
    SUBROUTINE omp_set_num_threads (number_of_threads)
    INTEGER number_of_threads
    END SUBROUTINE omp_set_num_threads
END INTERFACE
INTEGER scalar_integer_expression
CALL omp_set_num_threads (scalar_integer_expression)
```

Description:

The compiler evaluates the scalar integer expression and interprets its value as the number of threads to use. This function takes effect only when called from serial portions of the program. The behavior of the function is undefined if the function is called from a portion of the program where the omp_in_parallel function returns TRUE.

A call to omp_set_num_threads sets the maximum number of threads to use for the next parallel region when dynamic adjustment of the number of threads is enabled. A call to omp_set_num_threads overrides the OMP_NUM_THREADS environment variable.

See Also:

Section D.1.1.5, omp_get_num_threads
Section D.1.1.7, omp_in_parallel
OMP_NUM_THREADS environment variable in Table 6–4, OpenMP Fortran API Environment Variables

D.1.2 General-Purpose Lock Routines

The OpenMP run-time library includes a set of general-purpose locking routines. Your program must not attempt to access any lock variable, *var*, except through the routines described in this section. The *var* lock variable is an integer of a KIND large enough to hold an address. On Compaq Tru64 UNIX systems, *var* should be declared as INTEGER(KIND=8).

The lock control routines must be called in a specific sequence:

1. The lock to be associated with the lock variable must first be initialized.
2. The associated lock is made available to the executing thread.
3. The executing thread is released from lock ownership.
4. When finished, the lock must always be disassociated from the lock variable.

A simple SET_LOCK and UNSET_LOCK combination satisfies this requirement. If you want your program to do useful work while waiting for the lock to become available, you can use the combination of TRY_LOCK and UNSET_LOCK instead. For example:

```
      PROGRAM LOCK_USAGE
      implicit none
      integer(kind=4) ID
      include 'forompdef'       ! It's in /usr/include after installation
      INTEGER(KIND=8) LCK       ! This variable should be of size POINTER
      CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
      ID = OMP_GET_THREAD_NUM()
      CALL OMP_SET_LOCK(LCK)
      PRINT *, MY THREAD ID IS , ID
      CALL OMP_UNSET_LOCK(LCK)
      DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
      CALL SKIP(ID) ! Do not yet have lock, do something else
      END DO
      CALL WORK(ID) ! Have the lock, now do work
      CALL OMP_UNSET_LOCK(LCK)
!$OMP END PARALLEL
      CALL OMP_DESTROY_LOCK(LCK)
      END
```

The lock control routines are described in detail in the following sections.

D.1.2.1 omp_destroy_lock

Disassociates a given lock variable from any locks.

Syntax:

```
INTERFACE
    SUBROUTINE omp_destroy_lock (var)
    INTEGER(KIND=8) var
    END SUBROUTINE omp_destroy_lock
END INTERFACE
INTEGER(KIND=8) v
CALL omp_destroy_lock (v)
```

Restriction:

Attempting to call this routine with a lock variable that has not been initialized is an invalid operation and will cause a run-time error.

D.1.2.2 omp_init_lock

Initializes a lock associated with a given lock variable for use in subsequent calls.

Syntax:

```
INTERFACE
    SUBROUTINE omp_init_lock (var)
    INTEGER(KIND=8) var
    END SUBROUTINE omp_init_lock
END INTERFACE
INTEGER(KIND=8) v
CALL omp_init_lock (v)
```

Description:

The initial state of the lock variable *v* is unlocked.

Restriction:

Attempting to call this routine with a lock variable that is already associated with a lock is an invalid operation and will cause a run-time error.

D.1.2.3 omp_set_lock

Makes the executing thread wait until the specified lock is available.

Syntax:

```
INTERFACE
    SUBROUTINE omp_set_lock (var)
    INTEGER(KIND=8) var
    END SUBROUTINE omp_set_lock
END INTERFACE
INTEGER(KIND=8) v
CALL omp_set_lock (v)
```

Description:

When the lock becomes available, the thread is granted ownership.

Restriction:

Attempting to call this routine with a lock variable that has not been initialized is an invalid operation and will cause a run-time error.

D.1.2.4 omp_test_lock

Tries to set the lock associated with the lock variable *var*.

Syntax:

```
INTERFACE
    LOGICAL FUNCTION omp_test_lock (var)
    INTEGER(KIND=8) var
    END SUBROUTINE omp_test_lock
END INTERFACE
INTEGER(KIND=8) v
LOGICAL result
result = omp_test_lock (v)
```

Return Values:

If the attempt to set the lock specified by the variable succeeds, the function returns `TRUE`; otherwise it returns `FALSE`. In either case, the routine does not wait for the lock to become available.

Restriction:

Attempting to call this routine with a lock variable that has not been initialized is an invalid operation and will cause a run-time error.

D.1.2.5 omp_unset_lock

Releases the executing thread from ownership of the lock.

Syntax:

```
INTERFACE
    SUBROUTINE omp_unset_lock (var)
    INTEGER(KIND=8) var
    END SUBROUTINE omp_unset_lock
END INTERFACE
INTEGER(KIND=8) v
CALL omp_unset_lock (v)
```

Description:

If the thread does not own the lock specified by the variable, the behavior is undefined.

Restriction:

Attempting to call this routine with a lock variable that has not been initialized is an invalid operation and will cause a run-time error.

D.2 Other Parallel Threads Routines

Note

Compaq Fortran supports the set of parallel thread routines described in this section for existing programs. For creating new programs, use the set of routines described in Section D.1, OpenMP Fortran API Run-Time Library Routines.

Table D–2, Other Parallel Threads Routines shows additional parallel threads routines. The `_Otsxxx` (Compaq spelling) and the `mpc_xxx` (compatibility spelling) routine names are equivalent. For example, calling `_OtsGetNumThreads` is the same as calling `mpc_numthreads`.

Table D–2 Other Parallel Threads Routines

Routine Name	Description
_OtsGetMaxThreads mpc_maxnumthreads	Return the number of threads that would normally be used for parallel processing in the current environment. This is affected by the environment variable MP_THREAD_COUNT, by the number of processes in the current process's processor set, and by any call to _OtsInitParallel. Invoke as an integer function. See Section D.2.1.
_OtsGetNumThreads mpc_numthreads	Return the number of threads that are being used in the current parallel region (if running within one), or the number of threads that have been created so far (if not currently within a parallel region). Invoke as an integer function. See Section D.2.2.
_OtsGetThreadNum mpc_my_threadnum	Return a number that identifies the current thread. The main thread is 0, and slave threads are numbered densely from 1. Invoke as an integer function. See Section D.2.3.
_OtsInitParallel	Start slave threads for parallel processing if they have not yet been started implicitly (normally, the threads have been started by default at the first parallel region). Call as a subroutine with two arguments (see Section D.2.4): • The total number of threads desired (or specify zero to allow use of the environment variable MP_THREAD_COUNT or maximum number of processors). • A pointer to a pthreads attribute block, which can be used to control the attributes of the slave threads.
_OtsInParallel mpc_in_parallel_region	Return 1 if you are currently within a parallel region, or 0 if not. Invoke as an integer function. See Section D.2.5.
_OtsSetNumThreads	Sets the number of threads to use for the next parallel region.
_OtsStopWorkers mpc_destroy	Stop any slave threads created by parallel library support. This routine cannot be called from within a parallel region. After this call, new slave threads will be implicitly created the next time a parallel region is encountered, or can be created explicitly by calling _OtsInitParallel. Call as a subroutine. See Section D.2.7.

To call the _Ots*xxx* or mpc_*xxx* routines, use the cDEC$ ALIAS directive (described in the *Compaq Fortran Language Reference Manual*) to handle the mixed-case naming convention and missing trailing underscore.

For example, to call the _OtsGetThreadNum routine with an alias of OtsGetThreadNum, use the following code:

```
      integer a(10)
      INTERFACE
          INTEGER FUNCTION OtsGetThreadNum ()
!DEC$     ALIAS OtsGetThreadNum, '_OtsGetThreadNum'
          END FUNCTION OtsGetThreadNum
      END INTERFACE

!$par parallel do
      do i = 1,10
      print *, "i=",i, "  thread=", OtsGetThreadNum ()
      enddo

      end
```

Fortran INTERFACE blocks for all of the _Ots*xxx* routines are in a file named forompdef.f in /usr/include. Add the following line to your program and you can use the Fortran name ots*xxx* to call any of the _Ots*xxx* routines:

```
USE 'forompdef.f'
```

Alternatively, to use the compatibility naming convention of mpc_my_threadnum:

```
      integer a(10)
      INTERFACE
          INTEGER FUNCTION mpc_my_threadnum ()
!DEC$     ALIAS mpc_my_threadnum, 'mpc_my_threadnum'
          END FUNCTION mpc_my_threadnum
      END INTERFACE

!$par parallel do
      do i = 1,10
         print *, "i=",i, "  thread=", mpc_my_threadnum ()
      enddo

      end
```

These parallel threads are described in detail in the following sections.

See Also:

Section 6.1.3, Parallel Processing Thread Model

D.2.1 _OtsGetMaxThreads or mpc_maxnumthreads

Returns the maximum number of threads for the current environment.

Syntax:

```
      INTERFACE
          INTEGER FUNCTION otsgetmaxthreads ()
!DEC$     ALIAS otsgetmaxthreads, '_OtsGetMaxThreads'
          END FUNCTION otsgetmaxthreads
      END INTERFACE
      INTEGER result
      result = otsgetmaxthreads ()
```

Description:

Returns the number of threads that would normally be used for parallel processing in the current environment. This is affected by the environment variable MP_THREAD_COUNT, by the number of processes in the current process's processor set, and by any call to _OtsInitParallel.

D.2.2 _OtsGetNumThreads or mpc_numthreads

Returns the number of threads being used (in a parallel region) or created so far (if not in a parallel region).

Syntax:

```
      INTERFACE
          INTEGER FUNCTION otsgetnumthreads ()
!DEC$     ALIAS otsgetnumthreads, '_OtsGetNumThreads'
          END FUNCTION otsgetnumthreads
      END INTERFACE
      INTEGER result
      result = otsgetnumthreads ()
```

Description:

Returns the number of threads that are being used in the current parallel region (if running within one), or the number of threads that have been created so far (if not currently within a parallel region). You can use this call to decide how to partition a parallel loop. For example:

```
      nt = otsgetnumthreads ()
c$par parallel do
      do i = a,nt-1
            work(i) = 0
            k0 = 1+(i*n)/nt
            k1 = ((i+1)+n)/nt
            do j = 1,m
                  do k = k0,k1
                     ! use work(i)
                  enddo
            enddo
      enddo
```

D.2.3 _OtsGetThreadNum or mpc_my_threadnum

Returns the number of the current thread.

Syntax:

```
      INTERFACE
          INTEGER FUNCTION otsgetthreadnum ()
!DEC$     ALIAS otsgetthreadnum, '_OtsGetThreadNum'
          END FUNCTION otsgetthreadnum
      END INTERFACE
      INTEGER result
      result = otsgetthreadnum ()
```

Description:

Returns a number that identifies the current thread. The main thread is 0, and slave threads are numbered densely from 1.

D.2.4 _OtsInitParallel

Starts slave threads.

Syntax:

```
      INTERFACE
          SUBROUTINE otsinitparallel (nthreads, attr)
!DEC$     ALIAS otsinitparallel, '_OtsInitParallel'
          INTEGER              nthreads
          INTEGER              (KIND=8) attr
!DEC$     ATRRIBUTES, VALUE :: nthreads, attr
          END SUBROUTINE otsinitparallel
      END INTERFACE
```

Description:

Starts slave threads for parallel processing if they have not yet been started implicitly. Use this routine if you want to:

- Override number of threads
- Override the thread attributes
- Control when thread creation occurs (by default, at the first parallel region)

The arguments are:

- *nthreads* is the total number of threads desired, including the master. If *nthreads* is zero, the number of threads is controlled by the environment variable `MP_THREAD_COUNT`, if it is defined as a nonzero number, or by the number of processors in the current process's processor set. (See the `processor_sets`(3) reference page.)

- *attr* is a pointer to a pthreads attribute block, which can be used to control the attributes of the slave threads. If it is zero, all defaults are used except that the slaves' stack size in bytes can be set by the environment variable MP_STACK_SIZE.

D.2.5 _OtsInParallel or mpc_in_parallel_region

Returns the current status of processing activity in a parallel region.

Syntax:

```
      INTERFACE
          INTEGER FUNCTION otsinparallel ()
!DEC$     ALIAS otsinparallel, '_OtsInParallel'
          END FUNCTION OtsInParallel
      END INTERFACE
      INTEGER result
      result = otsinparallel ()
```

Description:

The routine returns 1 if the program is currently running within a parallel region; otherwise it returns 0.

D.2.6 _OtsSetNumThreads

Sets the number of threads to use for the next parallel region.

D.2.7 _OtsStopWorkers or mpc_destroy

Stops slave threads.

Syntax:

```
      INTERFACE
          SUBROUTINE otsstopworkers ()
!DEC$     ALIAS otsstopworkers, '_OtsStopWorkers'
          END SUBROUTINE otsstopworkers
      END INTERFACE
      CALL otsstopworkers ()
```

Description:

Stop any slave threads created by parallel library support. Use this routine if you need to perform some operation, such as a call to fork( ), that cannot tolerate extra threads running in the process. This routine cannot be called from within a parallel region. After this call, new slave threads will be implicitly created the next time a parallel region is encountered, or can be created explicitly by calling _OtsInitParallel.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z