VMS Software

VSI Fortran User Manual

Document Number: DO-DFRTUM-01A

Publication Date: April 2024

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI Fortran Version 8.3-3 for OpenVMS

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI Fortran User Manual

Legal Notice

Preface

This manual describes the VSI Fortran compiler command, compiler, and run-time environment. This includes how to compile, link, execute, and debug VSI Fortran programs on systems with Itanium or Alpha processor architectures running the VSI OpenVMS operating system. It also describes performance guidelines, I/O and error-handling support, calling other procedures, and compatibility.

Note

In this manual, the term OpenVMS refers to both OpenVMS I64 and OpenVMS Alpha systems. If there are differences in the behavior of the VSI Fortran compiler on the two operating systems, those differences are noted in the text.

---

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual assumes that:

- You already have a basic understanding of the Fortran 90/95 language. Tutorial Fortran 90/95 language information is widely available in commercially published books (see the online release notes or the Preface of the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).
- You are familiar with the operating system commands used during program development and a text editor. Such information is available in the OpenVMS documentation set.
- You have access to the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/], which describes the VSI Fortran 90/95 language.

3. Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 introduces the programmer to the VSI Fortran compiler, its components, and related commands.
- Chapter 2 describes the FORTRAN command qualifiers in detail.
- Chapter 3 describes how to link and run a VSI Fortran program.
- Chapter 4 describes the OpenVMS Debugger and some special considerations involved in debugging Fortran programs. It also lists some relevant programming tools and commands.
- Chapter 5 describes ways to improve run-time performance, including general software environment recommendations, appropriate FORTRAN command qualifiers, data alignment, efficiently performing I/O and array operations, other efficient coding techniques, profiling, and optimization.

- Chapter 6 provides information on VSI Fortran I/O, including statement forms, file organizations, I/O record formats, file specifications, logical names, access modes, logical unit numbers, and efficient use of I/O.
- Chapter 7 lists run-time messages and describes how to control certain types of I/O errors in your I/O statements.
- Chapter 8 describes the native Fortran OpenVMS data types, including their numeric ranges, representation, and floating-point exceptional values. It also discusses the intrinsic data types used with numeric data.
- Chapter 9 describes how to access unformatted files containing numeric little endian and big endian data different than the format used in memory.
- Chapter 10 describes how to call routines and pass arguments to them.
- Chapter 11 describes how to utilize OpenVMS Record Management Services (RMS) from a VSI Fortran program.
- Chapter 12 describes how to access records using indexed sequential access.
- Chapter 13 gives an introduction on how to exchange and share data among both local and remote processes.
- Chapter 14 describes facilities that can be used to handle—in a structured and consistent fashion—special conditions (errors or program-generated status conditions) that occur in large programs with many program units.
- Chapter 15 provides information on the VSI Extended Math Library (VXML) (Alpha only), a comprehensive set of mathematical library routines callable from Fortran and other languages.
- Appendix A describes the differences between VSI Fortran on I64 systems and on Alpha systems.
- Appendix B describes the compatibility between VSI Fortran for OpenVMS systems and VSI Fortran on other platforms, especially Compaq Fortran 77 for OpenVMS systems.
- Appendix C describes diagnostic messages issued by the VSI Fortran compiler and lists and describes messages from the VSI Fortran Run-Time Library (RTL) system.
- Appendix D lists the VSI Fortran logical names recognized at compile-time and run-time.
- Appendix E identifies the VSI Fortran include files that define symbols for use in VSI Fortran programs.
- Appendix F contains examples of the use of a variety of system services.

Note

If you are reading the printed version of this manual, be aware that the version at the VSI Fortran Web site and the version on the Documentation CD-ROM from VSI may contain updated and/or corrected information.

4. Related Documents

The following documents are also useful:

- *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]

  Describes the VSI Fortran 90/95 source language for reference purposes, including the format and use of statements, intrinsic procedures, and other language elements.

- *VSI Fortran Installation Guide for OpenVMS I64 Systems* or *VSI Fortran Installation Guide for OpenVMS Alpha Systems*

  Explain how to install VSI Fortran.

- VSI Fortran online release notes

  Provide the most recent information on this version of VSI Fortran. You can view or print the online release notes from:

  ```
  SYS$HELP:FORTRAN.RELEASE_NOTES (text version)
  SYS$HELP:FORTRAN_RELEASE_NOTES.PS (PostScript version)
  ```

- VSI Fortran online DCL HELP

  Summarizes the VSI Fortran command-line qualifiers, explains run-time messages, and provides a quick-reference summary of language topics. To use online HELP, use this command:

  ```
  $ HELP FORTRAN
  ```

- *Intel Itanium Architecture Software Developer's Manual*

- Operating system documentation

  The operating system documentation set describes the DCL commands (such as LINK), OpenVMS routines (such as system services and run-time library routines), OpenVMS concepts, and other aspects of the programming environment.

  For OpenVMS systems, sources of programming information include the following:

  - *OpenVMS Programming Environment Manual*
  - *VSI OpenVMS Programming Concepts Manual*
  - *OpenVMS Programming Interfaces: Calling a System Routine*
  - *VSI OpenVMS Debugger Manual*
  - *Alpha Architecture Reference Manual*
  - *Alpha Architecture Handbook*

  For information on the documentation for the OpenVMS operating system, including a list of books in the programmer's kit, see the *Overview of OpenVMS Documentation*.

  OpenVMS VAX to OpenVMS Alpha porting information can be found in *Migrating an Application from OpenVMS VAX to OpenVMS Alpha*. (For Fortran-specific porting information, see Appendix B).

  OpenVMS Alpha to OpenVMS I64 porting information can be found in *Porting Applications from VSI OpenVMS Alpha to VSI OpenVMS Industry Standard 64 for Integrity Servers.*

You can also use online DCL HELP for various OpenVMS commands and most routines by typing HELP. For the Debugger (and other tools), type HELP after you invoke the Debugger. For information on operating system messages, use the HELP/MESSAGE command.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. Conventions

The following product names may appear in this manual:

- OpenVMS Industry Standard 64 for Integrity Servers
- OpenVMS I64
- I64

All three names — the longer form and the two abbreviated forms — refer to the version of the OpenVMS operating system that runs on the Intel ® Itanium ® architecture.

The following conventions might be used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the

	command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold type	Bold type represents the name of an argument, an attribute, or a reason.
italic type	Italic type indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TYPE	Uppercase type indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes — binary, octal, or hexadecimal — are explicitly indicated.
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 intrinsic data types as a group.
integer	This term refers to integer intrinsic data types as a group.
Fortran	This term refers to language information that is common to ANSI FORTRAN-77, ANSI/ISO Fortran 90, ANSI/ISO Fortran 95, and VSI Fortran 90.
Fortran 90	This term refers to language information that is common to ANSI/ISO Fortran 90 and VSI Fortran. For example, a new language feature introduced in the Fortran 90 standard.
Fortran 95	This term refers to language information that is common to ISO Fortran 95 and VSI Fortran. For example, a new language feature introduced in the Fortran 95 standard.
VSI Fortran	Unless otherwise specified, this term (formerly Compaq Fortran) refers to language information that is common to the Fortran 90 and 95 standards, and any VSI Fortran extensions, running on the OpenVMS operating system. Since the Fortran 90 standard is a superset of the FORTRAN-77 standard, VSI Fortran also supports the FORTRAN-77 standard. VSI Fortran supports all of the deleted features of the Fortran 95 standard.

Chapter 1. Getting Started

This chapter describes:

- Section 1.1: Fortran Standards Overview
- Section 1.2: VSI Fortran Programming Environment
- Section 1.3: Commands to Create and Run an Executable Program
- Section 1.4: Creating and Running a Program Using a Module and Separate Function
- Section 1.5: Program Development Stages and Tools

1.1. Fortran Standards Overview

VSI Fortran for OpenVMS (formerly Compaq Fortran for OpenVMS) conforms to the:

- American National Standard Fortran 90 (ANSI X3.198-1992), which is the same as the International Standards Organization standard (ISO/IEC 1539:1991 (E))
- Fortran 95 standard (ISO/IEC 1539:1998 (E))

VSI Fortran supports all of the deleted features of the Fortran 95 standard.

VSI 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.

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

VSI Fortran provides a number of extensions to the Fortran 90 and 95 standards. VSI Fortran extensions to the latest Fortran standard are generally provided for compatibility with Compaq Fortran 77 extensions to the ANSI FORTRAN-77 standard.

When creating new programs that need to be standards-conforming for portability reasons, you should avoid or minimize the use of extensions to the latest Fortran standard. Extensions to the appropriate Fortran standard are identified visually in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/], which defines the VSI Fortran language.

1.2. VSI Fortran Programming Environment

The following aspects of Fortran 90/95 are relevant to the compilation environment and should be considered before extensive coding begins:

- To install VSI Fortran on your system, see the *VSI Fortran Installation Guide for OpenVMS I64 Systems* or the *VSI Fortran Installation Guide for OpenVMS Alpha Systems*.

- Once VSI Fortran is installed, you can:
  - Use the FORTRAN command to compile source files. Use the LINK command to link object files into executable programs.
  - Use the online HELP FORTRAN command and this manual to provide information about the FORTRAN command.
- Make sure you have adequate process memory space, especially if your programs use large arrays as data. Your system manager (or designated privileged user) may be able to overcome this problem by checking and possibly increasing the following:
  - Your process memory (working set)

    Your system manager can use the Authorize Utility to adjust your process working set quotas, page file quota, and limits.
  - System-wide virtual memory limits

    Your system manager can use SYSGEN to change parameters (such as WSMAX and VIRTUALPAGECNT), which take effect after the system is rebooted.
  - Page file space on your system

    Your system manager can use SYSGEN or AUTOGEN to increase page file sizes or create new page files. Your system manager needs to INSTALL any new page files available to the system by modifying system startup command procedures and rebooting the system.
  - System hardware resources, such as physical memory and disk space

  You can check the current memory limits using the SHOW WORKING_SET command. To view peak memory use after compiling or running a program, use the SHOW PROCESS/ACCOUNTING command. Your system manager can use these commands (or SHOW PROCESS/CONTINUOUS) for a currently running process and the system-side MONITOR command.

  For example, the following DCL (shell) commands check the current limits and show the current use of some of these limits:

```
$ SHOW WORKING_SET
...
$ SHOW PROCESS/ACCOUNTING
...
```

- Make sure you have an adequate process open file 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 open file limit allows. In this case, the VSI Fortran compiler will close a previously opened module file before it opens another to stay within your open file limit. This results in slower compilation time. Increasing the open file limit may improve compilation time in such cases.

  You can view the per-process limit on the number of open files (Open file quota or FILLM) by using the SHOW PROCESS/QUOTA command:

```
$ SHOW PROCESS/QUOTA
...
```

Your system manager needs to determine the maximum per-process limit for your system by checking the value of the CHANNELCNT SYSGEN parameter and (if necessary) increasing its value.

- You can define logical names to specify libraries and directories.

  You can define the FORT$LIBRARY logical name to specify a user-defined text library that contains source text **library modules** referenced by INCLUDE statements. The compiler searches first for libraries specified on the command line and also in the system-supplied default library (see Section 2.4).

  For more information on using FORT$LIBRARY, see Section 2.2.4.

  You can define the FORT$INCLUDE logical name to specify a directory to be searched for the following files:

  - Module files specified by a USE statement (module name is used as a file name)
  - Source files specified by an INCLUDE statement, where a file name is specified without a directory name
  - Text library files specified by an INCLUDE statement, where a file name is specified without a library name

  For more information on the FORT$INCLUDE logical name, see Section 2.2.3 and Section 2.2.4.

  If you need to set logical names frequently, consider setting them in your LOGIN.COM file, or ask your system manager to set them as system-wide logical names in a system startup command procedure.

  Several other logical names can similarly be used during program execution (see Appendix D).

- Your VSI Fortran source files can be in free or fixed form. You can indicate the source form used in your source files by using certain file types or a command-line qualifier:

  - For files using fixed form, specify a file type of FOR or F.
  - For files using free form, specify a file type of F90.
  - You can also specify the /SOURCE_FORM qualifier on the FORTRAN command line to specify the source form for all files on that command line.

  For example, if you specify a file as PROJ_BL1.F90 on a FORTRAN command line (and omit the /SOURCE_FORM=FIXED qualifier), the FORTRAN command assumes the file PROJ_BL1.F90 contains free-form source code.

  A special type of fixed source form is tab form (a VSI extension described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

- Each source file to be compiled must contain at least one program unit (main program, subroutine, 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 application run-time performance is important, compile related source files together (or the entire application). When compiling multiple source files, separate file names with plus signs (+) to concatenate source files and create a single object file. This allows certain interprocedure optimizations to minimize run-time execution time (unless you specify certain qualifiers).

- Code reuse

  Modules, external subprograms, and included files allow reuse 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 2.2.3.
- On include files, see Section 2.2.4.
- On VSI Fortran source forms, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On recognized file types, see Section 2.2.1.
- On the types of subprograms and using an explicit interface to a subprogram, see Chapter 10.
- On performance considerations, including compiling source programs for optimal run-time performance, see Chapter 5.
- On logical names, see the *VSI OpenVMS User's Manual*.

1.3. Commands to Create and Run an Executable Program

Example 1.1 shows a short Fortran 90/95 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 the Extensible Versatile Editor (EVE). For instance, to use EVE to edit the file HELLO.F90, enter:

```
$ EDIT HELLO.F90
```

The following FORTRAN command compiles the program named HELLO.F90. The LINK command links the compiled object file into an executable program file named HELLO.EXE:

```
$ FORTRAN HELLO.F90
$LINK HELLO
```

In this example, because all external routines used by this program reside in standard OpenVMS libraries searched by the LINK command, additional libraries or object files are not specified on the LINK command line.

To run the program, enter the RUN command and the program name:

```
$ RUN HELLO
```

If the executable program is not in your current default directory, specify the directory before the file name. Similarly, if the executable program resides on a different device than your current default device, specify the device name and directory name before the file name.

For More Information:

- On the OpenVMS programming environment, see the operating system documents listed in the Preface of this manual.
- On specifying files on an OpenVMS system, see the *VSI OpenVMS User's Manual*.

1.4. Creating and Running a Program Using a Module and Separate Function

Example 1.2 shows a sample VSI Fortran main program using free-form source 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  (*,'(BN,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 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
```

1.4.1. Commands to Create the Executable Program

During the early stages of program development, the three files might be compiled separately and then linked together, using the following commands:

```
$ FORTRAN ARRAY_CALC.F90
$ FORTRAN CALC_AVER.F90
$ FORTRAN MAIN.F90
$ LINK/EXECUTABLE=CALC.EXE MAIN, ARRAY_CALC, CALC_AVER
```

In this sequence of FORTRAN commands:

- Each of the FORTRAN commands creates one object file (OBJ file type).
- The first FORTRAN command creates the object file ARRAY_CALC.OBJ and the module file ARRAY_CALCULATOR.F90$MOD. The name in the MODULE statement (ARRAY_CALCULATOR) in Example 1.3 determines the file name of the module file. The FORTRAN command creates module files in the process default device and directory with a F90$MOD file type.

- The second FORTRAN command creates the file CALC_AVER.OBJ (Example 1.4).
- The third FORTRAN command creates the file MAIN.OBJ (Example 1.2) and uses the module file ARRAY_CALCULATOR.F90$MOD.
- The LINK command links all object files (OBJ file type) into the executable program named CALC.EXE.

To allow more optimizations to occur (such as the inline expansion of called subprograms), compile the entire set of three source files together using a single FORTRAN command:

```
$ FORTRAN/OBJECT=CALC.OBJ ARRAY_CALC.F90 + CALC_AVER.F90 + MAIN.F90
```

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

- Compiles the file ARRAY_CALC.F90, which contains the module definition (shown in Example 1.3), and creates its object file and the file ARRAY_CALCULATOR.F90$MOD in the process default device and directory.
- Compiles the file CALC_AVER.F90, which contains the external function CALC_AVERAGE (shown in Example 1.4).
- Compiles the file MAIN.F90 (shown in Example 1.2). The USE statement references the module file ARRAY_CALCULATOR.F90$MOD.
- Creates a single object file named CALC.OBJ.

When you omit the file type on the FORTRAN command line, the FORTRAN command searches for a file with the F90 file type before a file with the FOR or F file type, so you can enter the previous command (without file types) as follows:

```
$ FORTRAN/OBJECT=CALC.OBJ ARRAY_CALC + CALC_AVER + MAIN
```

Use a LINK command to link the single object file into an executable program:

```
$ LINK CALC
```

When you omit the file type on the LINK command line, the Linker searches for a file with a file type of OBJ. Unless you will specify a library on the LINK command line, you can omit the OBJ file type.

1.4.2. Running the Sample Program

If the current default directory contains the file named CALC you can run the program by entering the RUN command followed by its name:

```
$ RUN CALC
```

When you omit the file type on the RUN command line, the image activator searches for a file with a file type of EXE (you can omit the EXE file type).

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
```

1.4.3. Debugging the Sample Program

To debug a program using the OpenVMS Debugger, compile and link with the /DEBUG qualifier to request additional symbol table information for source line debugging in the object and executable program files. The following FORTRAN command names the object file CALC_DEBUG.OBJ. The LINK command then creates the program file CALC_DEBUG.EXE with full debugging information:

```
$ FORTRAN/DEBUG/OBJECT=CALC_DEBUG.OBJ/NOOPTIMIZE ARRAY_CALC + CALC_AVER
  + MAIN
$ LINK/DEBUG CALC_DEBUG
```

The OpenVMS debugger has a character-cell interface and a windowing interface (available with the DECwindows Motif product). To debug an executable program named CALC_DEBUG.EXE, enter the following command:

```
$ RUN CALC_DEBUG
```

For more information on running the program within the debugger and the windowing interface, see Chapter 4.

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 or appropriate commercially published documentation.

VSI Fortran provides the standard features of a compiler and the OpenVMS operating system provides a linker.

Use a LINK command to link the object file into an executable program.

Table 1.1 describes some of the software tools you can use when developing (implementing) and testing a program:

Table 1.1. Tools for Program Development and Testing

Task or Activity	Tool and Description
Manage source files	Use the Code Management System (CMS).
Create and modify source files	Use a text editor, such as the EDIT command to use the EVE editor. You can also use the optional Language-Sensitive Editor (LSE). For more information using OpenVMS text editors, see the *VSI OpenVMS User's Manual*.
Analyze source code	Use DCL commands such as SEARCH and DIFFERENCES.
Build program (compile and link)	You can use the FORTRAN and LINK commands to create small programs, perhaps using command procedures, or use the Module Management System (MMS) to build your application in an automated fashion.

Task or Activity	Tool and Description
	For more information on the FORTRAN and LINK commands, see Chapter 2 and Chapter 3 respectively.
Debug and Test program	Use the OpenVMS Debugger to debug your program or run it for general testing. For more information on the OpenVMS Debugger, see Chapter 4 in this manual.
Analyze performance	To perform program timings and profiling of code, use the LIB$ `xxxx`_TIMER routines, a command procedure, or the Performance Coverage Analyzer (PCA). For more information on timing and profiling VSI Fortran code, see Chapter 5.

To perform other program development functions at various stages of program development, use the following DCL commands:

- Use the LIBRARIAN command to create an object or text library, add or delete library modules in a library, list the library modules in a library, and perform other functions. For more information, enter HELP LIBRARIAN or see the *VMS Librarian Utility Manual*.
- Use the LINK/NODEBUG command to remove symbolic and other debugging information to minimize image size. For more information, see Chapter 3.
- The LINK/MAP command creates a link map, which shows information about the program sections, symbols, image section, and other information.
- The ANALYZE/OBJECT command shows which compiler compiled the object file and the version number used. It also does a partial error analysis and shows other information.
- The ANALYZE/IMAGE command checks information about an executable program file. It also shows the date it was linked and the version of the operating system used to link it.

For More Information:

- On the OpenVMS programming environment, see the Preface of this manual.
- On DCL commands, use DCL HELP or see the *VSI OpenVMS DCL Dictionary*.

Chapter 2. Compiling VSI Fortran Programs

This chapter describes:

- Section 2.1: Functions of the Compiler
- Section 2.2: FORTRAN Command Syntax, Use, and Examples
- Section 2.3: FORTRAN Command Qualifiers
- Section 2.4: Creating and Maintaining Text Libraries
- Section 2.5: Using CDD/Repository
- Section 2.6: Compiler Limits, Diagnostic Messages, and Error Conditions
- Section 2.7: Compiler Output Listing Format

2.1. Functions of the Compiler

The primary functions of the VSI Fortran compiler are to:

- Verify the VSI Fortran source statements and to issue messages if the source statements contain any errors
- Venerate machine language instructions from the source statements of the VSI Fortran program
- Group these instructions into an object module for the OpenVMS Linker

When the compiler creates an object file, it provides the linker with the following information:

- The program unit name. The program unit name is the name specified in the PROGRAM, MODULE, SUBROUTINE, FUNCTION, or BLOCK DATA statement in the source program. If a program unit does not contain any of these statements, the source file name is used with $MAIN (or $DATA, for block data subprograms) appended.
- A list of all entry points and common block names that are declared in the program unit. 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.
- Traceback information, used by the system default condition handler when an error occurs that is not handled by the program itself. The traceback information permits the default handler to display a list of the active program units in the order of activation, which aids program debugging.
- A symbol table, if specifically requested (/DEBUG qualifier). A symbol table lists the names of all external and internal variables within a object module, with definitions of their locations. The table is of primary use in program debugging.

For More Information:

On the OpenVMS Linker, see Chapter 3.

2.2. FORTRAN Command Syntax, Use, and Examples

The FORTRAN command initiates compilation of a source program.

The command has the following form:

```
FORTRAN [/qualifiers] file-spec-list[/qualifiers]
```

/qualifiers

Indicates either special actions to be performed by the compiler or special properties of input or output files.

file-spec-list

Specifies the source files containing the program units to be compiled. You can specify more than one source file:

- If source file specifications are separated by commas (,), the programs are compiled separately.
- If source file specifications are separated by plus signs (+), the files are concatenated and compiled as one program.

When compiling source files with the default optimization (or additional optimizations), concatenating source files allows full interprocedure optimizations to occur.

In interactive mode, you can also enter the file specification on a separate line by entering the command FORTRAN, followed by a carriage return. The system responds with the following prompt:

```
_File:
```

Enter the file specification immediately after the prompt and then press Return.

2.2.1. Specifying Input Files and Source Form

If you omit the file type on the FORTRAN command line, the compiler searches first for a file with a file type of F90. If a file with a file type of F90 is not found, it then searches for file with a file type of FOR and then F.

For example, the following FORTRAN command line shows how file type searching occurs:

```
$ FORTRAN PROJ_ABC
```

This FORTRAN command searches for the following files:

1. It searches first for PROJ_ABC.F90.
2. If PROJ_ABC.F90 does not exist, it then searches for PROJ_ABC.FOR.
3. If PROJ_ABC.F90 and PROJ_ABC.FOR do not exist, it then searches for PROJ_ABC.F.

Indicate the Fortran 90 source form used in your source files by using certain file types or a command-line qualifier:

- For files using fixed form, use a file type of FOR or F.
- For files using free form, use a file type of F90.
- You can also specify the /SOURCE_FORM qualifier on the FORTRAN command line to specify the source form (FIXED or FREE) for:
  - All files on that command line (when used as a command qualifier)
  - Individual source files in a comma-separated list of files (when used as a positional qualifier)

For example, if you specify a file as PROJ_BL1.F90 on an FORTRAN command line (and omit the /SOURCE_FORM=FIXED qualifier), the FORTRAN command assumes the file PROJ_BL1.F90 contains free-form source code.

2.2.2. Specifying Multiple Input Files

When you specify a list of input files on the FORTRAN command line, you can use abbreviated file specifications for those files that share common device names, directory names, or file names.

The system applies temporary file specification defaults to those files with incomplete specifications. The defaults applied to an incomplete file specification are based on the previous device name, directory name, or file name encountered in the list.

The following FORTRAN command line shows how temporary defaults are applied to a list of file specifications:

```
$ FORTRAN USR1:[ADAMS]T1,T2,[JACKSON]SUMMARY,USR3:[FINAL]
```

The preceding FORTRAN command compiles the following files:

```
USR1:[ADAMS]T1.F90 (or .FOR or .F)
USR1:[ADAMS]T2.F90 (or .FOR  or .F)
USR1:[JACKSON]SUMMARY.F90 (or .FOR or .F)
USR3:[FINAL]SUMMARY.F90 (or .FOR or .F)
```

To override a temporary default with your current default directory, specify the directory as a null value. For example:

```
$ FORTRAN [OMEGA]T1, []T2
```

The empty brackets indicate that the compiler is to use your current default directory to locate T2.

FORTRAN qualifiers typically apply to the entire FORTRAN command line. One exception is the /LIBRARY qualifier, which specifies that the file specification it follows is a text library (positional qualifier). The /LIBRARY qualifier is discussed in Section 2.3.27.

You can specify multiple files on the FORTRAN command line. You can separate the multiple source file specifications with:

- Plus signs (+)

  All files separated by plus signs are concatenated and compiled as one program into a single object file.

  A positional qualifier after one of the file specifications applies to *all* files concatenated by plus signs (+). One exception is the /LIBRARY qualifier (used only as a positional qualifier).

Concatenating source files allows full interprocedure optimizations to occur across the multiple source files (unless you specify certain command qualifiers, such as /NOOPTIMIZE and /SEPARATE_COMPILATION).

- Commas (,)

  When separated by commas, the files are compiled separately into multiple object files.

  A positional qualifier applies only to the file specification it immediately follows.

  Separate compilation by using comma-separated files (or by using multiple FORTRAN commands) prevents certain interprocedure optimizations.

If you use multiple FORTRAN commands to compile multiple VSI Fortran source files that are linked together into the same program, you must be consistent when specifying any qualifiers that affect run-time results. For example, suppose you do the following:

1. Specify /FLOAT=IEEE_FLOAT on one command line
2. Specify /FLOAT=G_FLOAT on another command line
3. Link the resulting object files together into a program

When you run the program, the values returned for floating-point numbers in a COMMON block will be unpredictable. For qualifiers related only to compile-time behavior (such as /LIST and /SHOW), this restriction does not apply.

2.2.3. Creating and Using Module Files

VSI Fortran creates **module files** for each module declaration and automatically searches for a module file referenced by a USE statement (introduced in Fortran 90). A module file contains the equivalent of the module source declaration in a post-compiled, binary form.

2.2.3.1. Creating Module Files

When you compile a VSI Fortran source file that contains module declarations, VSI Fortran creates a separate file for each module declaration in the current process default device and directory. The name declared in a MODULE statement becomes the base prefix of the file name and is followed by the F90$MOD file type.

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

```
MODULE MOD1
```

The compiler creates a post-compiled module file MOD1.F90$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 need to create module files only, specify the /NOOBJECT qualifier to prevent object file creation.

To specify a directory other than the current directory for the module file(s) to be placed, use the /MODULE qualifier (see Section 2.3.31).

Note that an object file is not needed if there are only INTERFACE or constant (PARAMETER) declarations; however, it is needed for all other types of declarations including variables.

2.2.3.2. Using Module Files

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

```
USE MOD1
```

By default, the compiler searches for a module file named MOD1.F90$MOD in the current directory.

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

- Whether the module files need to be available privately, such as 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 or a different directory for module files.

VSI Fortran allows you to use multiple methods to specify which directories are searched for module files:

- You can specify one or more additional directories for the compiler to search by using the /INCLUDE= `directory` qualifier.
- You can specify one or more additional directories for the compiler to search by defining the logical name FORT$INCLUDE. To search multiple additional directories with FORT$INCLUDE, define it as a search list. Like other logical names, FORT$INCLUDE can be for a system-wide location (such as a group or system logical name) or a private or project location (such as a process, job, or group logical name).
- You can prevent the compiler from searching in the directory specified by the FORT$INCLUDE logical name by using the /NOINCLUDE qualifier.

To locate modules referenced by USE statements, the compiler searches directories in the following order:

1. The current process device and directory
2. Each directory specified by the /INCLUDE qualifier
3. The directory specified by the logical name FORT$INCLUDE (unless /NOINCLUDE was specified).

You cannot specify a module (.F90$MOD) file directly on the FORTRAN command line.

Suppose you need to compile a main program PROJ_M.F90 that contains one or more USE statements. To request that the compiler look for module files in the additional directories DISKA:[PROJ_MODULE.F90] and then DISKB:[COMMON.FORT] (after looking in the current directory), enter the following command line:

```
$ FORTRAN PROJ_M.F90 /INCLUDE=(DISKA:[PROJ_MODULE.F90],DISKB:[COMMON.FORT])
```

If you specify multiple directories with the /INCLUDE qualifier, the order of the directories in the /INCLUDE qualifier determines the directory search order.

Module nesting depth is unlimited. If you will use many modules in a program, check the process and system file limits (see Section 1.2).

For More Information:

- On the FORTRAN /INCLUDE qualifier, see Section 2.3.25.
- On an example program that uses a module, see Section 1.4.

2.2.4. Using Include Files and Include Text Library Modules

You can create include files with a text editor. The include files can be placed in a text library. If needed, you can copy include files or include text library 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 or text library needs to be available privately, such as 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 or a different directory.
- Instead of placing include files in a directory, consider placing them in a text library. The text library can contain multiple include files in a single library file and is maintained by using the OpenVMS LIBRARY command.

2.2.4.1. Using Include Files and INCLUDE Statement Forms

Include files have a file type like other VSI Fortran source files (F90, FOR, or F). Use an INCLUDE statement to request that the specified file containing source lines be included in place of the INCLUDE statement.

To include a file, the INCLUDE statement has the following form:

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

You can specify /LIST or /NOLIST after the file name. You can also specify the /SHOW=INCLUDE or /SHOW=NOINCLUDE qualifier to control whether source lines from included files or library modules appear in the listing file (see Section 2.3.43).

You can also include a file with a directory (and optionally the device name) specified with the following form:

```
INCLUDE '[directory]name'
INCLUDE '[directory]name.typ'
```

If a directory is specified, only the specified directory is searched. The remainder of this section addresses an INCLUDE statement where the directory has not been specified.

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

- You can request that the VSI Fortran compiler search either in the current process default directory or in the directory where the source file resides that references the include file. To do this, specify the /ASSUME=NOSOURCE_INCLUDE (process default directory) or /ASSUME=SOURCE_INCLUDE qualifier (source file directory). The default is /ASSUME=NOSOURCE_INCLUDE.
- You can specify one or more additional directories for the compiler to search by using the /INCLUDE= `directory` qualifier.
- You can specify one or more additional directories for the compiler to search by defining the logical name FORT$INCLUDE. To search multiple additional directories with FORT$INCLUDE, define it as a search list. Like other logical names, FORT$INCLUDE can be for:
  - A system-wide location (such as a group or system logical name)
  - A private or project location (such as a process, job, or group logical name)
- You can prevent the compiler from searching in the directory specified by the FORT$INCLUDE logical name by using the /NOINCLUDE qualifier.

To locate include files specified in INCLUDE statements (without a device or directory name), the VSI Fortran compiler searches directories in the following order:

1. The current process default directory or the directory that the source file resides in (depending on whether /ASSUME=SOURCE_INCLUDE was specified)
2. Each directory specified by the /INCLUDE qualifier
3. The directory specified by the logical name FORT$INCLUDE (unless /NOINCLUDE was specified).

2.2.4.2. INCLUDE Statement Forms for Including Text Library Modules

VSI Fortran provides certain include library modules in the text library FORSYSDEF.TLB. Users can create a text library and populate it with include library modules (see Section 2.4). Within a library, text library modules are identified by a library module name (no file type).

To include a text library module, the INCLUDE statement specifies the name of the library module within parentheses, as follows:

```
INCLUDE '(name)'
```

You can specify the library name before the library module name in the INCLUDE statement. For example:

```
INCLUDE 'MYLIB(PROJINC)'
```

Use one of the following methods to access a source library module in a text library:

- Specify only the name of the library module in an INCLUDE statement in your VSI Fortran source program. You use FORTRAN command qualifiers and logical names to control the directory search for the library.
- Specify the name of both the library and library module in an INCLUDE statement in your VSI Fortran source program.
- When the INCLUDE statement does not specify the library name, you can define a default library by using the logical name FORT$LIBRARY.
- When the INCLUDE statement does not specify the library name, you can specify the name of the library using the /LIBRARY qualifier on the FORTRAN command line that you use to compile the source program.

2.2.4.3. Using Include Text Library Modules for a Specified Library Name

When the library is named in the INCLUDE statement, the FORTRAN command searches various directories for the named library, similar to the search for an include file.

VSI Fortran allows you to use multiple methods to specify which directories are searched for named text libraries:

- You can request that the VSI Fortran compiler search either in the current process default directory or in the directory where the source file resides that references the text library. To do this, specify the /ASSUME=NOSOURCE_INCLUDE (process default directory) or /ASSUME=SOURCE_INCLUDE qualifier (source file directory).
- You can specify one or more additional directories for the compiler to search by using the /INCLUDE= `directory` qualifier.
- You can specify one or more additional directories for the compile to search by defining the logical name FORT$INCLUDE. To search multiple additional directories with FORT$INCLUDE, define it as a search list. Like other logical names, FORT$INCLUDE can be for:
  - A system-wide location (such as a group or system logical name)
  - A private or project location (such as a process, job, or group logical name)
- You can prevent the compiler from searching in the directory specified by the FORT$INCLUDE logical name by using the /NOINCLUDE qualifier.

The VSI Fortran compiler searches directories in the following order:

1. The current process default directory or the directory that the source file resides in (depending on whether /ASSUME=SOURCE_INCLUDE was specified)
2. Each directory specified by the /INCLUDE qualifier
3. The directory specified by the logical name FORT$INCLUDE (unless /NOINCLUDE was specified).

You can specify /LIST or /NOLIST after the library module name. For example:

```
INCLUDE 'PROJLIB(MYINC)/LIST'
```

You can also specify the /SHOW=INCLUDE or /SHOW=NOINCLUDE qualifier to control whether source lines from included files or library modules appear in the listing file (see Section 2.3.43).

For More Information:

- On the /ASSUME=[NO]SOURCE_INCLUDE qualifier, see Section 2.3.7.
- On the /INCLUDE qualifier, see Section 2.3.25.
- On creating and using text libraries, see Section 2.4.

2.2.4.4. Using Include Text Library Modules for an Unspecified Library Name

When the INCLUDE statement does not specify the library, you can specify additional text libraries to be searched on the FORTRAN command line or by defining a logical name. The order in which the compiler searches for a library file follows:

- Specify the library name on the FORTRAN command line, appended with the /LIBRARY positional qualifier. The /LIBRARY qualifier identifies a file specification as a text library.

  Concatenate the name of the text library to the name of the source file and append the /LIBRARY qualifier to the text library name (use the plus sign (+)) separator. For example:

  ```
  $ FORTRAN APPLIC+DATAB/LIBRARY
  ```

  Whenever an INCLUDE statement occurs in APPLIC.FOR, the compiler searches the library DATAB.TLB for the source text module identified in the INCLUDE statement and incorporates it into the compilation.

  When more than one library is specified on a FORTRAN command line, the VSI Fortran compiler searches the libraries each time it processes an INCLUDE statement that specifies a text module name. The compiler searches the libraries in the order specified on the command line. For example:

  ```
  $ FORTRAN APPLIC+DATAB/LIBRARY+NAMES/LIBRARY+GLOBALSYMS/LIBRARY
  ```

  When the VSI Fortran compiler processes an INCLUDE statement in the source file APPLIC.FOR, it searches the libraries DATAB.TLB, NAMES.TLB, and GLOBALSYMS.TLB, in that order, for source text modules identified in the INCLUDE statement.

  When the FORTRAN command requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

  ```
  $ FORTRAN METRIC+DATAB/LIBRARY, APPLIC+DATAB/LIBRARY
  ```

  In this example, VSI Fortran compiles METRIC.FOR and APPLIC.FOR separately and uses the library DATAB.TLB for each compilation.

  If the text library is not in the current process default directory, specify the device and/or directory. For example:

  ```
  $ FORTRAN PROJ_MAIN+$DISK2:[PROJ.LIBS]COMMON_LIB/LIBRARY
  ```

  Instead of specifying the device and directory name, you can use the /ASSUME=SOURCE_INCLUDE and /INCLUDE qualifiers and FORT$INCLUDE logical name to control the directory search.

- After the compiler has searched all libraries specified on the command line, it searches the default user library (if any) specified by the logical name FORT$LIBRARY. When you want to define one of your private text libraries as a default library for the VSI Fortran compiler to search, consider using the FORT$LIBRARY logical name.

  For example, define a default library using the logical name FORT$LIBRARY before compilation, as in the following example of the DCL command DEFINE:

  ```
  $ DEFINE FORT$LIBRARY $DISK2:[LIB]DATAB
  $ FORTRAN PROJ_MAIN
  ```

  While this assignment is in effect, the compiler automatically searches the library $DISK2:[LIB]DATAB.TLB for any include library modules that it cannot locate in libraries explicitly specified, if any, on the FORTRAN command line.

  You can define the logical name FORT$LIBRARY in any logical name table defined in the logical name table search list LMN$FILE_DEV. For example:

  ```
  $ DEFINE /GROUP FORT$LIBRARY $DISK2:[PROJ.LIBS]APPLIB.TLB
  $ FORTRAN PROJ_MAIN
  ```

  If the name is defined in more than one table, the VSI Fortran compiler uses the equivalence for the first match it finds in the normal order of search – first the process table, then intermediate tables (job, group, and so on), and finally the system table. If the same logical name is defined in both the process and system logical name tables, the process logical name table assignment overrides the system logical name table assignment.

  If FORT$LIBRARY is defined as a search list, the compiler opens the first text library specified in the list. If the include library module is not found in that text library, the search is terminated and an error message is issued.

  The logical name FORT$LIBRARY is recognized by both Compaq Fortran 77 and VSI Fortran.

- When the VSI Fortran compiler cannot find the include library modules in libraries specified on the FORTRAN command line or in the default library defined by FORT$LIBRARY, it then searches the standard text library supplied by VSI Fortran. This library resides in SYS$LIBRARY with a file name of FORSYSDEF.TLB and simplifies calling OpenVMS system services.

  SYS$LIBRARY identifies the device and directory containing system libraries and is normally defined by the system manager. FORSYSDEF.TLB is a library of include library modules supplied by VSI Fortran. It contains local symbol definitions and structures required for use with system services and return status values from system services.

  For more information on the contents of FORSYSDEF, see Appendix E.

You can specify /LIST or /NOLIST after the library module name. For example:

```
INCLUDE '(MYINC)/NOLIST'
```

You can also specify the /SHOW=INCLUDE or /SHOW=NOINCLUDE qualifier to control whether source lines from included files or library modules appear in the listing file (see Section 2.3.43).

For More Information:

- On the /ASSUME=[NO]SOURCE_INCLUDE qualifier, see Section 2.3.7.
- On the /INCLUDE qualifier, see Section 2.3.25.

- On creating and using text libraries, see Section 2.4.

2.2.5. Specifying Output Files

The output produced by the compiler includes the object and listing files. You can control the production of these files by using the appropriate qualifiers on the FORTRAN command line.

The production of listing files depends on whether you are operating in interactive mode or batch mode:

- In interactive mode, the compiler does not generate listing files by default; you must use the /LIST qualifier to generate the listing file.
- In batch mode, the compiler generates a listing file by default. To suppress it, you must use the /NOLIST qualifier.

For command procedures that compile the application in either batch or interactive mode, consider explicitly specifying /NOLIST (or /LIST).

The compiler generates an object file by default. During the early stages of program development, you may find it helpful to use the /SYNTAX_ONLY or /NOOBJECT qualifiers to prevent the creation of object files until your source program compiles without errors. If you omit /NOOBJECT, the compiler generates object files as follows:

- If you specify one source file, one object file is generated.
- If you specify multiple source files separated by commas, each source file is compiled separately, and an object file is generated for each source file.
- If you specify multiple source files separated by plus signs, the source files are concatenated and compiled, and one object file is generated.

You can use both commas and plus signs in the same command line to produce different combinations of concatenated and separate object files (see the examples of the FORTRAN command at the end of this section).

Otherwise, the object file has the file name of its corresponding source file and a file type of OBJ.To name an object file, use the /OBJECT qualifier in the form /OBJECT= *file-spec*.By default, the object file produced from concatenated source files has the name of the first source file. All other file specification fields (node, device, directory, and version) assume the default values.

When creating object files that will be placed in an object library, consider using the /SEPARATE_COMPILATION qualifier, which places individual compilation units in a source file as separate components in the object file. This minimizes the size of the routines included by the linker as it creates the executable image. However, to allow more interprocedure optimizations, use the default /NOSEPARATE_COMPILATION.

For More Information:

- On creating and naming object files, see Section 2.2.6.1 and Section 2.3.33.
- On the /SEPARATE_COMPILATION qualifier, see Section 2.3.41.
- On using object libraries, see Chapter 3.

2.2.6. Examples of the FORTRAN Command

The following examples show the use of the FORTRAN command.

2.2.6.1. Naming the Object File

The following FORTRAN command compiles the VSI Fortran free-form source file (CIRCLE.F90) into an object file:

```
$ FORTRAN /OBJECT=[BUILD]SQUARE /NOLIST Return
_File: CIRCLE
```

The source file CIRCLE.F90 is compiled, producing an object file named SQUARE.OBJ in the [BUILD] directory, but no listing file.

2.2.6.2. Compiler Source Checking Only (No Object File)

The following FORTRAN command examines VSI Fortran fixed-form source file (ABC.FOR) without creating an object file:

```
$ FORTRAN /NOOBJECT ABC.FOR
```

The source file ABC.FOR is compiled, syntax checking occurs, but no object file is produced. The /NOOBJECT qualifier performs full compilation checking without creating an object file.

2.2.6.3. Requesting a Listing File and Contents

The following FORTRAN command compiles VSI Fortran free-form source file (XYZ.F90) and requests a listing file

```
$ FORTRAN /LIST /SHOW=INCLUDE /MACHINE_CODE XYZ.F90
```

The source file XYZ.F90 is compiled, and the listing file XYZ.LIS and object file XYZ.OBJ are created. The listing file contains the optional included source files (/SHOW=INCLUDE) and machine code representation (/MACHINE_CODE).

The default is /NOLIST for interactive use and /LIST for batch use, so consider explicitly specifying either /LIST or /NOLIST.

2.2.6.4. Compiling Multiple Files

The following FORTRAN command compiles the VSI Fortran free-form source files (AAA.F90, BBB.F90, and CCC.F90) into separate object files:

```
$ FORTRAN/LIST AAA.F90, BBB.F90, CCC.F90
```

Source files AAA.F90, BBB.F90, and CCC.F90 are compiled as separate files, producing:

- Object files named AAA.OBJ, BBB.OBJ, and CCC.OBJ
- Listing files named AAA.LIS, BBB.LIS, and CCC.LIS

The default level of optimization is used (/OPTIMIZE=LEVEL=4), but interprocedure optimizations are less effective because the source files are compiled separately.

The following FORTRAN command compiles the VSI Fortran fixed-form source files XXX.FOR, YYY.FOR, and ZZZ.FOR into a single object file named XXX.OBJ:

```
$ FORTRAN XXX.FOR+YYY.FOR+ZZZ.FOR
```

Source files XXX.FOR, YYY.FOR, and ZZZ.FOR are concatenated and compiled as one file, producing an object file named XXX.OBJ. The default level of optimization is used, allowing interprocedure optimizations since the files are compiled together (see Section 5.1.2). In interactive mode, no listing file is created; in batch mode, a listing file named XXX.LIS would be created.

The following FORTRAN command compiles the VSI Fortran free-form source files AAA.F90, BBB.F90, and CCC.F90 into two object files:

```
$ FORTRAN AAA+BBB,CCC/LIST
```

Two object files are produced: AAA.OBJ (comprising AAA.F90 and BBB.F90) and CCC.OBJ (comprising CCC.F90). One listing file is produced: CCC.LIS (comprising CCC.F90), since the positional /LIST qualifier follows a specific file name (and not the FORTRAN command). Because the default level of optimization is used, interprocedure optimizations for both AAA.F90 and BBB.F90 occur. Only interprocedure optimizations within CCC.F90 occur.

The following FORTRAN command compiles the VSI Fortran free-form source files ABC.F90, CIRC.F90, and XYZ.F90, but no object file is produced:

```
$ FORTRAN ABC+CIRC/NOOBJECT+XYZ
```

The /NOOBJECT qualifier applies to *all* files in the concatenated file list and suppresses object file creation. Source files ABC.F90, CIRC.F90, and XYZ.F90 are concatenated and compiled, but no object file is produced.

2.2.6.5. Requesting Additional Compile-Time and Run-Time Checking

The following FORTRAN command compiles the VSI Fortran free-form source file TEST.F90, requesting all possible compile-time diagnostic messages (/WARN=ALL) and run-time checking (/CHECK=ALL):

```
$ FORTRAN /WARNINGS=ALL /CHECK=ALL TEST.F90
```

This command creates the object file TEST.OBJ and can result in more informational or warning compilation messages than the default level of warning messages. Additional run-time messages can also occur.

2.2.6.6. Checking Fortran 90 or 95 Standard Conformance

The following FORTRAN command compiles the VSI Fortran free-form source file PROJ_STAND.F90, requesting compile-time diagnostic messages when extensions to the Fortran 90 language are detected without creating an object file:

```
$ FORTRAN /STANDARD=F90 /NOOBJECT PROJ_STAND.F90
```

This command does not create an object file but issues additional compilation messages about the use of any nonstandard extensions to the Fortran 90 standard it detects.

To check for Fortran 95 standard conformance, specify /STANDARD=F95 instead of /STANDARD=F90.

2.2.6.7. Requesting Additional Optimizations

The following FORTRAN command compiles the VSI Fortran free-form source files (M_APP.F90 and SUB.F90) as a single concatenated file into one object file:

```
$ FORTRAN /OPTIMIZE=(LEVEL=5,UNROLL=3) M_APP.F90+SUB.F90/NOLIST
```

The source files are compiled together, producing an object file named M_APP.OBJ. The software pipelining optimization (/OPTIMIZE=LEVEL=4) is requested. Loops within the program are unrolled three times (UNROLL=3). Loop unrolling occurs at optimization level three (LEVEL=3) or above.

For More Information:

- On linking object modules into an executable image (program) and running the program, see Chapter 3.
- On requesting debugger symbol table information, see Section 4.2.1.
- On requesting additional directories to be searched for module files or included source files, see Section 2.3.25 and Section 2.4.
- On the FORTRAN command qualifiers, see Section 2.3.

2.3. FORTRAN Command Qualifiers

FORTRAN command qualifiers influence the way in which the compiler processes a file. In some cases, the simple FORTRAN command is sufficient. Use optional qualifiers when needed.

You can override some qualifiers specified on the command line by using the OPTIONS statement. The qualifiers specified by the OPTIONS statement affect only the program unit where the statement occurs. For more information about the OPTIONS statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.1. FORTRAN Command Qualifier Syntax

Normal DCL parsing rules apply to the FORTRAN command, allowing you to abbreviate qualifier names and keywords as long as they are unique. For example, the following use of the FORTRAN command to specify the /ALIGNMENT qualifier as NATURAL and the /LIST qualifier is valid:

```
$ FORTRAN /ALIG=NAT /LIS EXTRAPOLATE.FOR
```

When creating command procedures, avoid using abbreviations for qualifiers and their keywords. VSI Fortran might add new qualifiers and keywords for a subsequent release that makes an abbreviated qualifier and keyword nonunique in a subsequent release.

If you specify multiple keywords for a qualifier, enclose the keywords in parentheses:

```
$ FORTRAN /WARN=(ALIGN, DECLARATION) EXTRAPOLATE.FOR
```

To concatenate source files, separate file names with a plus sign (+) instead of a comma (,). (See Section 2.2.2).

2.3.2. Summary of FORTRAN Command Qualifiers

Table 2.1 lists the FORTRAN Command qualifiers, including their default values.

Sections 2.3.3 through 2.3.51 describe each qualifier in detail.

Table 2.1. FORTRAN Command Qualifiers

Qualifiers and Defaults
/ALIGNMENT or /NOALIGNMENT or **/ALIGNMENT=** { *rule* | *class* = *rule* | (*class* = *rule* [, ...]) | [NO]SEQUENCE | ALL (or NATURAL) | NONE (or PACKED) } where: ***class* =** { COMMONS | RECORDS | STRUCTURES } and ***rule* =** { NATURAL | PACKED | STANDARD[1] | [NO]MULTILANGUAGE[1] } **Default:** /ALIGNMENT=(COMMONS=(PACKED [2], NOMULTILANGUAGE), NOSEQUENCE[2], RECORDS=NATURAL)
/ANALYSIS_DATA[= *filename*] or /NOANALYSIS_DATA **Default:** /NOANALYSIS_DATA
/ANNOTATIONS or /NOANNOTATIONS or **/ANNOTATIONS=** { { CODE | DETAIL | INLINING | LOOP_TRANSFORMS | LOOP_UNROLLING | PREFETCHING | SHRINKWRAPPING | SOFTWARE_PIPELINING | TAIL_CALLS | TAIL_RECURSION } [, ...] | { ALL | NONE } } **Default:** /NOANNOTATIONS
/ARCHITECTURE= { GENERIC | HOST | EV4 | EV5 | EV56 | PCA56 | EV6 | EV67 } **Default:** /ARCHITECTURE=GENERIC[2]
/NOASSUME or **/ASSUME=** { { [NO]ACCURACY_SENSITIVE | [NO]ALTPARAM | [NO]BUFFERED_IO | [NO]BYTERECL | [NO]DUMMY_ALIASES | [NO]FP_CONSTANT | [NO]INT_CONSTANT | [NO]MINUS0 | [NO]PROTECT_CONSTANTS | [NO]SOURCE_INCLUDE } [, ...] | { ALL | NONE } } **Default:** /ASSUME= (ACCURACY_SENSITIVE[2], ALTPARAM, NOBUFFERED_IO, NOBYTERECL, NODUMMY_ALIASES, NOFP_CONSTANT, NOINT_CONSTANT, NOMINUS0, PROTECT_CONSTANTS, NOSOURCE_INCLUDE)
/AUTOMATIC or /NOAUTOMATIC **Default:** /NOAUTOMATIC
/BY_REF_CALL=(`routine-list`) No default
/CCDEFAULT or **/CCDEFAULT=** { { FORTRAN | LIST | NONE | DEFAULT } } **Default:** /CCDEFAULT=DEFAULT

Qualifiers and Defaults
/NOCHECK or **/CHECK=** { { [NO]ARG_INFO (I64 only) | [NO]ARG_TEMP_CREATED | [NO]BOUNDS | [NO]FORMAT | [NO]FP_EXCEPTIONS | [NO]FP_MODE (I64 only) | [NO]OUTPUT_CONVERSION | [NO]OVERFLOW | [NO]POWER | [NO]UNDERFLOW } [, ...] | { ALL | NONE } } **Default:** /CHECK= (NOBOUNDS, FORMAT[3], NOFP_EXCEPTIONS, OUTPUT_CONVERSION[3], NOOVERFLOW, POWER, NOUNDERFLOW)
/CONVERT= { BIG_ENDIAN | CRAY | FDX | FGX | IBM | LITTLE_ENDIAN | NATIVE | VAXD | VAXG } **Default:** /CONVERT=NATIVE
/D_LINES or /NOD_LINES **Default:** /NOD_LINES
/DEBUG or /NODEBUG or **/DEBUG=** { { [NO]SYMBOLS | [NO]TRACEBACK } [, ...] | { ALL | NONE } } **Default:** /DEBUG= (NOSYMBOLS, TRACEBACK)
/DIAGNOSTICS[= *filename*] or /NODIAGNOSTICS **Default:** /NODIAGNOSTICS
/DML Default is omitted
/DOUBLE_SIZE= { 64 | 128 } **Default:** /DOUBLE_SIZE=64
/ERROR_LIMIT= n or /NOERROR_LIMIT **Default:** /ERROR_LIMIT=30
/EXTEND_SOURCE or /NOEXTEND_SOURCE **Default:** /NOEXTEND_SOURCE
/F77 or /NOF77 **Default:** /F77
/FAST No default
/FLOAT= { D_FLOAT | G_FLOAT | IEEE_FLOAT } **Defaults:** /FLOAT=IEEE (I64) /FLOAT=G_FLOAT (Alpha)[4]

Qualifiers and Defaults
/GRANULARITY= { BYTE | LONGWORD | QUADWORD } **Default:** /GRANULARITY=QUADWORD
/IEEE_MODE= { FAST | UNDERFLOW_TO_ZERO | DENORM_RESULTS } **Default:** /IEEE_MODE=DENORM_RESULTS (I64) **Default:** /IEEE_MODE=FAST (Alpha)
/INCLUDE= `directory` [, ...] or /NOINCLUDE **Default:** /NOINCLUDE
/INTEGER_SIZE= { 16 | 32 | 64 } **Default:** INTEGER_SIZE=32[4]
/LIBRARY No default
/LIST[= *file-spec*] or /NOLIST **Defaults:** /NOLIST (interactive) /LIST (batch)
/MACHINE_CODE or /NOMACHINE_CODE /NOMACHINE_CODE
/MATH_LIBRARY= { ACCURATE | FAST } *(Alpha only)* **Default:** /MATH_LIBRARY=ACCURATE[2]
/MODULE= *directory* or /NOMODULE **Default:** /NOMODULE
/NAMES= { UPPERCASE | LOWERCASE | AS_IS } **Default:** /NAMES=UPPERCASE
/OBJECT[= *file-spec*] or /NOOBJECT **Default:** /OBJECT
/OLD_F77 (Alpha only) Omitted
/OPTIMIZE or /NOOPTIMIZE or **/OPTIMIZE=** { LEVEL= n | INLINE= { NONE | MANUAL | SIZE | SPEED | ALL } | NOINLINE | LOOPS | PIPELINE | TUNE= { GENERIC | HOST | EV4 | EV5 | EV56 | PCA56 | EV6 | EV67 } | UNROLL= n }**[, ...]**

Qualifiers and Defaults
Default: /OPTIMIZE (same as /OPTIMIZE= (LEVEL=4, INLINE=SPEED, NOLOOPS, NOPIPELINE, TUNE=GENERIC[2], UNROLL=0))
/PAD_SOURCE or /NOPAD_SOURCE **Default:** /NOPAD_SOURCE
/REAL_SIZE= { 32 | 64 | 128 } **Default:** /REAL_SIZE=32
/RECURSIVE or /NORECURSIVE **Default:** /NORECURSIVE
/NOREENTRANCY or **/REENTRANCY=** { ASYNC | NONE | THREADED } **Default:** /REENTRANCY=NONE
/ROUNDING_MODE= { NEAREST | CHOPPED | DYNAMIC | MINUS_INFINITY } **Default:** /ROUNDING_MODE=NEAREST
/SEPARATE_COMPILATION or /NOSEPARATE_COMPILATION **Default:** /NOSEPARATE_COMPILATION
/SEVERITY=WARNINGS= { WARNING | ERROR | STDERROR } **Default:** /SEVERITY=WARNINGS=(WARNING)
/SHOW or /NOSHOW or **/SHOW=** { { [NO]DICTIONARY | [NO]INCLUDE | [NO]MAP | [NO]PREPROCESSOR } [, ...] | { ALL | NONE } } **Default:** /SHOW= (NODICTIONARY, NOINCLUDE, MAP, NOPREPROCESSOR)
/SOURCE_FORM= { FREE | FIXED } **Default:** Depends on file type (F90 for free form and FOR or F for fixed form)
/STANDARD= { F90 | F95 } or /NOSTANDARD **Default:** /NOSTANDARD
/SYNCHRONOUS_EXCEPTIONS or /NOSYNCHRONOUS_EXCEPTIONS (Alpha only) **Default:** /NOSYNCHRONOUS_EXCEPTIONS
/SYNTAX_ONLY or /NOSYNTAX_ONLY **Default:** /NOSYNTAX_ONLY
/TIE or /NOTIE **Default:** /NOTIE
/VERSION

Qualifiers and Defaults
Default is omitted
/VMS or /NOVMS **Default:** /VMS
/WARNINGS or /NOWARNINGS or **/WARNINGS=** { { [NO]ALIGNMENT | [NO]ARGUMENT_CHECKING | [NO]DECLARATIONS | [NO]GENERAL | [NO]GRANULARITY | [NO]IGNORE_LOC | [NO]TRUNCATED_SOURCE | [NO]UNCALLED | [NO]UNINITIALIZED | [NO]UNUSED | [NO]USAGE } [, ...] | { ALL | NONE } } **Default:** /WARNINGS= (ALIGNMENT, NOARGUMENT_CHECKING, NODECLARATIONS, GENERAL, GRANULARITY, NOIGNORE_LOC, NOTRUNCATED_SOURCE, UNCALLED, UNINITIALIZED, NOUNUSED, USAGE)

[1]STANDARD and MULTILANGUAGE are valid for `class`=COMMON (not `class`=RECORDS).

[2]The default changes if you specify /FAST.

[3]The default changes if you specify /NOVMS.

[4]Use the /FLOAT qualifier instead of the /[NO]G_FLOAT qualifier; use the /INTEGER_SIZE qualifier instead of the /[NO]I4 qualifier.

Table 2.2 shows the functional groupings of the FORTRAN command qualifiers and the section in which they are described in more detail.

Table 2.2. FORTRAN Command Flags and Categories

Category	Flag Name and Section in this Manual
Carriage control for terminal or printer displays	/CCDEFAULT (see Section 2.3.10)
Code generation for specific Alpha chip	/ARCHITECTURE (Alpha only) (see Section 2.3.6)
Convert Nonnative Data Files	/CONVERT (see Section 2.3.12) /ASSUME=BYTERECL (see Section 2.3.7) Also see Chapter 9.
Data Size	/DOUBLE_SIZE (see Section 2.3.17) /INTEGER_SIZE (see Section 2.3.26) /REAL_SIZE (see Section 2.3.37) Also see Chapter 8.
Data Storage and Recursion	/AUTOMATIC (see Section 2.3.8) /RECURSIVE (see Section 2.3.38)
Debugging and Symbol Table	/D_LINES (see Section 2.3.13) /DEBUG (see Section 2.3.14) Also see Chapter 4.

Category	Flag Name and Section in this Manual
Floating-Point Exceptions and Accuracy	/ASSUME=([NO]ACCURACY_SENSITIVE, [NO]FP_CONSTANT, [NO]MINUS0,) (see Section 2.3.7) /CHECK=([NO]FP_EXCEPTIONS, [NO]POWER, NO]UNNO]UNDERFLOW) (see Section 2.3.11) /DOUBLE_SIZE (see Section 2.3.17) /FAST (see Section 2.3.21) /FLOAT (see Section 2.3.22) /IEEE_MODE (see Section 2.3.24) /MATH_LIBRARY (Alpha only) (see Section 2.3.30) /REAL_SIZE (see Section 2.3.37) /ROUNDING_MODE (see Section 2.3.40) /SYNCHRONOUS_EXCEPTIONS (Alpha only) (see Section 2.3.46)
Floating-Point Format in Memory	/FLOAT (see Section 2.3.22)
Language Compatibility	/DML (see Section 2.3.16) /F77 (see Section 2.3.20) /ALIGNMENT=COMMONS=STANDARD (see Section 2.3.3) /ASSUME=([NO]ALTPARAM, [NO]DUMMY_ALIASES, [NO]INT_CONSTANT, /[NO]PROTECT_CONSTANTS) (see Section 2.3.7) /BY_REF_CALL (see Section 2.3.9) /OLD_F77 (Alpha only) (see Section 2.3.34) /PAD_SOURCE (see Section 2.3.44) /SEVERITY (see Section 2.3.42) /STANDARD (see Section 2.3.45) /VMS (see Section 2.3.50)
Language-Sensitive Editor Use	/DIAGNOSTICS (see Section 2.3.15)
Listing File and Contents	/ANNOTATIONS (see Section 2.3.5) /LIST (see Section 2.3.28) /MACHINE_CODE (see Section 2.3.29) /SHOW (see Section 2.3.43)

Category	Flag Name and Section in this Manual
	Also see Section 2.7.
Module File Searching and Placement; Include File Searching	/INCLUDE (see Section 2.3.25) /ASSUME=[NO]SOURCE_INCLUDE (see Section 2.3.7) /LIBRARY (see Section 2.3.27) /MODULE (see Section 2.3.31) Also see Section 2.2.3 (modules), Section 2.2.4 (include files), and Section 2.4 (text libraries).
Multithreaded Applications	/REENTRANCY (see Section 2.3.39)
Object File Creation, Contents, Naming, and External Names	/OBJECT (see Section 2.3.33) /NAMES (see Section 2.3.32) /SEPARATE_COMPILATION (see Section 2.3.41) /SYNTAX_ONLY (see Section 2.3.47) /TIE (see Section 2.3.49)
Performance Optimizations and Alignment	/ALIGNMENT (see Section 2.3.3) /ASSUME=(NOACCURACY_SENSITIVE, NODUMMY_ALIASES) (see Section 2.3.7) /ARCHITECTURE (Alpha only) (see Section 2.3.6) /FAST (see Section 2.3.21) /MATH_LIBRARY (Alpha only) (see Section 2.3.30) /OPTIMIZE (see Section 2.3.35) Also see Chapter 5.
Recursion and Shared Access to Data	/GRANULARITY (see Section 2.3.23) /RECURSIVE (see Section 2.3.38) /REENTRANCY (see Section 2.3.39) /WARNINGS=GRANULARITY (see Section 2.3.51)
Source Code Analyzer Use	/ANALYSIS_DATA (see Section 2.3.4)
Source Form and Column Use	/D_LINES (see Section 2.3.13) /EXTEND_SOURCE (see Section 2.3.19) /PAD_SOURCE (see Section 2.3.36) /SOURCE_FORM (see Section 2.3.44) /WARNINGS=TRUNCATED_SOURCE (see Section 2.3.51)

Category	Flag Name and Section in this Manual
Warning Messages and Run-Time Checking	/CHECK (see Section 2.3.11) /ERROR_LIMIT (see Section 2.3.18) /IEEE_MODE (see Section 2.3.24) /SEVERITY (see Section 2.3.42) /WARNINGS (see Section 2.3.51) Also see the qualifiers related to “Floating-Point Exceptions and Accuracy” in this table.

2.3.3. /ALIGNMENT — Data Alignment

The /ALIGNMENT qualifier controls the data alignment of numeric fields in common blocks and structures.

If you omit the /ALIGNMENT and /FAST qualifiers:

- 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 77 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 VSI Fortran always aligns local data items on natural boundaries, certain data declaration statements and unaligned arguments can force unaligned data.

Use the /ALIGNMENT qualifier to control the alignment of fields associated with common blocks, derived-type structures, and record structures.

The compiler issues messages when it detects unaligned data by default (/WARNINGS=ALIGNMENT). For information about the causes of unaligned data and detection at run time, see Section 5.3.

Note

Unaligned data significantly increases the time it takes to execute a program, depending on the number of unaligned fields encountered. Specifying /ALIGNMENT=ALL (same as /ALIGNMENT=NATURAL) minimizes unaligned data.

The qualifier has the following form:

/ALIGNMENT= { *rule* | *class* = *rule* | (*class* = *rule* [, ...]) | [NO]SEQUENCE | ALL (or NATURAL) | NONE (or PACKED) }

where:

class= { COMMONS | RECORDS | STRUCTURES }

and

rule= { NATURAL | PACKED | STANDARD | [NO]MULTILANGUAGE }

STANDARD and MULTILANGUAGE are valid for `class`=COMMON (not `class`=RECORDS).

The /ALIGNMENT qualifier keywords specify whether the VSI Fortran compiler should naturally align or arbitrarily pack the following:

class

Specifies the type of data blocks:

- COMMONS= *rule* applies to common blocks (COMMON statement); *rule* can be PACKED, STANDARD, NATURAL, or MULTILANGUAGE.
- RECORDS= *rule* applies to derived-type and record structures; *rule* can be PACKED or NATURAL. However, if a derived-type data definition specifies the SEQUENCE statement, the FORTRAN /ALIGNMENT qualifier has no effect on unaligned data, so data declaration order must be carefully planned to naturally align data.
- STRUCTURES= *rule* applies to derived-type and record structures; *rule* can be PACKED or NATURAL. For the VSI Fortran language, STRUCTURES and RECORDS are the same (they may have a different meaning in other OpenVMS languages).

rule

Specifies the alignment for the specified *class* of data blocks:

- NATURAL requests that fields in derived-type and record structures and data items in common blocks be naturally aligned on up to *8-byte* boundaries, including INTEGER (KIND=8) and REAL (KIND=8) data.

  Specifying /ALIGNMENT=NATURAL is equivalent to any of the following:

  ```
  /ALIGNMENT
  /ALIGNMENT=ALL
  /ALIGNMENT=(COMMONS=(NATURAL,NOMULTILANGUAGE),RECORDS=NATURAL,SEQUENCE)
  ```

- PACKED requests that fields in derived-type and record structures and data items in common blocks be packed on arbitrary byte boundaries and not naturally aligned.

  Specifying /ALIGNMENT=PACKED is equivalent to any of the following:

  ```
  /ALIGNMENT=NONE
  /NOALIGNMENT
  /ALIGNMENT=(COMMONS=(PACKED,NOMULTILANGUAGE),RECORDS=PACKED,NOSEQUENCE)
  ```

- STANDARD specifies that data items in common blocks will be naturally aligned on up to 4-byte boundaries (consistent with the FORTRAN-77, Fortran 90, and Fortran 95 standards).

  The compiler will not naturally align INTEGER (KIND=8) and REAL (KIND=8) data declarations. Such data declarations should be planned so they fall on natural boundaries. Specifying /ALIGNMENT=/ALIGNMENT=COMMONS=STANDARD alone is the same as /ALIGNMENT=(COMMONS=(STANDARD,NOMULTILANGUAGE),RECORDS=NATURAL).

You cannot specify /ALIGNMENT=RECORDS=STANDARD or /ALIGNMENT=STANDARD.

- MULTILANGUAGE specifies that the compiler pad the size of common block program sections to ensure compatibility when the common block program section is shared by code created by other OpenVMS compilers.

  When a program section generated by a Fortran common block is overlaid with a program section consisting of a C structure, linker error messages can result. This is because the sizes of the program sections are inconsistent; the C structure is padded and the Fortran common block is not.

  Specifying /ALIGNMENT=COMMONS=MULTILANGUAGE ensures that VSI Fortran follows a consistent program section size allocation scheme that works with VSI C program sections that are shared across multiple images. Program sections shared in a single image do not have a problem. The equivalent VSI C qualifier is /PSECT_MODEL=[NO]MULTILANGUAGE.

  The default is /ALIGNMENT=COMMONS=NOMULTILANGUAGE, which also is the default behavior of Compaq Fortran 77 and is sufficient for most applications.

  The [NO]MULTILANGUAGE keyword only applies to common blocks. You can specify /ALIGNMENT=COMMONS=[NO]MULTILANGUAGE, but you cannot specify /ALIGNMENT=[NO]MULTILANGUAGE.

[NO]SEQUENCE

Specifying /ALIGNMENT=SEQUENCE means that components of derived types with the SEQUENCE attribute will obey whatever alignment rules are currently in use. The default alignment rules align components on natural boundaries.

The default value of /ALIGNMENT=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 /ALIGNMENT=SEQUENCE so that components of derived types with the SEQUENCE attribute will be naturally aligned for improved performance. Specifying /ALIGNMENT=ALL also sets /ALIGNMENT=SEQUENCE.

ALL

Specifying /ALIGNMENT=ALL is equivalent to /ALIGNMENT, /ALIGNMENT=NATURAL), or /ALIGNMENT=(COMMONS= (NATURAL,NOMULTILANGUAGE),RECORDS=NATURAL,SEQUENCE).

NONE

Specifying /ALIGNMENT=NONE is equivalent to /NOALIGNMENT, /ALIGNMENT=PACKED, or /ALIGNMENT=(COMMONS=(PACKED, NOMULTILANGUAGE), RECORDS=PACKED,NOSEQUENCE).

Defaults depend on whether you specify or omit the /ALIGNMENT and /FAST qualifiers, as follows:

Command Line	Default
Omit /ALIGNMENT and omit /FAST	/ALIGNMENT=(COMMONS=(PACKED, NOMULTILANGUAGE),NOSEQUENCE, RECORDS=NATURAL)

Command Line	Default
Omit /ALIGNMENT and specify /FAST	/ALIGNMENT=(COMMONS=NATURAL, RECORDS=NATURAL,SEQUENCE)
/ALIGNMENT=COMMONS= *rule*	Use whatever the /ALIGNMENT qualifier specifiers for COMMONS, but use the default of RECORDS=NATURAL
/ALIGNMENT=RECORDS= *rule*	Use whatever the /ALIGNMENT qualifier specifies for RECORDS, but use the default for COMMONS (depends on whether /FAST was specified or omitted)

You can override the alignment specified on the command line by using a cDEC$ OPTIONS directive, as described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

The /ALIGNMENT and /WARNINGS=ALIGNMENT qualifiers can be used together in the same command line.

For More Information:

- On run-time performance guidelines, see Chapter 5.
- On data alignment, see Section 5.3.
- On checking for alignment traps with a condition handler, see Section 14.12.
- On intrinsic data sizes, see Chapter 8.
- On the /FAST qualifier, see Section 2.3.21.

2.3.4. /ANALYSIS_DATA – Create Analysis Data File

The /ANALYSIS_DATA qualifier produces an analysis data file that contains cross-reference and static-analysis information about the source code being compiled.

Analysis data files are reserved for use by products such as, but not limited to, the Source Code Analyzer (SCA).

The qualifier has the following form:

```
/ANALYSIS_DATA[=filename.type]
```

If you omit the file specification, the analysis file name has the name of the primary source file and a file type of ANA (*filename*.ANA).

The compiler produces one analysis file for each source file that it compiles. If you are compiling multiple files and you specify a particular name as the name of the analysis file, each analysis file is given that name (with an incremental version number).

If you do not specify the /ANALYSIS_DATA qualifier, the default is /NOANALYSIS_DATA.

2.3.5. /ANNOTATIONS — Code Descriptions

The /ANNOTATIONS qualifier controls whether an annotated listing showing optimizations is included with the listing file.

The qualifier has the following form:

/ANNOTATIONS= { { CODE | DETAIL | INLINING | LOOP_TRANSFORMS | LOOP_UNROLLING | PREFETCHING | SHRINKWRAPPING | SOFTWARE_PIPELINING | TAIL_CALLS | TAIL_RECURSION } [, ...] | { ALL | NONE } }

CODE

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

DETAIL

Provides additional level of annotation detail, where available.

INLINING

Indicates where code for a called procedure was expanded inline.

LOOP_TRANSFORMS

Indicates where advanced loop nest optimizations have been applied to improve cache performance.

LOOP_UNROLLING

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

PREFETCHING

Indicates where special instructions were used to reduce memory latency.

SHRINKWRAPPING

Indicates removal of code establishing routine context when it is not needed.

SOFTWARE_PIPELINING

Indicates where instructions have been rearranged to make optimal use of the processor's functional units.

TAIL_CALLS

Indicates an optimization where a call can be replaced with a jump.

TAIL_RECURSION

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

ALL

All annotations, including DETAIL, are selected. This is the default if no keyword is specified.

NONE

No annotations are selected. This is the same as /NOANNOTATIONS.

2.3.6. /ARCHITECTURE — Architecture Code Instructions (Alpha only)

The /ARCHITECTURE qualifier specifies the type of Alpha architecture code instructions generated for a particular program unit being compiled; it uses the same options (keywords) as used by the /OPTIMIZE=TUNE (Alpha only) qualifier (for instruction scheduling purposes).

OpenVMS Version 7.1 and subsequent releases provide 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 runtime.

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

The qualifier has the following form:

/ARCHITECTURE= { GENERIC | HOST | EV4 | EV5 | EV56 | PCA56 | EV6 | EV67 }

GENERIC

Generates code that is appropriate for all Alpha processor generations. This is the default.

Programs compiled with the GENERIC option run on all implementations of the Alpha architecture without any instruction emulation overhead.

HOST

Generates code for the processor generation in use on the system being used for compilation.

Programs compiled with this option may encounter instruction emulation overhead if run on other implementations of the Alpha architecture.

EV4

Generates code for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture. Programs compiled with the EV4 option run on all Alpha processors without instruction emulation overhead.

EV5

Generates code for some 21164 chip implementations of the Alpha architecture that use only the base set of Alpha instructions (no extensions). Programs compiled with the EV5 option run on all Alpha processors without instruction emulation overhead.

EV56

Generates code for some 21164 chip implementations that use the BWX (Byte/Word manipulation) instruction extensions of the Alpha architecture.

Programs compiled with the EV56 option may incur emulation overhead on EV4 and EV5 processors but still run correctly on OpenVMS Version 7.1 (or later) systems.

PCA56

Generates code for the 21164PC chip implementation that uses the BWX (Byte/Word manipulation) and MAX (Multimedia) instruction extensions of the Alpha architecture.

Programs compiled with the PCA56 option may incur emulation overhead on EV4, EV5, and EV56 processors but will still run correctly on OpenVMS Version 7.1 (or later) systems.

EV6

Generates code for the 21264 chip implementation that uses the following extensions to the base Alpha instruction set: BWX (Byte/Word manipulation) and MAX (Multimedia) instructions, square root and floating-point convert instructions, and count instructions.

Programs compiled with the EV6 option may incur emulation overhead on EV4, EV5, EV56, and PCA56 processors but will still run correctly on OpenVMS Version 7.1 (or later) systems.

EV67

Generates code for chip implementations that use advanced instruction extensions of the Alpha architecture. This option permits the compiler to generate any EV67 instruction, including the following extensions to the base Alpha instruction set: BWX (Byte/Word manipulation), MVI (Multimedia) instructions, square root and floating-point convert extensions (FIX), and count extensions (CIX).

Programs compiled with the EV67 keyword might incur emulation overhead on EV4, EV5, EV56, PCA56, and EV6 processors but still run correctly on OpenVMS Alpha systems.

2.3.7. /ASSUME — Compiler Assumptions

The /ASSUME qualifier specifies a variety of assumptions:

- Whether the compiler should use certain code transformations that affect floating-point operations. These changes may affect the accuracy of the program's results.
- What the compiler can assume about program behavior without affecting correct results when it optimizes code.
- Whether a single-precision constant assigned to a double-precision variable should be evaluated in single or double precision.
- Changes the directory where the compiler searches for files specified by an INCLUDE statement to the directory where the source files reside, not the current default directory.

The qualifier has the following form:

/ASSUME= { { [NO]ACCURACY_SENSITIVE | [NO]ALTPARAM | [NO]BUFFERED_IO | [NO]BYTERECL | [NO]DUMMY_ALIASES | [NO]FP_CONSTANT | [NO]INT_CONSTANT | [NO]MINUS0 | [NO]PROTECT_CONSTANTS | [NO]SOURCE_INCLUDE } [, ...] | { ALL | NONE } }

[NO]ACCURACY_SENSITIVE

If you use ACCURACY_SENSITIVE (the default unless you specified /FAST), the compiler uses a limited number of rules for calculations, which might prevent some optimizations.

Specifying 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 (ACCURACY_SENSITIVE) because of the way intermediate results are rounded. Specifying the /FAST qualifier (described in Section 2.3.21) changes the default to NOACCURACY_SENSITIVE.

Numeric results with NOACCURACY_SENSITIVE are not categorically less accurate. They can produce more accurate results for certain floating-point calculations, such as dot product summations.

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)
```

Optimizations that result in calls to a special reciprocal square root routine for expressions of the form 1.0/SQRT(x) or A/SQRT(B) are enabled only if /ASSUME=NOACCURACY_SENSITIVE is in effect.

[NO]ALTPARAM

Specifying the /ASSUME=ALTPARAM qualifier allows the alternate syntax for PARAMETER statements. The alternate 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 /ASSUME=ALTPARAM.

[NO]BUFFERED_IO

The /ASSUME=[NO]BUFFERED_IO qualifier is provided for compatibility with other platforms. On OpenVMS systems, the SET RMS command controls the number of output buffers

Specifying /ASSUME=BUFFERED_IO requests that buffered I/O be used for all Fortran logical units opened for sequential writing (the default is NOBUFFERED_IO). On OpenVMS systems, this qualifier has an effect only if the system or process RMS default buffers are set to 1.

[NO]BYTERECL

Specifying the /ASSUME=BYTERECL qualifier:

- Indicates that the OPEN statement RECL value for unformatted files is in byte units. If you omit /ASSUME=BYTERECL, VSI 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, VSI 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, VSI Fortran returns the RECL value for an INQUIRE in longword (4-byte) units.

To specify /ASSUME=BYTERECL your application must be running on a system that has the associated Fortran Run-Time Library support. This support is provided by installing one of the following on the system where the application will be run:

- VSI Fortran Version 7.0 or later

- OpenVMS Version 7.0 or later

If you specify /ASSUME=BYTERECL and your application is running on a system without the proper Run-Time Library support, it will fail with an INVARGFOR, Invalid argument to Fortran Run-Time Library error.

[NO]DUMMY_ALIASES

Specifies whether dummy (formal) arguments are permitted to share memory locations with COMMON block variables or other dummy arguments.

If you specify DUMMY_ALIASES, 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 90 and Fortran 95 Standards and slow performance.

If you use NODUMMY_ALIASES, the default, 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 DUMMY_ALIASES.

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

[NO]FP_CONSTANT

Specifies whether a single-precision constant assigned to a double-precision variable will be evaluated in double precision.

If you use NOFP_CONSTANT, the default, a single-precision constant assigned to a double-precision variable is evaluated in single precision. The Fortran 90 and 95 standards require that the constant be evaluated in single precision.

If you specify FP_CONSTANT, a single-precision constant assigned to a double-precision variable is evaluated in double precision.

Certain programs created for FORTRAN-77 compilers (including Compaq Fortran 77) may show different results with FP_CONSTANT than when you use NOFP_CONSTANT, 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 FP_CONSTANT, identical values are assigned to D1 and D2. If you use NOFP_CONSTANT, VSI Fortran will follow 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
```

[NO]INT_CONSTANT

Specifies whether or Fortran 90/95 semantics are used to determine the type for integer constants.

If you specify /ASSUME=INT_CONSTANT, integer constants take their type from the value, as interpreted by Compaq Fortran 77.

If you specify /ASSUME=NOINT_CONSTANT, integer constants have Fortran 90/95 “default integer” type. This is the default.

[NO]MINUS0

Controls whether the compiler uses Fortran 95 standard semantics for the IEEE floating-point value of -0.0 (minus zero) in the SIGN intrinsic, if the processor is capable of distinguishing the difference between -0.0 and +0.0.

The default is /ASSUME=NOMINUS0, which uses Fortran 90 and FORTRAN 77 semantics where the value -0.0 or +0.0 in the SIGN function is treated as an 0.0.

To request Fortran 95 semantics, specify /ASSUME=MINUS0 to allow use of the IEEE value -0.0 in the SIGN intrinsic and printing -0.0. This option applies only to programs compiled with the /FLOAT=IEEE_FLOAT qualifier.

[NO]PROTECT_CONSTANTS

Specifies whether constant actual arguments can be changed. By default, actual arguments that are constants are read-only (/ASSUME=PROTECT_CONSTANTS): any attempt to modify them in the called routine results in an error. /ASSUME=NOPROTECT_CONSTANTS specifies that a copy of a constant actual argument is to be passed, so it can be modified by the called routine, even though the Fortran standard prohibits such modification. The constant is not modified in the calling routine.

If NOPROTECT_CONSTANTS is specified, a writeable copy of the constant is passed as the actual argument.

[NO]SOURCE_INCLUDE

Controls whether the directory where the compiler searches for source files or text libraries specified by an INCLUDE statement is either:

- The current default directory (NOSOURCE_INCLUDE)
- The directory where the source file reside (SOURCE_INCLUDE)

The default, NOSOURCE_INCLUDE, indicates that the compiler should search in the current default directory.

Specifying SOURCE_INCLUDE causes the compiler to search the directory of the source file specified on the FORTRAN command line, instead of the current default directory.

You can specify additional directories for the compiler to search for module files, include files, or include libraries by using the /INCLUDE qualifier.

For More Information:

- On the effect of /ASSUME=NOACCURACY_SENSITIVE on optimizations, see Section 5.8.8.
- On converting and using nonnative unformatted data files, see Chapter 9.
- On the INQUIRE statement, see Chapter 6.
- On the effect and an example of /ASSUME=DUMMY_ALIASES, see Section 5.8.9.

- On the /INCLUDE qualifier, see Section 2.3.25.

2.3.8. /AUTOMATIC — Data Storage

The /AUTOMATIC and /NOAUTOMATIC qualifiers are synonyms of /RECURSIVE or /NORECURSIVE (see Section 2.3.38). This qualifier is provided for Compaq Fortran 77 compatibility.

2.3.9. /BY_REF_CALL — Character Literal Argument Passing

Specifying /BY_REF_CALL= `routine-list` indicates that character constants used as actual arguments in calls to the specified routines be passed by reference and not by character descriptor. This helps applications that pass quoted character constants to numeric dummy parameters, such as applications ported from OpenVMS VAX systems that rely on the OpenVMS VAX Linker to change the argument passing mechanism for character constant actual arguments.

By default, VSI Fortran passes character constants used as actual arguments by the usual OpenVMS character descriptor mechanism.

You can specify a list of routines or use wildcard characters (such as an asterisk (*)) to indicate all routines, for example:

```
$ FORTRAN/BY_REF_CALL=(PROJA_ROUT,PROJB_ROUT) TEST.FOR
$ FORTRAN/BY_REF_CALL=(*) APPLIC.FOR
```

The first FORTRAN command requests special character constant actual argument handling for two specific routines. The second requests this for all routines. You can select routines whose names match certain characters and a wildcard, such as all routines that start with MY:

```
$ FORTRAN/BY_REF_CALL=(MY*) X.FOR
```

2.3.10. /CCDEFAULT — Carriage Control for Terminals

The /CCDEFAULT qualifiers specify default carriage control when a terminal or printer displays a file.

The qualifier has the following form:

/CCDEFAULT= { { FORTRAN | LIST | NONE | DEFAULT } }

FORTRAN

Specifying /CCDEFAULT=FORTRAN results in normal Fortran interpretation of the first character, such as the character “0” resulting in a blank line before output.

LIST

Specifying /CCDEFAULT=LIST results in one linefeed between records.

NONE

Specifying /CCDEFAULT=NONE results in no carriage control processing.

DEFAULT

Specifying /CCDEFAULT=DEFAULT results in the possibility of other qualifiers, such as /VMS, affecting this default setting: if /NOVMS is specified, the default is LIST; otherwise, the default is FORTRAN.

2.3.11. /CHECK — Generate Code for Run-Time Checking

The /CHECK qualifier requests certain error checking during program execution (run time). The compiler produces extra code that performs the checks.

The qualifier has the following form:

/CHECK= { { [NO]ARG_INFO (I64 only) | [NO]ARG_TEMP_CREATED | [NO]BOUNDS | [NO]FORMAT | [NO]FP_EXCEPTIONS | [NO]FP_MODE (I64 only) | [NO]OUTPUT_CONVERSION | [NO]OVERFLOW | [NO]POWER | [NO]UNDERFLOW } [, ...] | { ALL | NONE } }

[NO]ARG_INFO (I64 only)

Controls whether run-time checking of the actual argument list occurs. For actual arguments that correspond to declared formal parameters, the check compares the run-time argument type information for arguments passed in registers with the type that is expected. An informational message is issued at run time for each miscompare. Extra actual arguments or too few actual arguments are not reported.

With the default, NOARG_INFO, no check is made.

ARG_TEMP_CREATED

Controls whether a run-time warning message is displayed (execution continues) if a temporary is created for an array actual argument passed to a called routine.

[NO]BOUNDS

Controls whether run-time checking occurs for each dimension of an array reference or substring reference to determine whether it is within the range of the dimension specified by the array or character variable declaration.

Specify BOUNDS to request array bounds and substring checking. With the default, NOBOUNDS, array bounds and substring checking does not occur.

[NO]FORMAT

Controls whether the run-time message number 61 (FORVARMIS) is displayed and halts program execution when the data type for an item being formatted for output does not match the format descriptor being used (such as a REAL data item with an I format).

The default, FORMAT, causes FORVARMIS to be a fatal error and halts program execution.

Specifying NOFORMAT ignores the format mismatch, which suppresses the FORVARMIS error and allows program continuation.

If you omit /NOVMS and omit /CHECK=NOFORMAT, the default is /CHECK=FORMAT.

If you specify /NOVMS, the default is NOFORMAT (unless you also specify /CHECK=FORMAT).

[NO]FP_EXCEPTIONS

Controls whether run-time checking counts calculations that result in exceptional values. With the default, NOFP_EXCEPTIONS, no run-time messages are reported.

Specifying FP_EXCEPTIONS requests reporting of the first two occurrences of each type of exceptional value and a summary run-time message at program completion that displays the number of times exceptional values occurred. Consider using FP_EXCEPTIONS when the /IEEE_MODE qualifier allows generation of exceptional values.

To limit reporting to only denormalized numbers (and not other exceptional numbers), specify UNDERFLOW instead of FP_EXCEPTIONS.

Using FP_EXCEPTIONS applies to all types of native floating-point data.

[NO]FP_MODE (I64 only)

Controls whether run-time checking of the current state of the processor's floating-point status register (FPSR) occurs. For every call of every function or subroutine, the check will compare the current state of the FPSR register against the expected state. That state is based on the /FLOAT, /IEEE_MODE and /ROUND qualifier values specified by the FORTRAN command. An informational message is issued at run time for miscompares.

With the default, NOFP_MODE, no check is made.

[NO]OUTPUT_CONVERSION

Controls whether run-time message number 63 (OUTCONERR) is displayed when format truncation occurs. Specifying /CHECK=NOOUTPUT_CONVERSION disables the run-time message (number 63) associated with format truncation. The data item is printed with asterisks. When OUTPUT_CONVERSION is in effect and a number could not be output in the specified format field length without loss of significant digits (format truncation), the OUTCONERR (number 63) error occurs.

If you omit /NOVMS and omit /CHECK=NOOUTPUT_CONVERSION, the default is OUTPUT_CONVERSION.

If you specify /NOVMS, the default is NOOUTPUT_CONVERSION (unless you also specify /CHECK=OUTPUT_CONVERSION).

[NO]OVERFLOW

Controls whether run-time checking occurs for arithmetic overflow of all integer calculations (INTEGER, INTEGER with a kind parameter, or INTEGER with a length specifier). Specify OVERFLOW to request integer overflow checking.

Real and complex calculations are always checked for overflow and are not affected by /NOCHECK. Integer exponentiation is performed by a routine in the mathematical library. The routine in the mathematical library always checks for overflow, even if NOOVERFLOW is specified.

With the default, NOOVERFLOW, overflow checking does not occur.

[NO]POWER

Specifying the /CHECK=NOPOWER qualifier allows certain arithmetic expressions containing floating-point numbers and exponentiation to be evaluated and return a result rather than cause 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.

If you omit /CHECK=NOPOWER for such expressions, an exception occurs, error message number 65 is displayed, and the program stops (default is /CHECK=POWER).

[NO]UNDERFLOW

Controls whether run-time messages are displayed for floating underflow (denormalized numbers) in floating-point calculations. Specifying UNDERFLOW might be used in combination with the /IEEE_MODE=DENORM_RESULTS qualifier. Specify UNDERFLOW to request reporting of the first two occurrences of denormalized numbers and a summary run-time message at program completion that displays the number of times denormalized numbers occurred.

The default, NOUNDERFLOW, means that floating underflow messages are not displayed. To check for all exceptional values (not just denormalized numbers), specify /CHECK=FP_EXCEPTIONS.

ALL

Requests that all run-time checks (BOUNDS, FORMAT, FP_EXCEPTIONS, OUTPUT_CON2VERSION, OVERFLOW, and UNDERFLOW) be performed. Specifying /CHECK and /CHECK=ALL are equivalent.

NONE

Requests no run-time checking. This is the default. Specifying /NOCHECK and /CHECK=NONE are equivalent.

For More Information:

- On exceptional floating-point values, see Section 8.4.8.
- On controlling IEEE arithmetic exception handling (/IEEE_MODE qualifier), see Section 2.3.24.
- On the ranges of the various data types (including denormalized ranges), see Chapter 8.

2.3.12. /CONVERT — Unformatted Numeric Data Conversion

The /CONVERT qualifier specifies the format of numeric unformatted data in a file, such as IEEE little endian, VAX G_float, VAX D_float floating-point format, or a nonnative big endian format.

By default, an unformatted file containing numeric data is expected to be in the same floating-point format used for memory representation or /CONVERT=NATIVE. You set the floating-point format used for memory representation using the /FLOAT qualifier (see Section 2.3.22).

Instead of specifying the unformatted file format by using the /CONVERT qualifier, you can use one of the other methods (predefined logical names or the OPEN CONVERT keyword) described in Chapter 9, which allow the same program to use different floating-point formats, as shown in Figure 2.1.

The qualifier has the following form (specify one keyword):

/CONVERT= { BIG_ENDIAN | CRAY | FDX | FGX | IBM | LITTLE_ENDIAN | NATIVE | VAXD | VAXG }

BIG_ENDIAN

Specifies that unformatted files containing numeric data are in IEEE big endian (nonnative) format.

If you specify BIG_ENDIAN, the resulting program will read and write unformatted files containing numeric data assuming:

- Big endian integer format (INTEGER declarations of the appropriate size)
- Big endian IEEE floating-point formats (REAL and COMPLEX declarations of the appropriate size).

CRAY

Specifies that unformatted files containing numeric data are in CRAY (nonnative) big endian format.

If you specify CRAY, the resulting program will read and write unformatted files containing numeric data assuming:

- Big endian integer format (INTEGER declarations of the appropriate size)
- Big endian CRAY proprietary floating-point formats (REAL and COMPLEX declarations of the appropriate size)

FDX

Specifies that unformatted files containing numeric data are in I64/Alpha-compatible D_float-centered little endian format, as follows:

- REAL (KIND=4) and COMPLEX (KIND=4) (same as REAL*4 and COMPLEX*8) single-precision data is in VAX little endian F_float format.
- REAL (KIND=8) and COMPLEX (KIND=8) (same as REAL*8 and COMPLEX*16) double-precision data is in VAX little endian D_float format.
- REAL (KIND=16) (same as REAL*16) data is in IEEE-style little endian X_float format.

FGX

Specifies that unformatted files containing numeric data are in I64/Alpha-compatible G_float-centered little endian format, as follows:

- REAL (KIND=4) and COMPLEX (KIND=4) (same as REAL*4 and COMPLEX*8) single-precision data is in VAX little endian F_float format.
- REAL (KIND=8) and COMPLEX (KIND=8) (same as REAL*8 and COMPLEX*16) double-precision data is in VAX little endian G_float format.
- REAL (KIND=16) (same as REAL*16) data is in IEEE-style little endian X_float format.

IBM

Specifies that unformatted files containing numeric data are in IBM ® (nonnative) big endian format (such as IBM System \370 and similar systems).

If you specify IBM, the resulting program will read and write unformatted files containing numeric data assuming:

- Big endian integer format (INTEGER declarations of the appropriate size)
- Big endian IBM proprietary floating-point formats (REAL and COMPLEX declarations of the appropriate size)

LITTLE_ENDIAN

Specifies that unformatted files containing numeric data are in native little endian integer format and IEEE little endian floating-point format, as follows:

- Integer data is in native little endian format.
- REAL (KIND=4) and COMPLEX (KIND=4) (same as REAL*4 and COMPLEX*8) single-precision data is in IEEE little endian S_float format.
- REAL (KIND=8) and COMPLEX (KIND=8) (same as DOUBLE PRECISION and DOUBLE COMPLEX) double-precision data is in IEEE little endian T_float format.
- REAL (KIND=16) data is in IEEE-style little endian X_float format.

NATIVE

Specifies that the format for unformatted files containing numeric data is not converted. When using NATIVE (the default), the numeric format in the unformatted files must match the floating-point format representation in memory, which is specified using the /FLOAT qualifier.

This is the default.

VAXD

Specifies that unformatted files containing numeric data are in VAX-compatible D_float-centered little endian format, as follows:

- Integer data is in native little endian format.
- REAL (KIND=4) and COMPLEX (KIND=4) (same as REAL*4 and COMPLEX*8) single-precision data is in VAX F_float floating-point format.
- REAL (KIND=8) and COMPLEX (KIND=8) (same as REAL*8 and COMPLEX*16) double-precision data is in VAX D_float little endian format.
- REAL (KIND=16) (same as REAL*16) data is in VAX H_float little endian format.

VAXG

Specifies that unformatted files containing numeric data are in VAX-compatible G_float-centered little endian format, as follows:

- Integer data is in native little endian format.
- REAL (KIND=4) and COMPLEX (KIND=4) (same as REAL*4 and COMPLEX*8) single-precision data is in VAX F_float floating-point format.
- REAL (KIND=8) and COMPLEX (KIND=8) (same as REAL*8 and COMPLEX*16) double-precision data is in VAX G_float little endian format.

- REAL (KIND=16) (same as REAL*16) data is in VAX H_float little endian format.

For More Information:

- On limitations of data conversion, see Section 9.4.
- On converting unformatted data files, including using the OPEN statement CONVERT specifier and using FOR$CONVERT `nnn` logical names, see Section 9.5.
- On the ranges and formats of the various native intrinsic floating-point data types, see Section 8.4.
- On porting Compaq Fortran 77 data from OpenVMS VAX systems, see Section B.7.
- On the ranges of the nonnative VAX H_float data type, see Section B.8.

2.3.13. /D_LINES — Debugging Statement Indicator, Column 1

Specify /D_LINES to request that the compiler should treat lines in fixed-form source files that contain a D in column 1 as source code rather than comment lines. Such lines might print the values of variables or otherwise provide useful debugging information. This qualifier is ignored for free-form source files.

The default is /NOD_LINES, which means that lines with a D in column 1 are treated as comments.

2.3.14. /DEBUG — Object File Traceback and Symbol Table

The /DEBUG qualifier requests that the object module contain information for use by the OpenVMS Debugger and the run-time error traceback mechanism.

The qualifier has the following form:

/DEBUG= { { [NO]SYMBOLS | [NO]TRACEBACK } [, ...] | { ALL | NONE } }

[NO]SYMBOLS

Controls whether the compiler provides the debugger with local symbol definitions for user-defined variables, arrays (including dimension information), structures, parameter constants, and labels of executable statements.

[NO]TRACEBACK

Controls whether the compiler provides an address correlation table so that the debugger and the run-time error traceback mechanism can translate virtual addresses into source program routine names and compiler-generated line numbers.

ALL

Requests that the compiler provide both local symbol definitions and an address correlation table. If you specify /DEBUG without any keywords, it is the same as /DEBUG=ALL or /DEBUG=(TRACEBACK,SYMBOLS). When you specify /DEBUG, also specify /NOOPTIMIZE to prevent optimizations that complicate debugging.

NONE

Requests that the compiler provide no debugging information. The /NODEBUG, /DEBUG=NONE, and /DEBUG=(NOTRACEBACK, NOSYMBOLS) qualifiers are equivalent.

If you omit /DEBUG, the default is /DEBUG=(TRACEBACK, NOSYMBOLS).

Note

The use of /NOOPTIMIZE is strongly recommended when the /DEBUG qualifier is used. Optimizations performed by the compiler can cause several different kinds of unexpected behavior when using the OpenVMS Debugger.

For More Information:

- On using the OpenVMS Debugger, see Chapter 4.
- On debugging qualifiers for the FORTRAN and LINK commands, see Section 4.2.1.
- On LINK command qualifiers related to traceback and debugging, see Section 3.2.2 and Table 3.2.

2.3.15. /DIAGNOSTICS — Create Diagnostics File

The /DIAGNOSTICS qualifier creates a file containing compiler messages and diagnostic information.

The qualifier has the following form:

```
/DIAGNOSTICS[=file-spec]
```

The default is /NODIAGNOSTICS.

If you omit the file specification, the diagnostics file has the name of your source file and a file type of DIA.

The diagnostics file is reserved for use with third-party layered products such as, but not limited to, the Language Sensitive Editor (LSE).

For More Information:

On using LSE, see the Guide to Source Code Analyzer for VMS Systems or online LSE HELP.

2.3.16. /DML — Invoke Fortran DML Preprocessor

The /DML qualifier invokes the Fortran Data Manipulation Language (DML) preprocessor before the compiler. The preprocessor produces an intermediate file of VSI Fortran source code in which Fortran DML commands are expanded into VSI Fortran statements. The compiler is then automatically invoked to compile this intermediate file.

The qualifier has the following form:

```
/DML
```

The default is not to invoke the Fortran DML preprocessor.

Use the /SHOW=PREPROCESSOR qualifier in conjunction with the /DML qualifier to cause the preprocessor-generated source code to be included in the listing file.

Any switches preceding the /DML qualifier in the command line are ignored.

Note

Unless you specify the /DEBUG qualifier, the intermediate file is deleted by the Fortran DML preprocessor immediately after compilation is complete, and the Language Sensitive Editor and the Source Code Analyzer cannot access the source program when you use the /DML qualifier. The results with the /DEBUG qualifier reflect the intermediate source.

For More Information:

On the DML preprocessor, see the *Oracle CODASYL DBMS Programming Reference Manual*.

2.3.17. /DOUBLE_SIZE — DOUBLE PRECISION Data Size

The /DOUBLE_SIZE qualifier allows you to specify the data size for floating-point DOUBLE PRECISION data declarations. The qualifier has the following form:

/DOUBLE_SIZE= { 64 | 128 }

To request that all DOUBLE PRECISION declarations, constants, functions, and intrinsics use the REAL (KIND=16) extended-precision data rather than REAL (KIND=8) double-precision data, specify /DOUBLE_SIZE=128. REAL (KIND=16) data is stored in memory using X_float format.

If you omit /DOUBLE_SIZE=128, the size of DOUBLE PRECISION declarations is REAL (KIND=8) or 64-bit double-precision data (default is /DOUBLE_SIZE=64). To select the floating-point format used in memory for 64-bit REAL (KIND=8) data, use the /FLOAT qualifier.

For More Information:

On the /FLOAT qualifier, see Section 2.3.22.

2.3.18. /ERROR_LIMIT — Limit Compiler Messages

The /ERROR_LIMIT qualifier specifies the maximum number of error-level or fatal-level compiler errors allowed for a given compilation unit (one or more files specified on the FORTRAN command line that create a single object file).

The qualifier has the following form:

```
/ERROR_LIMIT[=nn] or /NOERROR_LIMIT
```

If you specify /ERROR_LIMIT= `n`, the compilation can have up to `n` - 1 errors without terminating the compilation. When the error limit is reached, compilation is terminated.

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

By default, execution of the compiler is terminated when 30 error (E-level) and fatal (F-level) messages are detected (default is /ERROR_LIMIT=30).

When the error limit is surpassed, only compilation of the current comma-list element is terminated; the compiler will proceed to compile any other comma-list element. For example, consider the following:

```
$ FORTRAN A,B,C
```

If comma-list element A has more than 30 E- or F-level errors, its compilation is terminated, but the compiler proceeds to compile elements B and C.

A list of files separated by plus signs (+) form a single compilation unit. In the following example, compilation of the plus-sign separated files A, B, or C stops when the *total* of E- or F-level errors for *all three* files exceeds 30:

```
$ FORTRAN A+B+C
```

Specifying /ERROR_LIMIT=0 is equivalent to specifying /ERROR_LIMIT=1 (compilation terminates when the first error-level or fatal-level error occurs).

For More Information:

On compiler diagnostic messages, see Section 2.6.

2.3.19. /EXTEND_SOURCE — Line Length for Fixed-Form Source

Specify /EXTEND_SOURCE to request that the compiler increase the length of VSI Fortran statement fields to column 132 for fixed-form source files, instead of column 72 (the default). It is ignored for free-form source files.

You can also specify this qualifier by using the OPTIONS statement. The default in either case is /NOEXTEND_SOURCE.

To request warning messages for truncated fixed-form source lines, specify /WARNINGS=TRUNCATED_SOURCE.

For More Information:

- On recognized file name suffix characters and their relationship to fixed and free source formats, see Section 2.2.1.
- On column positions and more complete information on the fixed and free source formats, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On /WARNINGS=TRUNCATED_SOURCE, see Section 2.3.51.

2.3.20. /F77 — FORTRAN IV or FORTRAN-77 Compatibility

The /F77 qualifier requests that the compiler use FORTRAN-77 (and thus Fortran 90/95) interpretation rules for those statements that have different meanings in older versions of the Fortran standards. The default is /F77.

If you specify /NOF77, the compiler uses the FORTRAN 66 (FORTRAN IV) interpretation. This means, among other things, that:

- DO loops are always executed at least once FORTRAN-66 EXTERNAL statement syntax and semantics are allowed.
- If the OPEN statement STATUS specifier is omitted, the default changes to STATUS='NEW' instead of STATUS='UNKNOWN'.

- If the OPEN statement BLANK specifier is omitted, the default changes to BLANK='ZERO' instead of BLANK='NULL'.

2.3.21. /FAST — Request Fast Run-Time Performance

Specifying /FAST changes the defaults for certain qualifiers, usually improving run-time performance. The new defaults are:

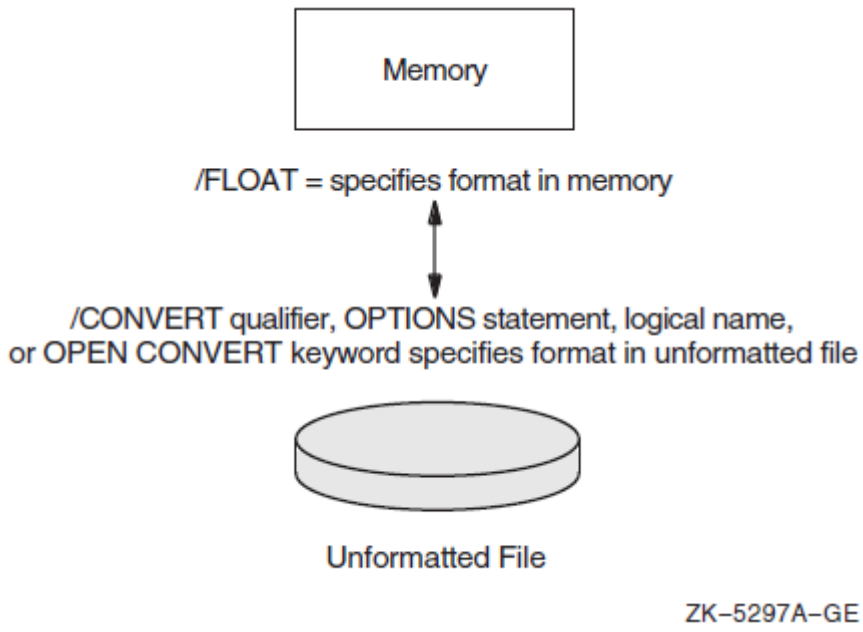
- /ALIGNMENT=(COMMONS=NATURAL,RECORDS=NATURAL,SEQUENCE) (same as /ALIGNMENT=NATURAL) (see Section 2.3.3)
- /ARCHITECTURE=HOST (see Section 2.3.6)
- /ASSUME=NOACCURACY_SENSITIVE (see Section 2.3.7)
- /MATH_LIBRARY=FAST (Alpha only) (see Section 2.3.30)
- /OPTIMIZE=TUNE=HOST (Alpha only) (see Section 2.3.35)

2.3.22. /FLOAT — Specify Floating-Point Format in Memory

The /FLOAT qualifier specifies the floating-point data format to be used in memory for REAL or COMPLEX data. For performance reasons, consider specifying the same floating-point memory format as the floating-point format used by unformatted files the program will access if the data falls within the acceptable range.

Figure 2.1 shows the FORTRAN command qualifiers used to specify the floating-point format used in memory and in an unformatted file. Because REAL (KIND=16) (same as REAL*16) data is always in X_float format on OpenVMS I64 and Alpha systems, the /FLOAT keyword specifies the format for REAL (KIND=4), COMPLEX (KIND=4), REAL (KIND=8), and COMPLEX (KIND=8) data (or equivalent declarations).

Figure 2.1. Specifying the Format of Numeric Data for Unformatted Files



To specify the floating-point format (such as big endian) for all unformatted *files* opened by the program, use the /CONVERT qualifier. To allow the same program to use different floating-point formats, you must use the predefined logical names or the OPEN CONVERT keyword to specify the format for specific unit numbers, as described in Chapter 9.

The qualifier has the following form:

/FLOAT= { D_FLOAT | G_FLOAT | IEEE_FLOAT }

Note

The OpenVMS Alpha instruction set does not support D_float computations, and the OpenVMS I64 instruction set does not support D_float, F_float or G_float computations. As a result, any data stored in those formats is converted to a native format for arithmetic computations and then converted back to its original format. On Alpha systems, the native format used for D_float is G_float. On I64 systems, S_float is used for F_float data, and T_float is used for D_float and G_float data.

This means that for programs that perform many floating-point computations, using D_float data on Alpha systems is slower than using G_float or T_float data. Similarly, using D_float, F_float, or G_float data on I64 systems is slower than using S_float or T_float data. Additionally, due to the conversions involved, the results might differ from native VAX D_float, F_float, and G_float computations and results.

You should not mix floating data type formats in routines that pass single-precision or double-precision quantities among themselves.

D_FLOAT

Specifies that the memory format for REAL (KIND=4) and COMPLEX (KIND=4) data is VAX F_float and that the memory format for REAL (KIND=8) and COMPLEX (KIND=8) data is VAX D_float. Same as the obsolete qualifier /NOG_FLOATING.

Due to the considerations noted above, we do not recommend use of the /FLOAT=D_FLOAT qualifier unless a program must use unformatted data files in D_float format. If range and accuracy constraints permit the use of the other REAL (KIND=8) data types, consider converting existing unformatted files that contain D_float data to another format, such as G_float on Alpha systems, or T_float on Alpha or I64 systems (see Chapter 9).

G_FLOAT

Specifies that the memory format for single precision REAL (KIND=4) and COMPLEX (KIND=4) data is VAX F_float and that the memory format for double precision REAL (KIND=8) and COMPLEX (KIND=8) data is VAX G_float. Same as the obsolete qualifier /G_FLOATING.

The default on Alpha systems is /FLOAT=G_FLOAT.

Due to the considerations noted above, on I64 systems we do not recommend use of the /FLOAT=G_FLOAT qualifier unless a program must use unformatted data files in G_float format. If range and accuracy constraints permit it, consider converting existing unformatted files that contain G_float data to T_float (see Chapter 9).

IEEE_FLOAT

Specifies that the memory format for single precision REAL (KIND=4) and COMPLEX (KIND=4) data is IEEE S_float and the memory format for double precision REAL (KIND=8) and COMPLEX (KIND=8) is IEEE T_float.

The default on I64 systems is /FLOAT=IEEE_FLOAT. If possible, this default should be used, because it provides the greatest performance and accuracy on I64.

Specifying /FLOAT=IEEE_FLOAT allows the use of certain IEEE exceptional values. When you specify /FLOAT=IEEE_FLOAT, you should be aware of the /CHECK=FP_EXCEPTIONS, /CHECK=FP_MODE, /IEEE_MODE, and /ROUNDING_MODE qualifiers.

Because REAL (KIND=16) (same as REAL*16) and COMPLEX (KIND=16) (same as COMPLEX*32) data is always in X_float format on I64 and Alpha systems, operations that use REAL (KIND=16) and COMPLEX (KIND=16) data may encounter certain exceptional values even when /FLOAT=IEEE_FLOAT is not used.

For More Information:

- On intrinsic floating-point data types, see Chapter 8.
- On converting unformatted files, see Section 2.3.12.
- On qualifiers of interest when you specify /FLOAT=IEEE_FLOAT, see:

  Section 2.3.11 (/CHECK=FP_EXCEPTIONS)
  Section 2.3.11 (/CHECK=FP_MODE) (I64 only)
  Section 2.3.24 (/IEEE_MODE)
  Section 2.3.40 (/ROUNDING_MODE)

2.3.23. /GRANULARITY — Control Shared Memory Access to Data

The /GRANULARITY qualifier controls the size of data 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.

The qualifier has the following form:

/GRANULARITY= { BYTE | LONGWORD | QUADWORD }

Data that can be written from multiple threads must be declared as VOLATILE (so it is not held in registers). To ensure alignment in common blocks, derived types, and record structures, use the /ALIGNMENT qualifier.

BYTE

Requests that all data (one byte or greater) can be accessed from different threads sharing data in memory. This option will slow run-time performance.

LONGWORD

Ensures that naturally aligned data of four bytes or greater can be accessed safely from different threads sharing access to that data in memory. Accessing data items of three bytes or less and unaligned data may result in data items written from multiple threads being inconsistently updated.

QUADWORD

Ensures that naturally aligned data of eight bytes can be accessed safely from different threads sharing data in memory. Accessing data items of seven bytes or less and unaligned data may result in data items written from multiple threads being inconsistently updated. This is the default.

For More Information:

- On the Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.
- On the Alpha architecture, see the *Alpha Architecture Reference Manual*.
- On intrinsic data types, see Chapter 8.

2.3.24. /IEEE_MODE — Control IEEE Arithmetic Exception Handling

The /IEEE_MODE qualifier specifies the arithmetic exception handling used for floating-point calculations, such as for exceptional values. On Alpha systems, it also controls the precision of exception reporting (like /SYNCHRONOUS_EXCEPTIONS (Alpha only)).

Exceptional values are associated with IEEE arithmetic and include Infinity (+ and -) values, Not-A-Number (NaN) values, invalid data values, and denormalized numbers (see Section 8.4.8).

Use the /IEEE_MODE qualifier to control:

- Whether exceptional values cause program termination or continuation
- Whether exception reporting is precise
- Whether underflow (denormalized) values are set to zero

This qualifier only applies to arithmetic calculations when:

- You omit the /MATH_LIBRARY=FAST (Alpha only) qualifier (or /FAST) qualifier. On Alpha systems, using /MATH_LIBRARY=FAST provides limited handling of exceptional values of arguments to and results from VSI Fortran intrinsic functions.
- You specify the /FLOAT=IEEE_FLOAT qualifier to request IEEE S_float (KIND=4) and T_float (KIND=8) data.

  If you specify /FLOAT=G_FLOAT (the default on Alpha) or /FLOAT=D_FLOAT, /IEEE_MODE must not also be specified.

The qualifier has the following form:

/IEEE_MODE= { FAST | UNDERFLOW_TO_ZERO | DENORM_RESULTS }

The default on I64 systems is /IEEE_MODE=DENORM_RESULTS.

The default on Alpha systems is /IEEE_MODE=FAST.

Note

You should choose the value for the /IEEE_MODE qualifier based on the floating-point semantics your application requires, not on possible performance benefits.

FAST

Specifies that the program should stop if any exceptional values are detected. This is the default on Alpha systems.

When the program encounters or calculates any exceptional values (infinity (+ or –), NaN, or invalid data) in a calculation, the program stops and displays a message.

Denormalized values calculated in an arithmetic expression are set to zero. Denormalized values encountered as variables in an arithmetic expression (including constant values) are treated as invalid data (an exceptional value), which stops the program.

On Alpha systems, exceptions are not reported until one or more instructions after the instruction that caused the exception. To have exceptions reported at the instruction that caused the exception when using /IEEE_MODE=FAST, also specify /SYNCHRONOUS_EXCEPTIONS (Alpha only).

UNDERFLOW_TO_ZERO

Specifies that the program should continue if any exceptional values are detected and set calculated denormalized (underflow) values to zero.

When the program encounters an exceptional value (infinity (+ or –), NaN, invalid data) in an arithmetic expression, the program continues. It also continues when the result of a calculation is an exceptional value.

Calculated denormalized values are set to zero (0). This prevents the denormalized number from being used in a subsequent calculation (propagated).

Exceptions are reported at the instruction that caused the exception (same as /SYNCHRONOUS_EXCEPTIONS (Alpha only)). This allows precise run-time reporting of exceptions for those programs that generate exceptional values, but this slows program run-time performance.

Using UNDERFLOW_TO_ZERO allows programs to handle exceptional values, but does not propagate numbers in the denormalized range.

To request run-time messages for arithmetic exceptions, specify the /CHECK=FP_EXCEPTIONS qualifier.

DENORM_RESULTS

Specifies that the program should continue if any exceptional values are detected and leave calculated denormalized values as is (allows underflow).

When the program encounters an exceptional value (infinity (+ or –), NaN, invalid data) in an arithmetic expression, the program continues. It also continues when the result of a calculation is an exceptional value.

Calculated denormalized values are left as denormalized values. When a denormalized number is used in a subsequent arithmetic expression, it requires extra software-assisted handling and slows performance. A program that generates denormalized numbers will be slower than the same program compiled using /IEEE_MODE=UNDERFLOW_TO_ZERO.

Exceptions are reported at the instruction that caused the exception (same as /SYNCHRONOUS_EXCEPTIONS (Alpha only)). This allows precise run-time reporting of

exceptions for those programs that generate exceptional values, but this slows program run-time performance.

Using DENORM_RESULTS allows programs to handle exceptional values, including allowing underflow of denormalized numbers.

To request run-time messages for arithmetic exceptions, specify the /CHECK=FP_EXCEPTIONS qualifier. To request run-time messages for only those arithmetic exceptions related to denormalized numbers, specify the /CHECK=UNDERFLOW qualifier.

For More Information:

- On exceptional floating-point values, see Section 8.4.8.
- On controlling run-time arithmetic exception messages (/CHECK=(FP_EXCEPTIONS,UNDERFLOW) qualifier), see Section 2.3.11.
- On the ranges of the various data types (including denormalized ranges), see Chapter 8.
- On IEEE floating-point exception handling, see the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985).
- On the Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.

2.3.25. /INCLUDE — Add Directory for INCLUDE and Module File Search

The /INCLUDE qualifier specifies one or more additional directories for the VSI Fortran compiler to search for:

- Module files (specified by a USE statement)

  Module files have a file type of F90$MOD and are created by the VSI Fortran compiler. For more information on module files, see Section 2.2.3.

- Include files (specified by an INCLUDE statement)

  Include files have a file type like other VSI Fortran source files (F90, FOR, or F) and are usually created by the user with a text editor.

  VSI Fortran also provides certain include library modules in the text library FORSYSDEF.TLB. Users can create a text library and then populate and maintain include library modules in the library. For more information on include files and include text library modules, see Section 2.2.4.

If the INCLUDE statement specifies an explicit device and/or directory, only that directory is searched.

If you omit /INCLUDE, the compiler searches in the current directory for module files and either the current directory or the directory containing the source file specified on the FORTRAN command line for include files or text libraries:

- If you omit /ASSUME=SOURCE_INCLUDE, only the current default directory is searched.
- If you specify /ASSUME=SOURCE_INCLUDE, the directory where the source file resides is searched instead of the current directory.

The order of directory searching follows:

1. The current directory (omit /ASSUME=SOURCE_INCLUDE) or the directory where the source file resides (specify /ASSUME=SOURCE_INCLUDE).

2. One or more directories specified by the /INCLUDE qualifier.

3. The location defined by the logical name FORT$INCLUDE (if defined). To prevent searching in this directory, specify /NOINCLUDE.

If you do not specify /INCLUDE or /NOINCLUDE, the compiler searches first in the current directory (or directory where the source file resides) and then the directory specified by FORT$INCLUDE (if defined).

To request that the compiler only search in the current directory (or directory where the source file resides), specify /NOINCLUDE. This prevents the compiler from searching the FORT$INCLUDE directory. If you use /NOINCLUDE, you cannot specify /INCLUDE.

To request that the compiler only search in the directory specified by FORT$INCLUDE, specify /INCLUDE=FORT$INCLUDE.

To control the searching for text libraries (not included files or modules), you can also use the logical name FORT$LIBRARY.

Like other OpenVMS logical names, it can specify the location for your process only or for multiple processes (including system-wide).

To specify the additional directories DISKA:[PROJ_MODULE.F90] and DISKB:[F_COMMON.F90] with the /INCLUDE qualifier, use a single /INCLUDE qualifier, as follows:

```
$ FORTRAN PROJ_M.F90 /INCLUDE=(DISKA:[PROJ_MODULE.F90],DISKB:
[F_COMMON.F90])
```

If you specify multiple directories, the order of the directories (and their devices) in the /INCLUDE qualifier determines the directory search order.

For More Information:

- On the /ASSUME=SOURCE_INCLUDE qualifier, see Section 2.3.7.
- On using library modules in a text library, see Section 2.2.4 and Section 2.3.27.
- On OpenVMS logical names, see the *OpenVMS User's Manual*.

2.3.26. /INTEGER_SIZE — Integer and Logical Data Size

The /INTEGER_SIZE qualifier controls how the compiler interprets INTEGER or LOGICAL declarations that do not have a specified length. The default is INTEGER_SIZE=32.

The qualifier has the following form:

/INTEGER_SIZE= { 16 | 32 | 64 }

16

Indicates that INTEGER declarations are interpreted as INTEGER (KIND=2) and LOGICAL declarations as LOGICAL (KIND=2). Same as the obsolete /NOI4 qualifier.

32

Indicates that INTEGER declarations are interpreted as INTEGER (KIND=4) and LOGICAL declarations as LOGICAL (KIND=4). Same as the obsolete /I4 qualifier.

64

Indicates that INTEGER declarations are interpreted as INTEGER (KIND=8) and LOGICAL declarations as LOGICAL (KIND=8)

For performance reasons, use INTEGER (KIND=4) data instead of INTEGER (KIND=2) or INTEGER (KIND=1) and whenever possible. You must explicitly declare INTEGER (KIND=1) data.

Note

To improve performance, use /INTEGER_SIZE=32 rather than /INTEGER_SIZE=16 and declare variables as INTEGER (KIND=4) (or INTEGER (KIND=8)) rather than INTEGER (KIND=2) or INTEGER (KIND=1). For logical data, avoid using /INTEGER_SIZE=16 and declare logical variables as LOGICAL (KIND=4) rather than LOGICAL (KIND=2) or LOGICAL (KIND=1).

For More Information:

- On intrinsic data types and their ranges, see Chapter 8.
- On run-time integer overflow checking (/CHECK=OVERFLOW), see Section 2.3.11.

2.3.27. /LIBRARY — Specify File as Text Library

The /LIBRARY qualifier specifies that a file is a text library file.

The qualifier has the following form:

text-library-file/LIBRARY

The /LIBRARY qualifier can be specified on one or more text library files in a list of files concatenated by plus signs (+). At least one of the files in the list must be a nonlibrary file. The default file type is TLB.

For More Information:

- On the use of text libraries, see Section 2.4.
- On the OpenVMS Librarian, see the *OpenVMS Librarian Utility Manual* or enter HELP LIBRARY.

2.3.28. /LIST — Request Listing File

The /LIST qualifier requests a source listing file. You can request additional listing information using the /MACHINE_CODE and /SHOW qualifiers.

The qualifier has the following form:

/LIST[=*file-spec*]

You can include a file specification for the listing file. If you omit the file specification, the listing file has the name of the first source file and a file type of LIS.

The default depth of a page in a listing file is 66 lines. To modify the default, assign the new number to the logical name SYS$LP_LINES, using the DCL command DEFINE.For example, the following DCL command sets the page depth at 88 lines:

```
$  DEFINE SYS$LP_LINES 88
```

The valid number of lines per page ranges from 30 to a maximum of 255. The definition can be applied to the entire system by using the command DEFINE/SYSTEM.

In interactive mode, the compiler does not produce a listing file unless you include the /LIST qualifier. In batch mode, the compiler produces a listing file by default. In either case, the listing file is not automatically printed; you must use the PRINT command to obtain a line printer copy of the listing file.

If a source line 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 VSI Fortran source files are replaced by a space character, and a warning message appears.

You can request additional information in the listing file using the /MACHINE_CODE and /SHOW qualifiers.

The listing file includes the VSI Fortran version number.

The /ANNOTATIONS qualifier controls whether an annotated listing showing optimizations is included with the listing file.

For More Information:

- On the format of listing files, see Section 2.7.1.
- On the /MACHINE_CODE qualifier, see Section 2.3.29.
- On the /SHOW qualifier, see Section 2.3.43.
- On the /ANNOTATIONS qualifier, see Section 2.3.5

2.3.29. /MACHINE_CODE — Request Machine Code in Listing File

Specifying /MACHINE_CODE requests that the listing file include a symbolic representation of the OpenVMS object code generated by the compiler. Generated code and data are represented in a form similar to an assembly code listing. The code produced by the /MACHINE_CODE qualifier is for informational purposes only. It is not intended to be assembled and is not supported by the MACRO assembler.

If a listing file is not being generated, the /MACHINE_CODE qualifier is ignored.

The default is /NOMACHINE_CODE.

For More Information:

- On the format of a machine code listing, see Section 2.7.2.
- On the /LIST qualifier, see Section 2.3.28.

2.3.30. /MATH_LIBRARY — Fast or Accurate Math Library Routines (Alpha only)

If you omit /MATH_LIBRARY=FAST (and /FAST), the compiler uses the standard, very accurate math library routines for each VSI Fortran intrinsic function, such as SQRT (default is /MATH_LIBRARY=ACCURATE).

Specify /MATH_LIBRARY=FAST to use a special version of certain math library routines that produce faster results, but with a slight loss of precision and less exception checking.

This qualifier applies only to IEEE data types (when you specify /FLOAT=IEEE_FLOAT). The qualifier has the following form:

/MATH_LIBRARY= { ACCURATE | FAST }

ACCURATE

On Alpha systems, using /MATH_LIBRARY=ACCURATE (the default if you omit /FAST) produces the very accurate results and error checking expected of quality compiler products. It uses the standard set of math library routines for the applicable intrinsics.

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 /IEEE_MODE qualifier.

FAST

Specifying /MATH_LIBRARY=FAST (the default if you specify /FAST) use versions of certain math library routines that perform faster computations than the standard, more accurate math library routines, but with slightly less fractional accuracy and less reliable arithmetic exception handling. 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 /IEEE_MODE qualifier is 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 (see Section 8.4) 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 OpenVMS operating system in use. Allowed error bounds vary with each routine.

For More Information:

- On controlling arithmetic exception handling, including using the /IEEE_MODE qualifier, see Section 2.3.24.
- On the /FAST qualifier, see Section 2.3.21.
- On requesting double-precision data during compilation for REAL data declarations (/REAL_SIZE=64 qualifier, see Section 2.3.37.
- On native floating-point formats, see Section 8.4.
- On the specific intrinsic routines that have special fast math routines, see the online release notes.

2.3.31. /MODULE — Placement of Module Files

The /MODULE qualifier controls where module files (.F90$MOD) are placed. If you omit this qualifier or specify /NOMODULE, the .F90$MOD files are placed in your current default directory.

The qualifier has the following form

`/MODULE=`*`directory`*

If you specify this qualifier, .F90$MOD files are placed in the specified directory location.

2.3.32. /NAMES — Control Case of External Names

The /NAMES qualifier specifies how the VSI Fortran compiler represents external (global) names to the linker.

The qualifier has the following form:

/NAMES= { UPPERCASE | LOWERCASE | AS_IS }

UPPERCASE

Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase. This is the default.

LOWERCASE

Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase.

AS_IS

Causes the compiler to distinguish case differences in identifiers and to preserve the case of external names.

The default, /NAMES=UPPERCASE, means that VSI Fortran converts external names to uppercase.

For More Information:

- On the EXTERNAL statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

- On specifying case names with the C property, see Section 10.4.2.1.
- On specifying names with the ALIAS property, see Section 10.4.2.2.

2.3.33. /OBJECT — Specify Name or Prevent Object File Creation

The /OBJECT qualifier specifies the name of the object file or prevents object file creation.

The qualifier has the following form:

`/OBJECT[=`*`file-spec`*`]`

The default is /OBJECT. If you omit the file specification, the object file has the name of the first source file and a file type of OBJ.

Use /NOOBJECT to suppress object code (for example, when you want to test only for compilation errors in the source program).

For More Information:

On using the /OBJECT qualifier, see Section 2.2.5.

2.3.34. /OLD_F77 — Use Old FORTRAN 77 Compiler (Alpha only)

To use the Compaq Fortran 77 compiler, specify /OLD_F77 as the first qualifier on the FORTRAN command line. The default is to use the VSI Fortran (90/95 language) compiler. The default VSI Fortran compiler supports the FORTRAN 77 language as well as the Fortran 90 and Fortran 95 standards.

If you specify the /OLD_F77 qualifier, certain FORTRAN command qualifiers will be ignored, including qualifiers associated with Fortran 90 and Fortran 95 features, Fortran 90 and 95 standards checking, and certain optimization keywords.

2.3.35. /OPTIMIZE — Specify Compiler Optimizations

The /OPTIMIZE qualifier requests that the compiler produce optimized code.

The qualifier has the following form:

`/OPTIMIZE=` { LEVEL= n | INLINE= { NONE | MANUAL | SIZE | SPEED | ALL } | NOINLINE | LOOPS | PIPELINE | TUNE= { GENERIC | HOST | EV4 | EV5 | EV56 | PCA56 | EV6 | EV67 } | UNROLL= n }**`[, …]`**

The default is /OPTIMIZE, which is equivalent to /OPTIMIZE=LEVEL=4. Use /NOOPTIMIZE or /OPTIMIZE=LEVEL=0 for a debugging session to ensure that the debugger has sufficient information to locate errors in the source program.

In most cases, using /OPTIMIZE will make the program execute faster. As a side effect of getting the fastest execution speeds, using /OPTIMIZE can produce larger object modules and longer compile times than /NOOPTIMIZE.

To allow full interprocedure optimization when compiling multiple source files, consider separating source file specifications with plus signs (+), so the files are concatenated and compiled as one program. Full interprocedure optimization can reduce overall program execution time (see Section 5.1.2). Consider not concatenating source files when the size of the source files is excessively large and the amount of memory or disk space is limited.

The /OPTIMIZE keywords follow:

LEVEL

You can specify the optimization level with /OPTIMIZE=LEVEL= `n`, where `n` is from 0 to 5, as follows:

- LEVEL=0 disables nearly all optimizations. Specifying LEVEL=0 causes the /WARNINGS=UNUSED qualifier to be ignored.

  This provides the same inlining as /OPTIMIZE=INLINE=NONE.

- LEVEL=1 enables local optimizations within the source program unit, recognition of common subexpressions, and integer multiplication and division expansion (using shifts).

- LEVEL=2 enables global optimizations and optimizations performed with LEVEL=1. Global optimizations include data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling.

  This provides the same inlining as /OPTIMIZE=INLINE=MANUAL.

- LEVEL=3 enables additional global optimizations that improve speed (at the cost of extra code size) and optimizations performed with LEVEL=2. Additional global optimizations include:

  - Loop unrolling

  - Code replication to eliminate branches

- LEVEL=4 enables interprocedure analysis, automatic inlining of small procedures (with heuristics limiting the amount of extra code), the software pipelining optimization (also set by /OPTIMIZE=PIPELINE), and optimizations performed with LEVEL=3. LEVEL=4 is the default.

  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 also enables the prefetching of data to reduce the impact of cache misses.

  For more information on software pipelining, see the PIPELINE keyword in this section.

  This provides the same inlining as /OPTIMIZE=INLINE=SPEED.

- LEVEL=5 activates the loop transformation optimizations (also set by /OPTIMIZE=LOOPS).

  - The loop transformation optimizations are a group of optimizations that apply to array references within loops. These optimizations can improve the performance of the memory system and can apply to multiple nested loops.

    Loop transformation optimizations include loop blocking, loop distribution, loop fusion, loop interchange, loop scalar replacement, and outer loop unrolling.

To specify loop transformation optimizations without software pipelining, do one of the following:

- Specify LEVEL=5 with NOPIPELINE (preferred method)
- Specify LOOPS with LEVEL=4, LEVEL=3, or LEVEL=2. This optimization is not performed at optimization levels below LEVEL=2.

For more information on the loop transformation optimizations, see the LOOPS keyword in this section.

In addition to loop transformation, specifying LEVEL=5 activates certain optimizations that are not activated by LOOPS and PIPELINE, including byte-vectorization, and insertion of additional NOP (No Operation) instructions for alignment of multi-issue sequences.

To determine whether using LEVEL=5 benefits your particular program, you should time program execution for the same program compiled at LEVEL=4 and LEVEL=5 (see Section 5.2).

For programs that contain loops that exhaust available registers, longer execution times may result with /OPTIMIZE=LEVEL=5, requiring use of /OPTIMIZE=UNROLL= `n` to limit loop unrolling (see the UNROLL keyword in this section).

Specifying LEVEL=5 implies the optimizations performed at LEVEL=1, LEVEL=2, LEVEL=3, and LEVEL=4.

This provides the same inlining as /OPTIMIZE=INLINE=SPEED.

INLINE

You can specify the level of inlining with /OPTIMIZE=INLINE= `xxxxx`, where `xxxx` is one of the following keywords:

- NONE (same as /OPTIMIZE=NOINLINE) prevents any procedures from being inlined, except statement functions, which are always inlined. This type of inlining occurs if you specify /OPTIMIZE=LEVEL=0, LEVEL=1, LEVEL=2, or LEVEL=3 and omit /OPTIMIZE=INLINE= `keyword`.
- MANUAL is the same as NONE for VSI Fortran (but not necessarily for other OpenVMS languages). This type of inlining occurs if you specify /OPTIMIZE=LEVEL=0, LEVEL=1, LEVEL=2, or LEVEL=3 and omit /OPTIMIZE=INLINE= `keyword`.
- SIZE inlines procedures that will improve run-time performance without significantly increasing program size. This type of inlining is relevant at /OPTIMIZE=LEVEL=1 or higher.
- SPEED inlines procedures that will improve run-time performance with a significant increase in program size. This type of inlining is relevant at /OPTIMIZE=LEVEL=1 or higher. INLINE=SPEED occurs if you specify /OPTIMIZE=LEVEL=4 or LEVEL=5 and omit /INLINE= `keyword`.
- ALL inlines every call that can possibly be inlined while generating correct code, including the following:
  - Statement functions (NONE or MANUAL)
  - Any procedures that VSI Fortran thinks will improve run-time performance (SPEED)

- Any other procedures that can possibly be inlined and generate correct code. Certain recursive routines are not inlined to prevent infinite loops.

  This type of inlining is relevant at /OPTIMIZE=LEVEL=1 or higher.

NOINLINE

Same as INLINE=NONE

LOOPS

Specifying /OPTIMIZE=LOOPS (or /OPTIMIZE=LEVEL=5) 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 multiple nested loops. The loops chosen for loop transformation optimizations are always counted loops (which include DO or IF loops, but not uncounted DO WHILE loops).

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 /OPTIMIZE=LOOPS include the following:

- Loop blocking
- Loop distribution
- Loop fusion
- Loop interchange
- Loop scalar replacement
- Outer loop unrolling

The loop transformation optimizations are a subset of optimizations activated by /OPTIMIZE=LEVEL=5. Instead of specifying both LOOPS and PIPELINE, you can specify /OPTIMIZE=LEVEL=5.

To specify loop transformation optimizations without software pipelining, do one of the following:

- Specify LEVEL=5 with NOPIPELINE (preferred method)
- Specify LOOPS with LEVEL=3 or LEVEL=2. This optimization is not performed at optimization levels below LEVEL=2.

To determine whether using /OPTIMIZE=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 /OPTIMIZE=LOOPS and /OPTIMIZE=NOLOOPS).

PIPELINE

Specifying /OPTIMIZE=PIPELINE (or /OPTIMIZE=LEVEL=4) 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.

For this version of VSI Fortran, loops chosen for software pipelining are always innermost loops and do not contain branches, procedure calls, or COMPLEX floating-point data.

On Alpha systems, software pipelining can be more effective when you combine /OPTIMIZE=PIPELINE with the appropriate /OPTIMIZE=TUNE= `xxxx` keyword for the target Alpha processor generation (see the TUNE keyword in this section).

Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

Software pipelining is a subset of the optimizations activated by /OPTIMIZE=LEVEL=4.

To specify software pipelining without loop transformation optimizations, do one of the following:

- Specify LEVEL=4 with NOLOOPS (preferred method)
- Specify PIPELINE with LEVEL=3 or LEVEL=2. This optimization is not performed at optimization levels below LEVEL=2.

To determine whether using /OPTIMIZE=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 /OPTIMIZE=PIPELINE and /OPTIMIZE=NOPIPELINE).

For programs that contain loops that exhaust available registers, longer execution times may result with /OPTIMIZE=LEVEL=5, requiring use of /OPTIMIZE=UNROLL= `n` to limit loop unrolling (see the UNROLL keyword in this section).

TUNE (Alpha only)

You can specify the types of processor-specific instruction tuning for implementations of the Alpha architecture using the /OPTIMIZE=TUNE= `xxxx` keywords. Regardless of the setting of /OPTIMIZE=TUNE= `xxxx` 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 target may run slower on another target.

The /OPTIMIZE=TUNE= `xxxx` keywords are as follows:

- GENERIC generates and schedules code that will execute well for all generations of Alpha processor chips. This provides generally efficient code for those cases where both processor generations are likely to be used. If /FAST is specified, the default is HOST; otherwise, the default is GENERIC.
- HOST generates and schedules code optimized for the processor generation in use on the system being used for compilation.
- EV4 generates and schedules code optimized for the 21064, 21064A, 21066, and 21068 implementations of the Alpha chip.
- EV5 generates and schedules code optimized for the 21164 implementation of the Alpha chip. This processor generation is faster and more recent than the implementations of the Alpha chip associated with EV4 (21064, 21064A, 21066, and 21068).
- EV56 generates and schedules code optimized for some 21164 Alpha architecture implementations that use the BWX (Byte/Word manipulation) instruction extensions of the Alpha architecture.
- PCA56 generates and schedules code optimized for 21164PC Alpha architecture implementation that uses BWX (Byte/Word manipulation) and MAX (Multimedia) instructions extensions.

- EV6 generates and schedules code optimized for the 21264 chip implementation that uses the following extensions to the base Alpha instruction set: BWX (Byte/Word manipulation) and MAX (Multimedia) instructions, square root and floating-point convert instructions, and count instructions.
- EV67 generates and schedules code optimized for the EV67 Alpha architecture implementation that uses BWX (Byte/Word manipulation), MVI (Multimedia) instructions, square root and floating-point convert extensions (FIX), and count extensions (CIX).

If /OPTIMIZE=TUNE= xxxx specifies a processor of less functionality than is specified by /ARCHITECTURE, code is optimized for the processor specified by /ARCHITECTURE.

If you omit /OPTIMIZE=TUNE= xxxx, HOST is used if /FAST is specified; otherwise, GENERIC is used.

UNROLL

You can specify the number of times loops are unrolled with /OPTIMIZE=UNROLL= n, where n is a number from 0 to 16. If you omit the UNROLL keyword or specify UNROLL=0 (the default), the optimizer determines how many times loops are unrolled. Usually loops are unrolled four times, but code analysis may result in certain loops being unrolled two times (twice).

VSI Fortran unrolls loops at /OPTIMIZE=LEVEL=3 or higher. When timings using LEVEL=5 show that performance has not improved, consider specifying UNROLL=1 with LEVEL=5, such as the following:

```
$ FORTRAN /OPTIMIZE=(LEVEL=5,UNROLL=1) M_APP.F90+SUB.F90/NOLIST
```

For More Information:

- On the effects of /OPTIMIZE=LEVEL=5, see Section 5.8.2.
- On limiting loop unrolling, see Section 5.7.4.1.
- On timing program execution, see Section 5.2.
- On the related /ASSUME=NOACCURACY_SENSITIVE qualifier, see Section 2.3.7.
- On guidelines for improving performance, see Chapter 5.
- On the optimizations performed at each level, see Section 5.7.

2.3.36. /PAD_SOURCE — Pad Source Lines with Spaces

Controls how the compiler treats fixed-form file source lines that are shorter than the statement field width (72 characters, or 132 characters if /EXTEND_SOURCE is in effect.) This determines how the compiler treats character and Hollerith constants that are continued across two or more source lines. This qualifier does not apply to free-form source files.

Specifying /PAD_SOURCE causes the compiler to treat short source lines as if they were padded with blanks out to the statement field width. This may be useful when compiling programs developed for non-VSI compilers that assume that short source lines are blank-padded.

The default, /NOPAD_SOURCE, is compatible with current and previous VSI Fortran compilers, which causes the compiler to not treat short source lines as padded with blanks so that the first character of a continuation line immediately follows the last character of the previous line.

If /NOPAD_SOURCE is in effect, the compiler issues an informational message if it detects a continued constant that might be affected by blank padding.

For More Information:

On /WARNINGS=USAGE qualifier, see Section 2.3.51.

2.3.37. /REAL_SIZE — Floating-Point Data Size

The /REAL_SIZE qualifier controls how the compiler interprets floating-point declarations that do not have a specified length.

The qualifier has the following form:

/REAL_SIZE= { 32 | 64 | 128 }

32

Defines REAL declarations, constants, functions, and intrinsics as REAL (KIND=4) (single precision) and COMPLEX declarations, constants, functions, and intrinsics as COMPLEX (KIND=4) (single complex).

64

Defines REAL and COMPLEX declarations, constants, functions, and intrinsics as REAL (KIND=8) (double precision) and COMPLEX declarations, constants, functions, and intrinsics as COMPLEX (KIND=8) (double complex).

This also causes intrinsic functions to produce a double precision REAL (KIND=8) or COMPLEX (KIND=8) result instead of a single precision REAL (KIND=4) or COMPLEX (KIND=4) result, except if the argument is explicitly typed.

For example, references to the CMPLX intrinsic produce DCMPLX results (COMPLEX (KIND=8)), except if the argument to CMPLX is explicitly typed as REAL (KIND=4) or COMPLEX (KIND=4), in which case the resulting data type is COMPLEX (KIND=4). Other affected intrinsic functions include CMPLX, FLOAT, REAL, SNGL, and AIMAG.

128

Specifying /REAL_SIZE=128 defines:

- REAL and DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL (KIND=16) (REAL*16)
- COMPLEX and DOUBLE COMPLEX declarations, constants, functions, and intrinsics as COMPLEX (KIND=16) (COMPLEX*32)

If you omit /REAL_SIZE=128, then:

- REAL declarations, constants, functions, and intrinsics are defined as REAL (KIND=4).
- DOUBLE PRECISION declarations, constants, functions, and intrinsics are defined as REAL (KIND=8).
- COMPLEX declarations, constants, functions, and intrinsics are defined as COMPLEX (KIND=4).

- DOUBLE COMPLEX declarations, constants, functions, and intrinsics are defined as COMPLEX (KIND=8).

Specifying /REAL_SIZE=128 causes REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX intrinsic functions to produce REAL (KIND=16) or COMPLEX (KIND=16) results unless their arguments are typed with an explicit KIND type parameter.

For example, a reference to the CMPLX intrinsic with /REAL_SIZE=128 produces a COMPLEX (KIND=16) result unless the argument is explicitly typed as REAL (KIND=4) or COMPLEX (KIND=4), in which case the result is COMPLEX (KIND=4).

The default is /REAL_SIZE=32.

For More Information:

- On data types, see Chapter 8.
- On intrinsic functions, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.38. /RECURSIVE — Data Storage and Recursion

The /RECURSIVE qualifier requests that VSI Fortran generate code and allocate data so that a subroutine or a function can be called recursively.

The /RECURSIVE qualifier:

- Changes the default allocation class for all local variables from STATIC to AUTOMATIC, except for variables that are data-initialized, named in a SAVE statement, or declared as STATIC.
- Permits references to a routine name from inside the routine.

Subprograms declared with the RECURSIVE keyword are always recursive (whether you specify or omit the /RECURSIVE qualifier).

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 qualifiers).

Specifying /RECURSIVE sets /AUTOMATIC.

For More Information:

On the RECURSIVE keyword, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.39. /REENTRANCY — Specify Threaded or Asynchronous Reentrancy

The /REENTRANCY qualifier specifies whether code generated for the main program and any Fortran procedures it calls will be relying on threaded or asynchronous reentrancy. The default is /REENTRANCY=NONE.

The qualifier has the following form:

/REENTRANCY= { ASYNC | NONE | THREADED }

ASYNC

Informs the VSI Fortran run-time library that the program may contain asynchronous handlers that could call the RTL. The run-time library will guard against asynchronous interrupts inside its own critical regions.

NONE

Informs the VSI Fortran run-time library that the program will not be relying on threaded or asynchronous reentrancy. The run-time library need not guard against such interrupts inside its own critical regions. Same as /NOREENTRANCY.

THREADED

Informs the VSI Fortran run-time library that the program is multithreaded, such as programs using the POSIX threads library. The run-time library will use thread locking to guard its own critical regions.

To use the kernel threads libraries, also specify the /THREADS_ENABLE qualifier on the LINK command (see the *Guide to the POSIX Threads Library*).

Specifying NOREENTRANCY is equivalent to /REENTRANCY=NONE.

For More Information:

On writing multithreaded applications, see the *Guide to the POSIX Threads Library*.

2.3.40. /ROUNDING_MODE — Specify IEEE Floating-Point Rounding Mode

The /ROUNDING_MODE qualifier allows you to control how rounding occurs during calculations. This qualifier applies only to IEEE data types (when you specify /FLOAT=IEEE_FLOAT). Note that if you specify /FLOAT=G_FLOAT or /FLOAT=D_FLOAT, /ROUNDING_MODE must not also be specified.

Note that the rounding mode applies to each program unit being compiled.

The qualifier has the following form:

/ROUNDING_MODE= { NEAREST | CHOPPED | DYNAMIC | MINUS_INFINITY }

NEAREST

This is the normal rounding mode, where results are rounded to the nearest representable value. If you omit the /ROUNDING_MODE qualifier, /ROUNDING_MODE=NEAREST is used

CHOPPED

Results are rounded to the nearest representable value in the direction toward zero.

MINUS_INFINITY

Results are rounded toward the next smallest representative value.

DYNAMIC

Lets you set the rounding mode at run time.

On OpenVMS I64, you can call the SYS$IEEE_SET_ROUNDING_MODE routine to set the rounding mode and obtain the previous rounding mode.

When you call SYS$IEEE_SET_ROUNDING_MODE, you can set the rounding mode to one of the following settings:

- Round toward zero (same as /ROUNDING_MODE=CHOPPED)
- Round toward nearest (same as /ROUNDING_MODE=NEAREST)
- Round toward plus infinity
- Round toward minus infinity (same as /ROUNDING_MODE=CHOPPED)

If you compile with /ROUNDING_MODE=DYNAMIC, the initial rounding mode is set to NEAREST. It will remain NEAREST until you call SYS$IEEE_SET_ROUNDING_MODE to change it.

On OpenVMS Alpha, there is no system routine to call to set the rounding mode dynamically. You have to write and call your own routines to set the rounding mode at run time. The mode is set by setting the rounding control bits in the floating-point control register (FPCR). You can do this in C using the `asm` feature from the system include file `c_asm.h`, or in assembly language.

Note

For the fastest run-time performance, avoid using /ROUNDING_MODE=DYNAMIC.

The rounding mode applies to each program unit being compiled.

For More Information:

- On IEEE floating-point rounding modes, see the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).
- On SYS$IEEE_SET_ROUNDING_MODE, see the *VSI OpenVMS System Services Reference Manual*.
- On the floating-point status register and Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.
- On the floating-point control register and Alpha architecture, see the *Alpha Architecture Reference Manual*.

2.3.41. /SEPARATE_COMPILATION — Control Compilation Unit Use in Object Files

Controls whether the VSI Fortran compiler:

- Places individual compilation units in a VSI Fortran source file as separate object modules in the object file (/SEPARATE_COMPILATION)

- Groups individual compilation units in a VSI Fortran source file as a single object module in the object file (/NOSEPARATE_COMPILATION), which allows more interprocedure optimization

The default is /NOSEPARATE_COMPILATION.

When creating object modules for use in an object library, consider using /SEPARATE_COMPILATION to minimize the size of the routines included by the linker as it creates the executable image. In most cases, to allow more interprocedure optimizations, use the default /NOSEPARATE_COMPILATION.

For More Information:

- On compiling multiple files, see Section 2.2.2.
- On the optimizations that can be performed (such as inlining), see Section 2.3.35.

2.3.42. /SEVERITY — Specify Compiler Diagnostic Severity

The /SEVERITY qualifier changes one or both of the following:

- Changes compiler diagnostic warning messages to have a severity of error (instead of warning). The severity of informational compiler diagnostic messages remains informational severity.

  To control the conditions checked during compilation that prevent or request messages, use the /WARNINGS qualifier keywords (see Section 2.3.51.

- When used with the /STANDARD qualifier, /SEVERITY changes standards checking warning messages to have a severity of error (instead of warning). The severity of informational standards checking diagnostic messages remains informational severity.

  To control the type of standards checking performed, specify the /STANDARDS qualifier with either the F90 keyword (default) or the F95 keyword (see Section 2.3.45.

The qualifier has the following form:

/SEVERITY=WARNINGS= { WARNING | ERROR | STDERROR }

The default is that compiler diagnostic warning messages and standards checking messages have a severity of warning or /SEVERITY=(WARNINGS=WARNING).

You can specify one of the following:

- ERROR

  Specifies that all warning messages are to be issued with ERROR severity.

- STDERROR

  Specifies that if /STANDARD is in effect and diagnostics indicating non-standard features are issued, the diagnostics are issued with ERROR severity (the default is that these are informational). All other warning messages are issued with WARNING severity.

- WARNINGS

  Specifies that all warning messages are to be issued with WARNING severity.

For example, the following command line requests that compiler diagnostic messages have a severity of warning (default) and standards checking messages have a severity of error (and requests Fortran 95 standards checking):

```
$ FORTRAN/SEVERITY=WARNINGS=STDERROR/STANDARD=F95 file.F90
```

For More Information:

- On using the /WARNINGS qualifier to control the conditions checked during compilation, see Section 2.3.51.
- On using the /STANDARDS qualifier to control the type of standards checking performed, see Section 2.3.45.

2.3.43. /SHOW — Control Source Content in Listing File

The /SHOW qualifier controls whether optionally listed source lines and a symbol map appear in the source listing. (Optionally listed source lines are text-module source lines and preprocessor-generated source lines).

For the /SHOW qualifier to take effect, you must specify the /LIST qualifier.

The qualifier has the following form:

/SHOW= { { [NO]DICTIONARY | [NO]INCLUDE | [NO]MAP | [NO]PREPROCESSOR } [, ...] | { ALL | NONE } }

ALL

Requests that all optionally listed source lines and a symbol map be included in the listing file. Specifying /SHOW is equivalent to /SHOW=ALL.

[NO]DICTIONARY

Controls whether VSI Fortran source representations of any CDD/Repository records referenced by DICTIONARY statements are included in the listing file.

[NO]INCLUDE

Controls whether the source lines from any file or text module specified by INCLUDE statements are included in the source listing.

[NO]MAP

Controls whether the symbol map is included in the listing file.

[NO]PREPROCESSOR

Controls whether preprocessor-generated source lines are included in the listing file.

NONE

Requests that no optionally listed source lines or a symbol map be included in the listing file. Specifying /NOSHOW is equivalent to /SHOW=NONE.

The /SHOW qualifier defaults are NOINCLUDE and MAP.

For More Information:

On the /LIST qualifier, see Section 2.3.28.

2.3.44. /SOURCE_FORM — Fortran 90/95 Source Form

The /SOURCE_FORM qualifier allows you to specify whether all VSI Fortran source files on the FORTRAN command line are in fixed or free source form. The qualifier has the following form:

/SOURCE_FORM= { FREE | FIXED }

FIXED

Specifies that the input source files will be in fixed source form, regardless of the file type. Source files with a file type of FOR or F (or any file type other than F90) are assumed to contain fixed source form.

FREE

Specifies that the input source files will be free form, regardless of the file type. Source files with a file type of F90 are assumed to contain free source form.

For More Information:

On column positions and source forms, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.45. /STANDARD — Perform Fortran 90/95 Standards Checking

The /STANDARD qualifier instructs the compiler to generate informational messages for language elements that are not standard in the Fortran 90 or Fortran 95 language and that can be identified at compile-time. The default is /NOSTANDARD. If you specify /STANDARD with no value, the default is Fortran 95.

The qualifier has the following form:

/[NO]STANDARD= { F90 | F95 }

The /STANDARD=F90 qualifier requests that the compiler issue informational messages for:

- Syntax extensions to the Fortran 90 standard. SYNTAX extensions include nonstandard statements and language constructs.
- 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.

The /STANDARD=F95 qualifier (or /STANDARD) requests that the compiler issue informational messages for:

- Syntax extensions to the Fortran 95 standard. SYNTAX extensions include nonstandard statements and language constructs.

- Fortran 95 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.
- Deleted Fortran language features.

Specifying /STANDARD=NONE is equivalent to /NOSTANDARD.

If you specify the /NOWARNINGS qualifier, the /STANDARD qualifier is ignored.

If you omit the /STANDARD qualifier, the default is /NOSTANDARD.

To change the severity of standards checking warning messages to error severity, specify /SEVERITY=WARNINGS=SDTERROR (see Section 2.3.42).

Source statements that do not conform to Fortran 90 or Fortran 95 language standards are detected by the VSI Fortran compiler under the following circumstances:

- The statements contain ordinary syntax and semantic errors.
- A source program containing nonconforming statements is compiled with the /STANDARD or /CHECK qualifiers.

Given these circumstances, the compiler is able to detect *most* instances of nonconforming usage. It does not detect all instances because the /STANDARD qualifier does 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

For More Information:

On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.46. /SYNCHRONOUS_EXCEPTIONS — Report Exceptions More Precisely (Alpha only)

The /SYNCHRONOUS_EXCEPTIONS qualifier associates an exception with the instruction that causes it. Specifying /SYNCHRONOUS_EXCEPTIONS slows program execution, you should specify it only when debugging a specific problem, such as locating the source of an exception.

If you omit /SYNCHRONOUS_EXCEPTIONS, /NOSYNCHRONOUS_EXCEPTIONS is used where exceptions can be reported imprecisely one or more instructions *after* the instruction that caused the exception.

Specifying /IEEE_MODE=FAST (default) provides imprecise exception reporting (same as /NOSYNCHRONOUS_EXCEPTIONS). Specifying other /IEEE_MODE keywords (when you also specify /FLOAT=IEEE_FLOAT) provides precise exception reporting (same as /SYNCHRONOUS_EXCEPTIONS on Alpha systems).

Note that floating-point exceptions are always synchronous on EV6 and later processors, regardless of whether /SYNCHRONOUS_EXCEPTIONS is used.

For More Information:

- On run-time arithmetic exception handling (/IEEE_MODE qualifier), see Section 2.3.24.
- On controlling run-time arithmetic exception messages (/CHECK=FP_EXCEPTIONS qualifier), see Section 2.3.11.
- On the Alpha architecture and instruction pipelining, see the *Alpha Architecture Reference Manual*.

2.3.47. /SYNTAX_ONLY — Do Not Create Object File

The /SYNTAX_ONLY qualifier requests that source file be checked only for correct syntax. If you specify the /SYNTAX_ONLY qualifier, 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 with /WARNINGS=UNINITIALIZED). This qualifier allows you to do a quick syntax check of your source file, and is especially useful in conjunction with /WARNINGS=ARGUMENT_CHECKING.

For More Information:

- On the related qualifier that prevents object file creation (/NOOBJECT), see Section 2.3.33.
- On the /WARNINGS qualifier, see Section 2.3.51.

2.3.48. /VERSION — Display the VSI Fortran Version Number

The /VERSION qualifier can be used alone on the FORTRAN command line to display the VSI Fortran version number. If you specify /VERSION, compilation does not occur.

For example:

```
$ FORTRAN /VERSION
VSI Fortran V8.n-nnn-nnnn
$
```

2.3.49. /TIE — Enable Code for Shared Translated Images

The /TIE qualifier enables compiled code to be used with translated shared images, either because the code might call into a translated image or might be called from a translated image. Specifying /NOTIE, the default, indicates the compiled code will not be associated with a translated image.

If you specify /TIE, link the object module using the LINK command /NONATIVE_ONLY qualifier, and follow the guidelines provided in Section B.6.

For More Information:

On appropriate LINK command qualifiers, see Section 3.2.1 and Section B.6.

2.3.50. /VMS — Request Compaq Fortran 77 for OpenVMS VAX Compatibility

The /VMS qualifier specifies that the run-time system behave like Compaq Fortran 77 for OpenVMS VAX Systems (VAX FORTRAN) in certain ways. To prevent this behavior, specify /NOVMS.

This qualifier is the default.

The /VMS qualifier specifies the following aspects of the run-time system:

- Sets the defaults for the following qualifiers:
  - /CHECK=FORMAT

    Specifying /NOVMS changes the default to /CHECK=NOFORMAT (see Section 2.3.11).
  - /CHECK=OUTPUT_CONVERSION

    Specifying /NOVMS changes the default to /CHECK=NOOUTPUT_CONVERSION (see Section 2.3.11).

Note that the /VMS qualifier allows use of /LIST or /NOLIST in INCLUDE statement specifications. (Also see /ASSUME=SOURCE_INCLUDE to control included directory).

To override the effects of the /VMS qualifier, specify /NOVMS.

For More Information:

On the OPEN statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.3.51. /WARNINGS — Warning Messages and Compiler Checking

The /WARNINGS qualifier instructs the compiler to generate informational (I-level) and warning (W-level) diagnostic messages in response to informational and warning-level errors.

The following /WARNINGS keywords prevent the display of warning messages:

NOALIGNMENT (default is ALIGNMENT)
NOGENERAL (default is GENERAL)
NOGRANULARITY (default is GRANULARITY)
NOUNCALLED (default is UNCALLED)
NOUNINITIALIZED (default is UNINITIALIZED)
NOUSAGE (default is USAGE)

The following /WARNINGS keywords can display additional warning messages (request additional checking):

ARGUMENT_CHECKING (default is NOARGUMENT_CHECKING)
DECLARATIONS (default is NODECLARATIONS)
IGNORE_LOC (default is NOIGNORE_LOC)
TRUNCATED_SOURCE (default is NOTRUNCATED_SOURCE)
UNUSED (default is NOUNUSED)

If you omit /WARNINGS, the defaults are:

/WARNINGS=(ALIGNMENTS,NOARGUMENT_CHECKING,
NODECLARATIONS, NOERRORS,FILEOPTS,GENERAL,GRANULARITY,
NOIGNORE_LOC,NOSTDERRORS,NOTRUNCATED_SOURCE,UNCALLED, UNINITIALIZED,
NOUNUSED,USAGE)

The qualifier has the following form:

/WARNINGS= { { [NO]ALIGNMENT | [NO]ARGUMENT_CHECKING | [NO]DECLARATIONS | [NO]GENERAL | [NO]GRANULARITY | [NO]IGNORE_LOC | [NO]TRUNCATED_SOURCE | [NO]UNCALLED | [NO]UNINITIALIZED | [NO]UNUSED | [NO]USAGE } [, ...] | { ALL | NONE } }

[NO]ALIGNMENT

Controls whether the compiler issues diagnostic messages when variables or arrays (created in COMMON or EQUIVALENCE statements) are declared in such a way that they cross natural boundaries for their data size. For example, a diagnostic message is issued if /WARNINGS=ALIGNMENT is in effect and the virtual address of a REAL (KIND=8) variable is not a multiple of 8.

The default is ALIGNMENT. To suppress diagnostic messages about unaligned data, specify NOALIGNMENT.

To control the alignment of fields in common blocks, derived types, and record structures, use the /ALIGNMENT qualifier (see Section 2.3.3).

[NO]ARGUMENT_CHECKING

Controls whether the compiler issues diagnostic messages for argument mismatches between caller and callee (when compiled together). The default is /WARNINGS=NOARGUMENT_CHECKING.

[NO]DECLARATIONS

Controls whether the compiler issues diagnostic messages for any untyped data item used in the program. DECLARATIONS acts as an external IMPLICIT NONE declaration. See the description of the IMPLICIT statement in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] for information about the effects of IMPLICIT NONE.

The default is NODECLARATIONS.

[NO]GENERAL

Controls whether the compiler issues I-level and W-level diagnostic messages. An I-level message indicates that a correct VSI Fortran statement may have unexpected results or contains nonstandard syntax or source form. A W-level message indicates that the compiler has detected acceptable, but nonstandard, syntax or has performed some corrective action; in either case, unexpected results may occur.

To suppress I-level and W-level diagnostic messages, specify the negative form of this qualifier (/WARNINGS=NOGENERAL).

The default is GENERAL.

[NO]GRANULARITY

Controls whether the compiler issues the compiler diagnostic message "Unable to generate code for requested granularity" to indicate when data access might not occur safely from different threads in a shared memory system (such as asynchronous write access from outside the user process). The default is /WARNINGS=GRANULARITY.

For more information on data granularity and controlling the size of data that can be safely accessed from different threads, see Section 2.3.23.

[NO]IGNORE_LOC

Requests that the compiler issue warnings when %LOC is stripped from an argument due to specification of the IGNORE_LOC attribute. The default is /WARNINGS=NOIGNORE_LOC (does not issue a warning for this condition).

[NO]TRUNCATED_SOURCE

Controls whether the compiler issues a warning diagnostic message (EXCCHASRC) when it reads a fixed-form source line with a statement field that exceeds the maximum column width. The maximum column width is column 72 or 132, depending whether /EXTEND_SOURCE qualifier was specified (or its OPTIONS statement qualifier).

This option has no effect on truncation; lines that exceed the maximum column width are always truncated.

The default is /WARNINGS=NOTRUNCATED_SOURCE.

[NO]UNCALLED

Suppresses the compiler warning diagnostic message when a statement function is never called. The default is /WARNINGS=UNCALLED.

[NO]UNINITIALIZED

Controls whether warning messages are issued when a variable is referenced before a value was assigned to it. Specify NOUNINITIALIZED to suppress such warning messages.

The default, /WARNINGS=UNINITIALIZED, issues warning messages when a variable is referenced before a value was assigned to it.

[NO]UNUSED

Requests warning messages for a variable that is declared but never used. The default is /WARNINGS=NOUNUSED.

[NO]USAGE

Specifying /WARNINGS=NOUSAGE suppresses informational messages about questionable programming practices and the use of intrinsic functions that use only two digits to represent years (such as 2000). The compiler allows such programming practices, although they are often an artifact of programming errors. For example, a continued character or Hollerith literal whose first part ends before the statement field and appears to end with trailing spaces.

The default is USAGE.

ALL

Causes the compiler to print all I-level and W-level diagnostic messages, including warning messages for any unaligned data and untyped data items. Specifying ALL is equivalent to /WARNINGS and has the effect of specifying /WARNINGS=(ALIGNMENT, DECLARATIONS, GENERAL, UNCALLED, UNINITIALIZED).

NONE

Suppresses all I-level and W-level messages. Specifying /NOWARNINGS is equivalent to /WARNINGS=NONE.

For More Information:

- On compiler diagnostic messages, see Section 2.6.
- On run-time checking for various conditions (/CHECK qualifier), see Section 2.3.11.
- On changing the severity of compiler diagnostic warnings messages to error severity, see Section 2.3.42.

2.4. Creating and Maintaining Text Libraries

A text library contains library modules of source text. To include a library module from a text library in a program, use an INCLUDE statement.

Library modules within a text library are like ordinary text files, but they differ in the following ways:

- They contain a unique name, called the library module name, that is used to access them.
- Several library modules can be contained within the same library file.

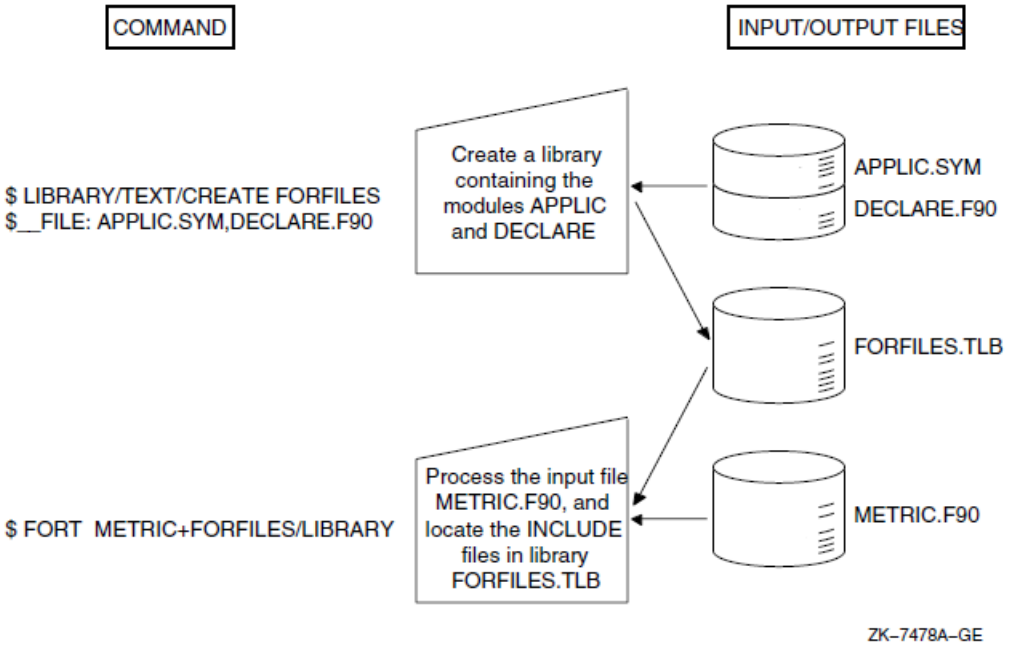
Library modules in text libraries can contain any kind of text; this section only discusses their use when VSI Fortran language source is used.

You should be aware of the difference between library modules that reside in text libraries and the VSI Fortran post-compiled module files (.F90$MOD file type) that support use association (see Section 2.2.3).

Use the LIBRARY command (OpenVMS Librarian Utility) to create text libraries and insert, replace, delete, copy, or list library modules in text libraries. Text libraries have a default file type of TLB.

Figure 2.2 shows the creation of a text library and its use in compiling VSI Fortran programs.

Figure 2.2. Creating and Using a Text Library



For More Information:

- On using the logical name FORT$LIBRARY to define a default library, see Section 2.2.4.
- On specifying the name of the library using the /LIBRARY qualifier on the FORTRAN command line, see Section 2.3.27.
- On specifying the directory location of the library using the /INCLUDE qualifier, see Section 2.3.25.
- On specifying the directory location of the library using the /ASSUME=SOURCE_INCLUDE qualifier, see Section 2.3.7.
- On using Fortran modules, see Section 2.2.3.
- On using INCLUDE or USE statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

2.4.1. Using the LIBRARY Commands

Table 2.3 summarizes the commands that create libraries and provide maintenance functions. For a complete list of the qualifiers for the LIBRARY command and a description of other DIGITAL Command Language (DCL) commands listed in Table 2.3, see the *Guide to Using VMS Command Procedures* or enter HELP LIBRARY.

Table 2.3. Commands to Control Library Files

Function	Command Syntax[1]
Create a library.	LIBRARY/TEXT/CREATE *library-name file-spec*,...
Add one or more library modules to a library.	LIBRARY/TEXT/INSERT *library-name file-spec*,...

Function	Command Syntax[1]
Replace one or more library modules in a library.	LIBRARY/TEXT/REPLACE *library-name file-spec*,... [2]
Specify the names of library modules to be added to a library.	LIBRARY/TEXT/INSERT *library-name file-spec*/MODULE= *module-name*
Delete one or more library modules from a library.	LIBRARY/TEXT/DELETE=(*module-name*,...) *library-name*
Copy a library module from a library into another file.	LIBRARY/TEXT/EXTRACT= *module-name*/OUTPUT=*file-spec library-name*
List the library modules in a library.	LIBRARY/TEXT/LIST= *file-spec library-name*

[1]The LIBRARY command qualifier /TEXT indicates a text module library. By default, the LIBRARY command assumes an object module library.

[2]REPLACE is the default function of the LIBRARY command if no other action qualifiers are specified. If no library module exists with the given name, /REPLACE is equivalent to /INSERT.

2.4.2. Naming Text Library Modules

When the LIBRARY command adds a library module to a library, by default it uses the file name of the input file as the name of the library module. In the example in Figure 2.2, the LIBRARY command adds the contents of the files APPLIC.SYM and DECLARE.FOR to the library and names the library modules APPLIC and DECLARE.

Alternatively, you can name a module in a library with the /MODULE qualifier. For example:

```
$ LIBRARY/TEXT/INSERT FORFILES DECLARE.FOR /MODULE=EXTERNAL_DECLARATIONS
```

The preceding command inserts the contents of the file DECLARE.FOR into the library FORFILES under the name EXTERNAL_DECLARATIONS. This library module can be included in a VSI Fortran source file during compilation with the following statement:

```
INCLUDE 'FORFILES(EXTERNAL_DECLARATIONS)'
```

For More Information:

- On the INCLUDE statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] and Section 2.2.4.1.
- On controlling the directories and libraries searched with FORTRAN command qualifiers and the FORT$INCLUDE logical name, see Section 2.2.4.1.

2.5. Using CDD/Repository

CDD/Repository is an optional software product available under a separate license. The CDD/Repository product allows you to maintain shareable data definitions (language-independent structure declarations) that are defined by a data or repository administrator.

Note

CDD/Repository supports both the Common Data Dictionary (Version 3) and CDD/Plus (Version 4) interfaces. Older dictionary versions need to be converted to repository (CDD/Repository) format using

a supplied conversion utility. For detailed information about CDD/Repository, see the CDD/Repository documentation.

CDD/Repository data definitions are organized hierarchically in much the same way that files are organized in directories and subdirectories. For example, a repository for defining personnel data might have separate directories for each employee type.

Descriptions of data definitions are entered into the dictionary in a special-purpose language called CDO (Common Dictionary Operator, which replaces the older interface called CDDL, Common Data Dictionary Language). CDD/Repository converts the data descriptions to an internal form—making them independent of the language used to access them—and inserts them into the repository.

During the compilation of a VSI Fortran program, CDD/Repository data definitions can be accessed by means of DICTIONARY statements. If the data attributes of the data definitions are consistent with VSI Fortran requirements, the data definitions are included in the VSI Fortran program. CDD/Repository data definitions, in the form of VSI Fortran source code, appear in source program listings if you specify the /SHOW=DICTIONARY qualifier on the FORTRAN command line or the /LIST qualifier in the DICTIONARY statement.

The advantage in using CDD/Repository instead of VSI Fortran source for structure declarations is that CDD/Repository record declarations are language-independent and can be used with several supported OpenVMS languages.

The following examples demonstrate how data definitions are written for CDD/Repository. The first example is a structure declaration written in CDDL. The second example shows the same structure as it would appear in a FORTRAN output listing.

CDO Representation:

```
define field name
datatype is text
size is 30.
define field address
datatype is text
size is 40.
define field salesman-id
datatype is longword unsigned
size is 4.
define record payroll-record
name.
address.
salesman-id.
end record.
```

VSI Fortran Source Code Representation:

```
STRUCTURE /PAYROLL_RECORD/
STRUCTURE SALESMAN
CHARACTER*30 NAME
CHARACTER*40 ADDRESS
STRUCTURE SALESMAN_ID
BYTE %FILL(4)
END STRUCTURE
END STRUCTURE
END STRUCTURE
```

2.5.1. Accessing CDD/Repository from VSI Fortran Programs

A repository or data administrator uses CDO to create repositories, define directory structures, and insert record and field definitions into the repository. Many repositories can be linked together to form one logical repository. If the paths are set up correctly, users can access definitions as if they were in a single repository regardless of physical location.

CDO also creates the record paths. Once established, records can be extracted from the repository by means of DICTIONARY statements in VSI Fortran programs. At compile time, the record definition and its attributes are extracted from the designated repository. Then the compiler converts the extracted record definition into a VSI Fortran structure declaration and includes it in the object module.

The DICTIONARY statement incorporates CDD/Repository data definitions into the current VSI Fortran source file during compilation. The DICTIONARY statement can occur anywhere in a VSI Fortran source file that a specification statement (such as a STRUCTURE...END STRUCTURE block) is allowed. The format of the DICTIONARY statement is described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

A DICTIONARY statement must appear as a statement by itself; it cannot be used within a VSI Fortran structure declaration. For example, consider the following DICTIONARY statement:

```
INTEGER*4 PRICE DICTIONARY 'ACCOUNTS'
```

This would result in a declaration of the following form:

```
INTEGER*4 PRICE
STRUCTURE /ACCOUNTS/
STRUCTURE NUMBER
CHARACTER*3 LEDGER
CHARACTER*5 SUBACCOUNT
END STRUCTURE
CHARACTER*12 DATE
    .
    .
    .
END STRUCTURE
```

When you extract a record definition from the repository, you can choose to include this translated record in the program's listing by using the /LIST qualifier in the DICTIONARY statement or the /SHOW=DICTIONARY qualifier in the FORTRAN command line.

CDD/Repository data definitions can contain explanatory text in the DESCRIPTION IS clause. If you specify /SHOW=DICTIONARY on the FORTRAN command (or /LIST in the DICTIONARY statement), this text is included in the VSI Fortran output listing as comments.

Because the DICTIONARY statement generally contains only structure declaration blocks, you will usually also need to include one or more RECORD statements in your program to make use of these structures. (See the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] for information about structure declaration blocks or the RECORD statement).

2.5.2. VSI Fortran and CDD/Repository Data Types

The CDD/Repository supports some data types that are not native to VSI Fortran. If a data definition contains a field declared with an unsupported data type, VSI Fortran replaces the field with one declared

as an inner STRUCTURE containing a single unnamed field (%FILL field) that is a BYTE array with an appropriate dimension. The VSI Fortran compiler does not attempt to approximate a data type that it does not support.

For example, CDD/Repository's data type UNSIGNED LONG is not supported by VSI Fortran. As a result, if the field FIELD1 is declared to be UNSIGNED LONG using CDO, VSI Fortran replaces the definition of FIELD1 with the following declaration:

```
STRUCTURE FIELD1
     BYTE %FILL(4)
END STRUCTURE
```

VSI Fortran does not declare it as INTEGER*4, which results in signed operations if the field was used in an arithmetic expression.

Table 2.4 summarizes the CDO data types and corresponding VSI Fortran data types.

Table 2.4. CDO Data Types and Corresponding VSI Fortran Data Types

CDO Data Type	VSI Fortran Data Type
DATE	STRUCTURE (length 8)
DATE AND TIME	STRUCTURE (length n)
VIRTUAL	ignored
BIT m ALIGNED	STRUCTURE (length n+7/8)
BIT m	STRUCTURE (length n+7/8)
UNSPECIFIED	STRUCTURE (length n)
TEXT	CHARACTER*n
VARYING TEXT	STRUCTURE (length n)
VARYING STRING	STRUCTURE (length n)
D_FLOATING	REAL*8 (/FLOAT=D_FLOAT only)
D_FLOATING COMPLEX	COMPLEX*16 (/FLOAT=D_FLOAT only)
F_FLOATING	REAL*4 (/FLOAT=D_FLOAT or /FLOAT=G_FLOAT)
F_FLOATING COMPLEX	COMPLEX*8 (/FLOAT=D_FLOAT or /FLOAT=G_FLOAT)
G_FLOATING	REAL*8 (/FLOAT=G_FLOAT only)
G_FLOATING COMPLEX	COMPLEX*16 (/FLOAT=G_FLOAT only)
H_FLOATING	STRUCTURE (length 16)
H_FLOATING COMPLEX	STRUCTURE (length 32)
SIGNED BYTE	LOGICAL*1
UNSIGNED BYTE	STRUCTURE (length 1)
SIGNED WORD	INTEGER*2
UNSIGNED WORD	STRUCTURE (length 2)
SIGNED LONGWORD	INTEGER*4
UNSIGNED LONGWORD	STRUCTURE (length 4)
SIGNED QUADWORD	STRUCTURE (length 8)
UNSIGNED QUADWORD	STRUCTURE (length 8)

CDO Data Type	VSI Fortran Data Type
SIGNED OCTAWORD	STRUCTURE (length 16)
UNSIGNED OCTAWORD	STRUCTURE (length 16)
PACKED NUMERIC	STRUCTURE (length n)
SIGNED NUMERIC	STRUCTURE (length n)
UNSIGNED NUMERIC	STRUCTURE (length n)
LEFT OVERPUNCHED	STRUCTURE (length n)
LEFT SEPARATE	STRUCTURE (length n)
RIGHT OVERPUNCHED	STRUCTURE (length n)
RIGHT SEPARATE	STRUCTURE (length n)

Note

D_floating and G_floating data types cannot be mixed in one subroutine because both types cannot be handled simultaneously. You can use both types, each in a separate subroutine, depending on the OPTIONS statement qualifier in effect for the individual subroutine. For a discussion of the handling of REAL*8 data types in VSI Fortran, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

The compiler issues an error message if it encounters a CDD/Repository feature that conflicts with VSI Fortran. It ignores any CDD/Repository features that it does not support.

2.6. Compiler Limits, Diagnostic Messages, and Error Conditions

The following sections discuss the compiler limits and error messages.

2.6.1. Compiler Limits

Table 2.5 lists the limits to the size and complexity of a single VSI Fortran program unit and to individual statements contained in it.

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined by process quotas and system parameters.

Table 2.5. Compiler Limits

Language Element	Limit
Actual number of arguments per CALL or function reference	255
Arguments in a function reference in a specification expression	255
Array dimensions	7
Array elements per dimension	9,223,372,036,854,775,807 or process limit
Constants; character and Hollerith	2000 characters
Constants; characters read in list-directed I/O	2048 characters

Language Element	Limit
Continuation lines	511
DO and block IF statement nesting (combined)	128
DO loop index variable	9,223,372,036,854,775,807 or process limit
Format group nesting	8
Fortran source line length	132 characters
INCLUDE file nesting	20 levels
Labels in computed or assigned GOTO list	500
Lexical tokens per statement	3000
Parentheses nesting in expressions	40
Structure nesting	20
Symbolic-name length	63 characters

For More Information:

On relevant process quotas and system parameters, see Section 1.2.

2.6.2. Compiler Diagnostic Messages and Error Conditions

The VSI Fortran compiler identifies syntax errors and violations of language rules in the source program.

If the compiler locates any errors, it writes messages to your default output device; so, if you enter the FORTRAN command interactively, the messages are displayed on your terminal. If the FORTRAN command is executed in a batch job, the messages appear in the log file for the batch job.

A sample message from the compiler as it would appear on a terminal screen follows:

```
40 FORMAT (I3,)
...............^
%F90-W-ERROR, An extra comma appears in the format list.
at line number 13 in file DISK$:[USER]SAMP_MESS.FOR;4
```

This sample message consists of the following lines:

- The source file line that caused the message
- A pointer (...^ on a terminal or ...1 in a listing file) to the part of the source line causing the error
- The actual message (facility, severity, mnemonic, and text)
- The line number and name of source file

The message line has the following format:

```
%F90-s-ident, message-text
```

The facility, *F90*, follows the percent sign (%). The severity of the message (I for informational, W for warning, E for error, or F for fatal) replaces `s`. A mnemonic for that message replaces `ident`. The explanatory text of the message replaces `message-text`.

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it. If you compile using the /OPTIMIZE qualifier (the default) and have difficulty determining the cause of an error (possibly because of the optimizations), consider compiling using a lower optimization level or /NOOPTIMIZE.

If the compiler creates a listing file, it also writes the messages to the listing. The pointer (...1) and messages follow the statement that caused the error.

2.7. Compiler Output Listing Format

A compiler output listing produced by a FORTRAN command with the /LIST qualifier consists of the following sections:

- A source code section
- A machine code section – optional
- A storage map section (cross-reference) – optional
- A compilation summary

Section 2.7.1 through Section 2.7.5 describe the compiler listing sections in detail.

For More Information:

- On the /LIST qualifier, see Section 2.3.28)
- On the /MACHINE_CODE qualifier, see Section 2.3.29.
- On the /SHOW qualifier, see Section 2.3.43.

2.7.1. Source Code Section

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 2.1 and Example 2.2 show a sample of a source code section of a free-form compiler output listing.

Example 2.1. Sample Listing of Source Code on OpenVMS I64

```
SIMPLE$MAIN$BLK   Source Listing  26-OCT-2004 15:29:44  HP Fortran
 V8.0-48291
                                                                Page
 1
                                 26-OCT-2004 15:29:26
                                 DISK$DKA100:[USERNAME.BUG]SIMPLE.F90;1

       1  integer i
       2  i = 10
       3  type *,i
       4  end


PROGRAM SECTIONS

    Name                 Bytes   Attributes
```

```
  1 $LITERAL$                     24   PIC CON REL LCL   SHR NOEXE   RD
 NOWRT OCTA
  2 $CODE$                       224   PIC CON REL LCL   SHR   EXE NORD
 NOWRT OCTA
  3 $SBSS$                        12 NOPIC CON REL LCL NOSHR NOEXE   RD
 WRT OCTA

    Total Space Allocated                 260


ENTRY POINTS

 Address      Name

  2-00000000  SIMPLE$MAIN


VARIABLES

 Address      Type  Name

  3-00000008  I*4   I
```

Example 2.2. Sample Listing of Source Code on OpenVMS Alpha

```
RELAX2      Source Listing  19-DEC-2004 16:12:46  HP Fortran V8.x-xxxx
   Page 1
                            12-DEC-2004 10:41:04  F90$DISK:
[TUCKER]LIST_EX.F90;3

       1    SUBROUTINE RELAX2(EPS)
       2      INTEGER, PARAMETER :: M=40
       3      INTEGER, PARAMETER :: N=60
       4
       5      COMMON X (M,N)
       6
       7      LOGICAL DONE
       8  10  DONE = .TRUE.
       9
      10      DO J=1,N-1
      11        DO I=1,M-1
      12          XNEW = (X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))/4
      13          IF (ABS(XNEW-X(I,J)) > EPS) DONE = .FALSE.
      14          X(I,J) = XNEW
      15        END DO
      16      END DO
      17
      18      IF (.NOT. DONE) GO TO 10
      19      RETURN
      20      END SUBROUTINE
```

Compiler-generated line numbers appear in the left margin and are used with the %LINE prefix in debugger commands. These compiler-generated line numbers are displayed by compiler-generated messages and certain run-time error messages. (For more information on messages, see Appendix C).

2.7.2. Machine Code Section

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 a MACRO assembly listing.

The machine code section is optional. To receive a listing file with a machine code section, you must specify the following:

```
$ FORTRAN/LIST/MACHINE_CODE
```

Example 2.3 shows a sample of part of a machine code section of a compiler output listing for OpenVMS I64 systems.

Example 2.3. Sample (Partial) Listing of Machine Code on OpenVMS I64

```
SIMPLE$MAIN$BLK  Machine Code Listing  26-OCT-2004 15:29:34
                                       HP Fortran V8.0-48291   Page 2

                                       26-OCT-2004 15:29:26
                                       DISK$DKA100:[USERNAME.BUG]SIMPLE.F90;1

          .psect $CODE$, CON, LCL, SHR, EXE, NOWRT, NOVEC, NOSHORT
          .proc   MAIN__
          .align 32
          .global  MAIN__
          .personality  DFOR$HANDLER
          .handlerdata  -32
   MAIN__:               // 000002
       { .mii
002C0081A980     0000        alloc   r38 = rspfs, 0, 8, 5, 0
0119F8C80300     0001        adds    sp = -64, sp   // r12 = -64, r12
0108001009C0     0002        mov     r39 = gp ;;    // r39 = r1
       }
       { .mib
010800C40080     0010        adds    r2 = 32, sp   // r2 = 32, r12
000188000940     0011        mov     r37 = rp      // r37 = br0
004000000000     0012        nop.b   0 ;;
       }
       { .mii
008CC0200000     0020        st8     [r2] = r0
0120001000C0     0021        add     r3 = @ltoff($LITERAL$+16), gp
                             // r3 = @ltoff($LITERAL$+16), r1
012000002640     0022        mov     ai = 1    // r25 = 1
       }
       { .mmi
012000014880     0030        mov     r34 = 10 ;;
0080C03000C0     0031        ld8     r3 = $LITERAL$   // r3 = [r3]
000008000000     0032        nop.i   0 ;;
       }
       { .mfb
010800300A00     0040        mov     out0 = r3    // r40 = r3
000008000000     0041        nop.f   0
00A000001000     0042        br.call.sptk.many rp = DFOR$SET_FPE ;;
                             // br0 = DFOR$SET_FPE
       }
       { .mii
010802700040     0050        mov     gp = r39    // r1 = r39
012000002640     0051        mov     ai = 1 ;;   // r25 = 1
012000100800     0052        add     r32 = @ltoff($LITERAL$+8), gp ;;
                             // r32 = @ltoff($LITERAL$+8), r1
       }
       { .mfb
0080C2000A00     0060        ld8     out0 = $LITERAL$   // r40 = [r32]
000008000000     0061        nop.f   0
00A000001000     0062        br.call.sptk.many rp = DFOR$SET_REENTRANCY ;;
```

```
                            // br0 = DFOR$SET_REENTRANCY
                  }
      { .mii
010802700040      0070        mov     gp = r39    // r1 = r39
010800C208C0      0071        adds    r35 = 16, sp   // r35 = 16, r12      // 000003
010800C40840      0072        adds    r33 = 32, sp   // r33 = 32, r12
      }
      { .mmi
01200000A640      0080        mov     ai = 5 ;;    // r25 = 5
012000100900      0081        add     r36 = @ltoffx($LITERAL$), gp
                              // r36 = @ltoffx($LITERAL$), r1
010800C20B00      0082        adds    out4 = 16, sp   // r44 = 16, r12
      }
      { .mlx
0119F80FCA40      0090        adds    out1 = -2, r0   // r41 = -2, r0
000000000002      0091        movl    out2 = 8716160 ;;   // r42 = 8716160
019FF20015
      }
      { .mii
0080C2400AC0      00A0        ld8.mov out3 = [r36], $LITERAL$
010800C40A00      00A1        adds    out0 = 32, sp   // r40 = 32, r12
000008000000      00A2        nop.i   0 ;;
      }
      { .mmb
008C82344000      00B0        st4     [r35] = r34
008C82100000      00B1        st4     [r33] = r0
00A000001000      00B2        br.call.sptk.many rp = DFOR$WRITE_SEQ_LIS ;;
                              // br0 = DFOR$WRITE_SEQ_LIS

      }
      { .mii
012000002200      00C0        mov     r8 = 1            // 000004
010802700040      00C1        mov     gp = r39    // r1 = r39       // 000003
00015404C000      00C2        mov.i   rspfs = r38 ;;           // 000004
      }
      { .mib
010800C80300      00D0        adds    sp = 64, sp   // r12 = 64, r12
000E0014A000      00D1        mov     rp = r37    // br0 = r37
000108001100      00D2        br.ret.sptk.many rp ;;   // br0
      }
          .endp   MAIN__

Routine Size: 224 bytes,    Routine Base: $CODE$ + 0000

          .psect $SBSS$, CON, LCL, NOSHR, NOEXE, WRT, NOVEC, SHORT
          .lcomm  var$0004, 1, 1
          .lcomm  fill$$1, 11, 16

          .psect $LITERAL$, CON, LCL, SHR, NOEXE, NOWRT, NOVEC, NOSHORT
00010109          0000        string  "\X09\X01\X01\X00"

00000000          0008        data8   0x0 // data8 0
00000000          000C
00200000          0010        data8   0x200000 // data8 2097152
00000000          0014
```

The following list explains how generated code and data are represented in machine code listings on OpenVMS I64 systems:

- Machine instruction bundles are marked by curly braces, and the bundle type is at the top after the open brace.

- Machine instructions in a bundle are shown in hexadecimal on the far left. The next column to the right is the hexadecimal address offset of the instruction from the routine base address. The next

column is a symbolic representation, using memnonics and symbolic operand expressions (such as “out1”). The actual operand registers are shown in comments in the next column to the right (such as “r40”).

- A predicate register controlling an instruction is shown in parentheses before the mnemonic representation.
- Line numbers corresponding to the generated code are shown on the far right side of the listing.

Example 2.4 shows a sample of part of a machine code section of a compiler output listing for OpenVMS Alpha systems.

Example 2.4. Sample (Partial) Listing of Machine Code on OpenVMS Alpha

```
RELAX2  Machine Code Listing  19-DEC-2004 16:12:46 HP Fortran V8.x-xxxx
                                                              Page 3
                              12-DEC-2004 10:41:04
                              F90$DISK:[TUCKER]LIST_EX.F90;3

             .PSECT $CODE$, OCTA, PIC, CON, REL, LCL, SHR,-
                  EXE, NORD, NOWRT

      0000 RELAX2::
23DEFFC0     0000  LDA SP, -64(SP)
80100000     0004  LDF F0, (R16)                ; 000013
A41B0020     0008  LDQ R0, 32(R27)              ; 000012
B77E0000     000C  STQ R27, (SP)                ; 000001
B75E0008     0010  STQ R26, 8(SP)
B7BE0010     0014  STQ FP, 16(SP)
9C5E0018     0018  STT F2, 24(SP)
9C7E0020     001C  STT F3, 32(SP)
9C9E0028     0020  STT F4, 40(SP)
9CBE0030     0024  STT F5, 48(SP)
9CDE0038     0028  STT F6, 56(SP)
47FE041D     002C  MOV SP, FP
803B0028     0030  LDF F1, 40(R27)              ; 000012
2FFE0000     0034  UNOP
2FFE0000     0038  UNOP
2FFE0000     003C  UNOP
      0040 .10:                                 ; 000008
2FFE0000     0040  UNOP
2FFE0000     0044  UNOP
2FFE0000     0048  UNOP
2FFE0000     004C  UNOP
203FFFFF     0050  MOV -1, DONE ; -1, R1
47E77411     0054  MOV 59, var$0002 ; 59, R17 ; 000010
47F41412     0058  MOV 160, R18               ; 000014
2FFE0000     005C  UNOP
      0060 lab$0004:                            ; 000010
2FFE0000     0060  UNOP
2FFE0000     0064  UNOP
2FFE0000     0068  UNOP
2FFE0000     006C  UNOP
40120413     0070  ADDQ R0, R18, R19
47E4F414     0074  MOV 39, var$0003 ; 39, R20 ; 000011
2273FF5C     0078  LDA R19, -164(R19)           ; 000012
2FFE0000     007C  UNOP
      0080 lab$0008:                            ; 000011
2FFE0000     0080  UNOP
2FFE0000     0084  UNOP
2FFE0000     0088  UNOP
2FFE0000     008C  UNOP
81530008     0090  LDF F10, 8(R19)
```

```
8193FF64     0094  LDF F12, -156(R19)
47FF0418     0098  CLR     R24                   ; 000013
81730000     009C  LDF F11, (R19)                ; 000012
81B300A4     00A0  LDF F13, 164(R19)
81D3000C     00A4  LDF F14, 12(R19)
8213FF68     00A8  LDF F16, -152(R19)
823300A8     00AC  LDF F17, 168(R19)
82730010     00B0  LDF F19, 16(R19)
82B3FF6C     00B4  LDF F21, -148(R19)
82D300AC     00B8  LDF F22, 172(R19)
82F30014     00BC  LDF F23, 20(R19)
8333FF70     00C0  LDF F25, -144(R19)
835300B0     00C4  LDF F26, 176(R19)
83730018     00C8  LDF F27, 24(R19)
   .
   .
   .
Routine Size: 1020 bytes,    Routine Base: $CODE$ + 0000


    .PSECT $LINK$, OCTA, NOPIC, CON, REL, LCL,-
     NOSHR, NOEXE, RD, NOWRT

      0000  ; Stack-Frame invocation descriptor
    Entry point:         RELAX2
    Entry Length:           48
    Registers used:         R0-R1, R16-R25, R27-FP,
                                                    F0-F6, F10-F30
    Registers saved:        FP, F2-F6
    Fixed Stack Size:       64

00000000     0020  .ADDRESS $BLANK
00003F80     0028  .LONG X^3F80 ; .F_FLOATING 0.2500000
```

The following list explains how generated code and data are represented in machine code listings on OpenVMS Alpha systems:

- Machine instructions are represented by MACRO mnemonics and syntax. To enable you to identify the machine code that is generated from a particular line of source code, the compiler-generated line numbers that appear in the source code listing are also used in the machine code listing. These numbers appear in the right margin, preceding the machine code generated from individual lines of source code.
- The first line contains a .TITLE assembler directive, indicating the program unit from which the machine code was generated:
  - For a main program, the title is as declared in a PROGRAM statement. If you did not specify a PROGRAM statement, the main program is titled *filename*$MAIN, where *filename* is the name of the source file.
  - For a subprogram, the title is the name of the subroutine or function.
  - For a BLOCK DATA subprogram, the title is either the name declared in the BLOCK DATA statement or *filename*$DATA (by default).
- The lines following .TITLE provide information such as the contents of storage initialized for FORMAT statements, DATA statements, constants, and subprogram argument call lists.
- General registers (0 through 31) are represented by R0 through R31 and floating-point registers are represented by F0 through F31. The relative PC for each instruction or data item is listed at the left margin, in hexadecimal.

- 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 some names may be missing).
- VSI 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 /NOOPTIMIZE.
- The compiler might generate labels for its own use. These labels appear as L$ *n* or lab$ *000n*, where the value of *n* is unique for each such label in a program unit.
- Integer constants are shown as signed integer values; real and complex constants are shown as unsigned hexadecimal values preceded by ^X.
- 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 PSECT lines.

2.7.3. Annotations Section

The annotations section of the compiler output listing describes special instructions used for optimizations such as prefetching, inlining, and loop unrolling. It is controlled by the /ANNOTATIONS qualifier. See Section 2.3.5.

Example 2.5 shows the list of annotations that is produced from the following source code:

```
1 2 3 4 5 6 7            287 work1=sqrt((xlatt*real(mindex,LONGreal))
                             **2+(ylatt*real(nindex,LONGreal))**2+(q-p)**2)
8 9 10 11 12             288 work2=q-p
```

Example 2.5. Sample (Partial) Listing of Annotations

```
1. Software pipelining across 4 iterations; unrolling loop 2 times;
   steady state estimated 180 cycles; 0 prefetch iterations
2. Unrolling loop 6 times
3. Prefetching MINDEX, 128 bytes ahead
4. Prefetching NINDEX, 128 bytes ahead
5. Prefetching Q, 128 bytes ahead
6. Write-hinting WORK1, distance 128
7. Prefetching P, 128 bytes ahead
8. Software pipelining across 2 iterations; unrolling loop 2 times;
   steady state estimated 17 cycles; 0 prefetch iterations
9. Prefetching P, 192 bytes ahead
10. Write-hinting WORK2, distance 128
11. Unrolling loop 7 times
12. Prefetching Q, 192 bytes ahead
```

2.7.4. Storage Map Section

The storage map section of the compiler output listing is printed after each program unit, or library module. It is not generated when a fatal compilation error is encountered.

The storage map section summarizes information in the following categories:

- Program sections

The program section summary describes each program section (PSECT) generated by the compiler. The descriptions include:

- PSECT number (used by most of the other summaries)
- Name
- Size in bytes
- Attributes

PSECT usage and attributes are described in Section 13.1.

- Total memory allocated

  Following the program sections, the compiler prints the total memory allocated for all program sections compiled in the following form:

  `Total Space Allocated` *nnn*

- Entry points

  The entry point summary lists all entry points and their addresses. If the program unit is a function, the declared data type of the entry point is also included.

- Variables

  The variable summary lists all simple variables, with the data type and address of each. If the variable is removed as a result of optimization, a double asterisk (**) appears in place of the address.

- Records

  The record summary lists all record variables and derived types. It shows the address, the structure that defines the fields of the individual records or types, and the total size of each record or type.

- Arrays

  The array summary is similar to the variable summary. In addition to data type and address, the array summary gives the total size and dimensions of the array. If the array is an adjustable array or assumed-size array, its size is shown as a double asterisk (**), and each adjustable dimension bound is shown as a single asterisk (*).

- Record Arrays

  The record array summary is similar to the record summary. The record array summary gives the dimensions of the record or derived-type array in addition to address, defining structure, and total size. If the record or derived-type array is an adjustable array or assumed-size array, its size is shown as a double asterisk (**), and each adjustable dimension bound is shown as a single asterisk (*).

- Labels

  The label summary lists all user-defined statement labels. FORMAT statement labels are suffixed with an apostrophe (’). If the label address field contains a double asterisk (**), the label was not used or referred to by the compiled code.

- Functions and subroutines

The functions and subroutines summary lists all external routine references made by the source program. This summary does not include references to routines that are dummy arguments, because the actual function or subroutine name is supplied by the calling program.

A heading for an information category is printed in the listing only when entries are generated for that category.

Example 2.6 shows an example of a storage map section.

Example 2.6. Sample Storage Map Section

```
RELAX2  Source Listing  19-DEC-2004 16:12:46 HP Fortran v8.x-xxxx          Page 1
                        12-DEC-2004 10:41:04  F90$DISK:[TUCKER]LIST_EX.F90;3
PROGRAM SECTIONS

    Name     Bytes   Attributes

  1 $CODE$                    1020   PIC CON REL LCL   SHR   EXE NORD NOWRT OCTA
  2 $LINK$                      44 NOPIC CON REL LCL NOSHR NOEXE   RD NOWRT OCTA
  3 $BLANK                    9600 NOPIC OVR REL GBL NOSHR NOEXE   RD   WRT OCTA

    Total Space Allocated               10664


ENTRY POINTS

 Address      Name

  1-00000000  RELAX2


VARIABLES

 Address  Type  Name  Address  Type Name Address Type  Name  Address Type   Name  Address
 Type  Name

      **  L*4   DONE       **  R*4  EPS       ** I*4   I          ** I*4    J          **
 R*4   XNEW


ARRAYS

 Address      Type  Name        Bytes  Dimensions

  3-00000000  R*4   X            9600  (40, 60)


LABELS

 Address      Label

  1-00000040  10
```

As shown in Example 2.6, a section size is specified as a number of bytes, expressed in decimal. A data address is specified as an offset from the start of a program section, expressed in hexadecimal.

2.7.5. Compilation Summary Section

The final entries on the compiler output listing are the compiler qualifiers and compiler statistics.

The body of the compilation summary contains information about OPTIONS statement qualifiers (if any), FORTRAN command line qualifiers, and compilation statistics.

“Compilation Statistics” shows the machine resources used by the compiler.

Example 2.7 shows a sample compilation summary.

Example 2.7. Sample Compilation Summary

```
COMMAND QUALIFIERS

  /ADDRESSING_MODEL=NORMAL
  /ALIGNMENT=(COMMONS=(NONATURAL,PACKED,NOSTANDARD,NOMULTILANGUAGE),
                      RECORDS=NATURAL,NOSEQUENCE)
  /ANNOTATIONS=(NOCODE,NODETAIL,NOFEEDBACK,NOINLINING,NOLINKAGES,
                NOLOOP_TRANSFORMS,NOLOOP_UNROLLING,NOPREFETCHING,
                NOSHRINKWRAPPING,NOSOFTWARE_PIPELINING,NOTAIL_CALLS)
  /ARCHITECTURE=GENERIC
  /ASSUME=(ACCURACY_SENSITIVE,ALTPARAM,NOBUFFERED_IO,NOBYTERECL,
           NODUMMY_ALIASES,NOF77RTL,NOFP_CONSTANT,NOINT_CONSTANT,
           NOMINUS0,PROTECT_CONSTANTS,NOSOURCE_INCLUDE,NOUNDERSCORE)
  /NOAUTOMATIC
  /NOBY_REF_CALL
  /CCDEFAULT=DEFAULT
  /CHECK=(NOARG_TEMP_CREATED,NOBOUNDS,FORMAT,NOFP_EXCEPTIONS,
          OUTPUT_CONVERSION,NOOVERFLOW,POWER,NOUNDERFLOW,
          NOARG_INFO,NOFP_MODE)
  /CONVERT=NATIVE
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /NODEFINE
  /DOUBLE_SIZE=64
  /NOD_LINES
  /ERROR_LIMIT=30
  /NOEXTEND_SOURCE
  /F77
  /NOFAST
  /FLOAT=IEEE_FLOAT
  /GRANULARITY=QUADWORD
  /IEEE_MODE=DENORM_RESULTS
  /INTEGER_SIZE=32
  /MACHINE_CODE
  /MATH_LIBRARY=ACCURATE
  /NOMODULE
  /NAMES=UPPERCASE
  /OPTIMIZE=(INLINE=SPEED,LEVEL=4,NOLOOPS,PIPELINE,TUNE=GENERIC,UNROLL=0)
  /NOPAD_SOURCE
  /REAL_SIZE=32
  /NORECURSIVE
  /REENTRANCY=NONE
  /ROUNDING_MODE=NEAREST
  /NOSEPARATE_COMPILATION
  /SEVERITY=(WARNING=WARNING)
  /SHOW=(NODICTIONARY,NOINCLUDE,MAP,NOPREPROCESSOR)
  /SOURCE_FORM=FREE
  /STANDARD=NONE
  /NOSYNCHRONOUS_EXCEPTIONS
  /NOSYNTAX_ONLY
  /NOTIE
  /VMS
  /WARNINGS=(ALIGNMENT,NOARGUMENT_CHECKING,NODECLARATIONS,GENERAL,
             GRANULARITY,NOIGNORE_LOC,NOTRUNCATED_SOURCE,UNCALLED,
             UNINITIALIZED,NOUNUSED,USAGE)
  /NOANALYSIS_DATA
```

```
  /NODIAGNOSTICS
  /INCLUDE=FORT$INCLUDE:
  /LIST=GEMI64$DKA100:[JBISHOP.BUG]SIMPLE.LIS;1
  /OBJECT=GEMI64$DKA100:[JBISHOP.BUG]SIMPLE.OBJ;1
  /NOLIBRARY

COMPILER: HP Fortran V8.0-xxxxx-xxxxx

COMPILATION STATISTICS

  CPU time:          0.13 seconds
  Elapsed time:      1.06 seconds
  Pagefaults:         720
  I/O Count:           28
```

Chapter 3. Linking and Running VSI Fortran Programs

This chapter describes:

- Section 3.1: Linker Overview
- Section 3.2: LINK Command Qualifiers and Messages
- Section 3.3: Running VSI Fortran Programs
- Section 3.4: Symbol Table and Traceback Information: Locating Run-Time Errors

3.1. Linker Overview

The primary functions of the linker are to allocate virtual memory within the executable image, to resolve symbolic references among object modules being linked, to assign values to relocatable global symbols, to perform relocation, and to perform any requested special-purpose tasks. The linker creates an executable or shareable image that you can run on an OpenVMS system.

For any VSI Fortran program unit, the object file generated by the compiler may contain calls and references to VSI Fortran run-time procedures, which the linker locates automatically in the default system object libraries, such as IMAGELIB.OLB.

For More Information:

On linker capabilities, default system object libraries, and detailed descriptions of LINK command qualifiers and options, see the *VSI OpenVMS Linker Utility Manual*.

3.2. LINK Command Qualifiers and Messages

The LINK command initiates the linking of the object file. The command has the following form:

```
LINK[/command-qualifiers] file-spec[/file-qualifiers]...
```

/command-qualifiers

Specify output file options.

file-spec

Specifies the input object file to be linked.

/file-qualifiers

Specify input file options.

In interactive mode, you can issue the LINK command without a file specification. The system then requests the file specifications with the following prompt:

```
_File:
```

You can enter multiple file specifications by separating them with commas (,) or plus signs (+). When used with the LINK command, the comma has the same effect as the plus sign; a single executable

image is created from the input files specified. If no output file is specified, the linker produces an executable image with the same name as that of the first object file and with a file type of EXE.

Table 3.1 lists the linker qualifiers of particular interest to VSI Fortran users.

Table 3.1. LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output file, define a file specification, and specify whether to create an executable or shareable image.	/EXECUTABLE[= *file-spec*] /SHAREABLE[= *file-spec*] (see Section 3.2.1.1)	/EXECUTABLE= *name*.EXE where *name* is the name of the first input file. /NOSHAREABLE
Allow the executable image to interoperate with nonnative shared libraries.	/[NO]NATIVE_ONLY (see Section 3.2.1.2)	/NATIVE_ONLY (see Section 3.2.1)
Request and specify the contents of an image map (memory allocation) listing.	/BRIEF /[NO]CROSS_REFERENCE /FULL /[NO]MAP (see Section 3.2.1.3)	/NOCROSS_REFERENCE /NOMAP (interactive) /MAP= *name*.MAP (batch) where *name* is the name of the first input file.
Specify the amount of debugging information.	/[NO]DEBUG /[NO]TRACEBACK (see Section 3.2.2)	/NODEBUG /TRACEBACK
Indicate the type of input files and request the inclusion of only certain object modules.	/INCLUDE=(module-name...) /LIBRARY /SELECTIVE_SEARCH as positional qualifier: /SHAREABLE (see Section 3.2.3)	Not applicable
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB /[NO]SYSSHR /[NO]USERLIBRARY[=table]	/SYSLIB /SYSSHR /USERLIBRARY=ALL
Indicate that an input file is a linker options file.	/OPTIONS (see Section 3.2.5)	Not applicable
Request a symbol table.	/[NO]SYMBOL_TABLE[= *file-spec*] (see Section 3.2.4)	/NOSYMBOL_TABLE

3.2.1. Linker Output File Qualifiers

You can use qualifiers on the LINK command line to control the output produced by the linker. You can also specify whether traceback and debugging information is included. (The /DEBUG and /TRACEBACK qualifiers are described in Section 3.2.2).

Linker output consists of an image file and, optionally, a map file. The qualifiers that control image and map files are described under the headings that follow.

3.2.1.1. Image File Qualifiers

The LINK command /[NO]EXECUTABLE and /[NO]SHAREABLE qualifiers control whether the linker creates an executable image, a shareable image, or no image file.

- /EXECUTABLE

  If you omit /EXECUTABLE and /SHAREABLE, the default is /EXECUTABLE, and the linker produces an executable image.

  To suppress production of an image, specify /NOEXECUTABLE. For example, in the following command, the file CIRCLE.OBJ is linked, but no image is generated:

  ```
  $ LINK/NOEXECUTABLE CIRCLE
  ```

  The /NOEXECUTABLE qualifier is useful if you want to verify the results of linking an object file without actually producing the image.

  To designate a file specification for an executable image, use the /EXECUTABLE qualifier in the following form:

  ```
  /EXECUTABLE=file-spec
  ```

  For example, the following command links the file CIRCLE.OBJ. The linker generates an executable image named TEST.EXE:

  ```
  $ LINK/EXECUTABLE=TEST CIRCLE
  ```

- /SHAREABLE

  A shareable image has all of its internal references resolved, but must be linked with one or more object modules to produce an executable image. A shareable image file, for example, can contain a library of routines. To create a shareable image, specify the /SHAREABLE qualifier, as shown in the following example:

  ```
  $ LINK/SHAREABLE CIRCLE
  ```

  To include a shareable image as input to the linker, you can either use a linker options file or insert the shareable image into a shareable-image library and specify the library as input to the LINK command. By default, the linker automatically searches the system-supplied shareable-image library SYS$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command line.

  If you specify (or default to) /NOSHAREABLE, the linker creates an executable image, which cannot be linked with other images.

  You can also use the /SHAREABLE qualifier in an options file as a positional qualifier (following a file specification).

  To create a shareable-image library, use the LIBRARIAN command with the /CREATE, /INSERT, and /SHARE qualifiers.

For More Information:

- On creating and populating libraries, see the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

- On libraries searched by the linker, see the *VSI OpenVMS Linker Utility Manual*.

3.2.1.2. /NATIVE_ONLY Qualifier

The FORTRAN command /[NO]TIE qualifier and the LINK command /[NO]NATIVE_ONLY qualifier together control whether procedure signature block (PSB) information is placed in the executable image.

The image activator uses the procedure signature block (PSB) information in the executable image to build jackets that allow calls (interoperability) with translated VAX shareable images on Alpha systems, and translated Alpha images on I64 systems. To allow interoperability with translated shareable images, you must:

- Specify the /TIE qualifier on the FORTRAN command line. This places the PSB information into the object module.
- Specify /NONATIVE_ONLY on the LINK command line. This propagates the PSB information in the object module into the executable image, so the image activator can use it with translated images.

The following command combination creates an executable image for translated image interoperability:

```
$ FORTRAN/TIE file
$ LINK/NONATIVE_ONLY file
```

Unless you need the PSB information for calls to translated shareable images, avoid placing PSB information in the executable image to minimize its size for performance reasons.

The default for the FORTRAN command, /NOTIE, results in the PSB information not being placed in the object module. The default for the LINK command, /NATIVE_ONLY, results in the linker not passing any PSB information present in the object module into the executable image.

For example, the following command combination creates a executable image for native use only (no PSB information in the executable image):

```
$ FORTRAN/TIE file
$ LINK file
```

The FORTRAN /TIE qualifier places the PSB information in the object module. By default, the linker does not propagate the PSB information, if found in the object module, to the executable image (default is /NATIVE_ONLY). The same object module might be retained and used in a different application that requires interoperability.

For More Information:

On interoperability, see Section B.6 and the manuals *Migrating an Application from OpenVMS VAX to OpenVMS Alpha* and Porting Applications from *VSI OpenVMS Alpha to VSI OpenVMS Industry Standard 64 for Integrity Servers*.

3.2.1.3. Map File Qualifiers

The LINK command /MAP qualifier and its associated qualifiers indicate whether an image map file is to be generated and, if so, the amount of information to be included in the image map file.

The map qualifiers are specified as follows:

/[NO]MAP*[=file-spec]* { /BRIEF | /CROSS_REFERENCE | /FULL }

In interactive mode, the default is to suppress the map; in batch mode, the default is to generate the map.

If you do not include a file specification with the /MAP qualifier, the map file has the name of the first input file and a file type of MAP. It is stored on the default device in the default directory.

The /BRIEF and /FULL qualifiers define the amount of information included in the map file. They function as follows:

- /BRIEF produces a summary of the image's characteristics and a list of contributing object modules.
- /FULL produces a summary of the image's characteristics and a list of contributing object modules (as produced by /BRIEF). It also produces a list, in symbol-name order, of global symbols and values (program, subroutine, and common block names, and names declared EXTERNAL) and a summary of characteristics of image sections in the linked image.

If neither /BRIEF nor /FULL is specified, the map file, by default, contains a summary of the image's characteristics, a list of contributing object modules (as produced by /BRIEF), and a list of global symbols and values, in symbol-name order.

You can use the /CROSS_REFERENCE qualifier with either the default or /FULL map qualifiers to request cross-reference information for global symbols. This cross-reference information indicates the object modules that define or refer to global symbols encountered during linking. The default is /NOCROSS_REFERENCE.

For More Information:

On image maps, see the *VSI OpenVMS Linker Utility Manual*.

3.2.2. /DEBUG and /TRACEBACK Qualifiers

The /DEBUG qualifier includes the debugger and places local symbol information (contained in the object modules) in the executable image. The default is /NODEBUG.

When you use the /TRACEBACK qualifier, run-time error messages are accompanied by a symbolic traceback. This shows the sequence of calls that transferred control to the program unit in which the error occurred. If you specify /NOTRACEBACK, this information is not produced. The default is /TRACEBACK.

If you specify /DEBUG, the traceback capability is automatically included, and the /TRACEBACK qualifier is ignored.

For More Information:

- On the OpenVMS debugger, see Chapter 4.
- On a sample traceback list, see Section 3.4.1.

3.2.3. Linker Input File Qualifiers

The LINK command input file qualifier /LIBRARY identifies the input file as a library file. The LINK command /INCLUDE qualifier specifies the names of one or more object modules in a library to be used as input files. The /SELECTIVE_SEARCH qualifier indicates that only symbols referenced from previous input files be taken from the specified file's list of global symbols, rather than all of the input file's global symbols.

To specify a shareable image as an input file, place the /SHAREABLE positional qualifier after the file specification in a link options file. In contrast to the /SHAREABLE qualifier you use on a LINK command line to create a shareable image, the /SHAREABLE positional qualifier in a link options file specifies a shareable image input file.

Input files can be object files, previously linked shareable files, or library files.

- /LIBRARY

  The /LIBRARY qualifier specifies that the input file is an object-module or shareable-image library that is to be searched to resolve undefined symbols referenced in other input object modules. The default file type is OLB.

  The /LIBRARY qualifier has the following form:

  file.type/LIBRARY

- /INCLUDE

  The /INCLUDE qualifier specifies that the input file is an object-module or shareable-image library and that the object modules named are the only object modules in the library to be explicitly included as input. In the case of shareable-image libraries, the object module is the shareable-image name.

  The /INCLUDE qualifier has the following form:

  file.type/INCLUDE=*module-name*

  At least one object module name is required. To specify more than one, enclose the object module names in parentheses and separate the names with commas.

  The default file type is OLB. The /LIBRARY qualifier can also be used, with the same file specification, to indicate that the same library is to be searched for unresolved references.

- /SELECTIVE_SEARCH

  The /SELECTIVE_SEARCH qualifier specifies that the linker only include those global symbols already referenced from previous input files, instead of all the global symbols defined by the specified input file. The /SELECTIVE_SEARCH positional qualifier has the following form:

  file.OBJ/SELECTIVE_SEARCH

- /SHAREABLE

  When used as a positional qualifier in a linker options file, the /SHAREABLE qualifier specifies that the input file it follows is a shareable image. The /SHAREABLE positional qualifier has the following form:

  file.EXE/SHAREABLE

For More Information:

- On defining DCL commands, creating libraries, and related topics, see the *VSI OpenVMS Command Definition, Librarian, and Message Utilities*.
- On linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VSI OpenVMS Linker Utility Manual*.

3.2.4. Linker Symbol Table Qualifier

The /SYMBOL_TABLE qualifier requests that the symbol table be written to a file.

The qualifier has the following form:

`/SYMBOL_TABLE=`*`file-spec`*

If you omit the file name, the linker uses a default file type of STB.

3.2.5. Linker Options File Qualifier

The /OPTIONS qualifier identifies a file as a linker options file, which contains additional information about the current link operation.

The qualifier has the following form:

`file-spec``/OPTIONS`

The file specification is usually a file with default file type of OPT.

You can use an options file to specify additional information to the linker, including a shared image as input to the linker, declare universal symbols, and other features.

For more information on using an OPTIONS file, see the *VSI OpenVMS Linker Utility Manual*.

3.2.6. Other Linker Qualifiers

This chapter discusses the more frequently used linker qualifiers. Other linker qualifiers specify additional libraries to be searched or to prevent default libraries from being searched, control image section characteristics and placement, creation of a global symbol table, and other linker operations.

VSI Fortran programs should not be linked with certain linker qualifiers, such as /SYSTEM and /PROTECT.

3.2.7. Linker Messages

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors or fatal conditions occur (severities E or F), the linker does not produce an image file.

Linker messages are descriptive; you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking are as follows:

- An object module has compilation errors. This error occurs when you attempt to link a object module that had warnings or errors during compilation. Although you can usually link compiled object modules for which the compiler generated messages, you should verify that the object modules actually produce the expected results during program execution.
- The object modules that are being linked define more than one transfer address. The linker generates a warning if more than one main program has been defined. This can occur, for example, when an extra END statement exists in the program. In this case, the image file created by the linker can be run; the entry point to which control is transferred is the first one that the linker finds.

- A reference to a symbol name remains unresolved. This error occurs when you omit required object module or library names on the LINK command line, and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link object modules, you can often correct it simply by reentering the command string and specifying the correct object files or libraries.

You can use the OpenVMS HELP/MESSAGE facility to view error recovery descriptions associated with messages from any OpenVMS facility.

For More Information:

On using the HELP/MESSAGE command, see the *OpenVMS System Messages: Companion Guide for Help Message Users*.

3.3. Running VSI Fortran Programs

This section describes the following considerations for executing VSI Fortran programs on an OpenVMS operating system:

- Using the RUN command to execute programs interactively
- Passing status values to the command interpreter

3.3.1. RUN Command

The RUN command initiates execution of a program. The command has the following form:

```
RUN[/[NO]DEBUG] file-spec
```

You must specify the file name. If you omit optional elements of the file specification, the system automatically provides a default value. The default file type is EXE.

The /DEBUG qualifier allows you to use the debugger, even if you omitted this qualifier on the FORTRAN and LINK command lines. Refer to Section 3.4 for details.

3.3.2. System Processing at Image Exit

When the main program executes an END statement, or when any program unit in the program executes a STOP statement, the image is terminated. With the OpenVMS operating system, the termination of an image, or image exit, causes the system to perform a variety of clean-up operations during which open files are closed, system resources are freed, and so on.

3.3.3. Interrupting a Program

When you execute the RUN command interactively, you cannot execute any other program images or DCL commands until the current image completes. However, if your program is not performing as expected – if, for instance, you have reason to believe it is in an endless loop – you can interrupt it by using the **Ctrl/Y** key sequence. (You can also use the **Ctrl/C** key sequence, unless your program takes specific action in response to **Ctrl/C**). For example:

```
$ RUN APPLIC
```

```
Ctrl/Y
$
```

This command interrupts the program APPLIC. After you have interrupted a program, you can terminate it by entering a DCL command that causes another image to be executed or by entering the DCL commands EXIT or STOP.

Following a **Ctrl/Y** interruption, you can also force an entry to the debugger by entering the DEBUG command.

Some of the other DCL commands you can enter have no direct effect on the image. After using them, you can resume the execution of the image with the DCL command CONTINUE. For example:

```
$ RUN APPLIC
Ctrl/Y
$ SHOW LOGICAL INFILE
%SHOW-S-NOTRAN, no translation for logical name INFILE
$ DEFINE INFILE $1$DUA1:[TESTFILES]JANUARY.DAT
$ CONTINUE
```

As noted previously, you can use **Ctrl/C** to interrupt your program; in most cases, the effect of **Ctrl/C** and **Ctrl/Y** is the same. However, some programs (including programs you may write) establish particular actions to take to respond to **Ctrl/C**. If a program has no **Ctrl/C** handling routine, then **Ctrl/C** is the same as **Ctrl/Y**.

For More Information:

- On defining DCL commands, creating libraries, and related topics, see the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.
- On commands you can enter following a **Ctrl/Y** interruption without affecting the current image, see the *VMS DCL Concepts Manual*.
- On an example of handling **Ctrl/C** interrupts, see Section F.2.

3.3.4. Returning Status Values to the Command Interpreter

If you run your program as part of a command procedure, it is often useful to return a status value to the command procedure. This indicates whether the program actually executed properly.

To return such a status value, call the EXIT system subroutine rather than terminating execution with a STOP, RETURN, or END statement. The EXIT subroutine can be called from any executable program unit. It terminates your program and returns the value of the argument as the return status value of the program.

When the command interpreter receives a status value from a terminating program, it attempts to locate a corresponding message in a system message file or a user-defined message file. Every message that can be issued by a system program, command, or component, has a unique 32-bit numeric value associated with it. These 32-bit numeric values are called condition symbols. Condition symbols are described in Section 14.4.3.

The command interpreter does not display messages on completion of a program under the following circumstances:

- The EXIT argument specifies the value 1, corresponding to SUCCESS.
- The program does not return a value. If the program terminates with a RETURN, STOP, or END statement, a value of 1 is always returned and no message is displayed.

For More Information:

On the EXIT subroutine, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

3.4. Symbol Table and Traceback Information: Locating Run-Time Errors

Both the compiler and the OpenVMS Run-Time Library include facilities for detecting and reporting errors. You can use the OpenVMS Debugger and the traceback facility to help you locate errors that occur during program execution.

3.4.1. Effects of Error-Related Command Qualifiers

At each step in compiling, linking, and executing your program, you can specify command qualifiers that affect how errors are processed.

- At compile time, you can specify the /DEBUG qualifier on the FORTRAN command line to ensure that symbolic information is created for use by the debugger.
- At link time, you can also specify the /DEBUG qualifier on the LINK command line to make the symbolic information available to the debugger.
- At run time, you can specify the /DEBUG qualifier on the RUN command line to invoke the debugger.

Table 3.2 summarizes the /DEBUG and /TRACEBACK qualifiers.

Table 3.2. /DEBUG and /TRACEBACK Qualifiers

Command	Qualifier	Effect
FORTRAN	/DEBUG	The VSI Fortran compiler creates symbolic data needed by the debugger. **Default**: /DEBUG=(NOSYMBOLS,TRACEBACK)
LINK	/DEBUG	Symbolic data created by the VSI Fortran compiler is passed to the debugger. **Default**: /NODEBUG
	/TRACEBACK	Traceback information is passed to the debugger. Traceback will be produced. **Default**: /TRACEBACK
RUN	/DEBUG	Invokes the debugger. The DBG> prompt will be displayed. Not needed if LINK/DEBUG was specified.

Command	Qualifier	Effect
	/NODEBUG	If /DEBUG was specified in the LINK command line, RUN/NODEBUG starts program execution without first invoking the debugger.

If an exception occurs and these qualifiers are not specified at any point in the compile-link-execute sequence, a traceback list is generated by default.

To perform symbolic debugging, you must use the /DEBUG qualifier with both the FORTRAN and LINK command lines; you do not need to specify it with the RUN command. If /DEBUG is omitted from either the FORTRAN or LINK command lines, you can still use it with the RUN command to invoke the debugger. However, any debugging you perform must then be done by specifying virtual addresses rather than symbolic names.

If you linked your program with the debugger, but wish to execute the program without debugger intervention, specify the following command:

```
$ RUN/NODEBUG program-name
```

If you specify LINK/NOTRACEBACK, you receive no traceback if there are errors.

3.4.2. Sample Source Program and Traceback

Example 3.1 shows a sample source program and a traceback.

Example 3.1. Sample VSI Fortran Program

```
    PROGRAM TRACE_TEST

       REAL ST
       ST = 2.4E20   ! Constant inside range for REAL*4 formats

       CALL SUB1(ST)

    END PROGRAM TRACE_TEST

    SUBROUTINE SUB1(RV)
       REAL RV,RES
       RV = RV * RV  ! Generates overflow value
       RES=LOG(RV)   ! Uses + Infinity value for IEEE floating-point data

       RETURN
    END SUBROUTINE SUB1
```

The program (TRACEBK.F90) shown in Example 3.1 is compiled, linked, and run to generate the traceback information shown after the RUN command:

```
$ FORTRAN/NOOPTIMIZ/DEBUG=TRACEBACK/SYNCHRONOUS_EXCEPTIONS/
                      FLOAT=IEEE_FLOAT TRACEBK.F90
$ LINK TRACEBK
$ RUN  TRACEBK
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000000,
Fmask=00000001, summary=08, PC=0002007C, PS=0000001B
-SYSTEM-F-FLTOVF, arithmetic trap,floating overflow at
 PC=0002007C,PS=0000001B
%TRACE-F-TRACEBACK, symbolic stack dump follows
```

```
 Image Name   Module Name     Routine Name Line Number rel PC      abs PC
 LINK_TRACEBA TRACE_TEST      SUB1                 10 0000007C    0002007C
 LINK_TRACEBA TRACE_TEST      TRACE_TEST            6 00000030    00020030
                                                    0 88EEFA50    88EEFA50
```

On the FORTRAN command line, the following qualifiers are used:

- /NOOPTIMIZE prevents optimizations that might prevent accurate reporting of information.
- /DEBUG=TRACEBACK, the default, ensures there is sufficient traceback information in the object file.
- /SYNCHRONOUS_EXCEPTIONS (Alpha only) ensures that exceptions reporting will be precise (reported at the statement line causing the error).
- /FLOAT=IEEE_FLOAT requests use of IEEE floating-point data types in memory.

When an error condition is detected, you receive the appropriate message, followed by the traceback information. The Run-Time Library displays a message indicating the nature of the error and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls.

Values can be produced for relative and absolute PC, with no corresponding values for routine name and line. These PC values reflect procedure calls internal to the Run-Time Library.

Of particular interest are the values listed under “Routine Name” and “Line Number”:

- The names under “Routine Name” show which routine or subprogram called the Run-Time Library, which subsequently reported the error.
- The value given for “Line Number” corresponds to the compiler-generated line number in the source program listing.

With this information, you can usually isolate the error in a short time.

If you specify either LINK/DEBUG or RUN/DEBUG, the debugger assumes control of execution and you do not receive a traceback list if an error occurs. To display traceback information, you can use the debugger command SHOW CALLS.

You should use the /NOOPTIMIZE qualifier on the FORTRAN command line whenever you use the debugger.

For More Information:

- On the /OPTIMIZE qualifier, see Section 2.3.14.
- On VSI Fortran run-time errors, see Chapter 7.

Chapter 4. Using the OpenVMS Debugger

This chapter describes:

- Section 4.1: Debugger Overview
- Section 4.2: Getting Started with the Debugger
- Section 4.3: Sample Debugging Session
- Section 4.4: Displaying VSI Fortran Variables
- Section 4.5: Debugger Command Summary
- Section 4.6: Locating an Exception
- Section 4.7: Locating Unaligned Data

4.1. Debugger Overview

A debugger is a tool that helps you locate run-time errors quickly. It is used with a program that has already been compiled and linked successfully, but does not run correctly. For example, the output may be obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively so you can locate the point at which the program stopped working correctly.

The OpenVMS Debugger is a **symbolic debugger**, which means that you can refer to program locations by the symbols (names) you used for those locations in your program—the names of variables, subroutines, labels, and so on. You do not need to use virtual addresses to refer to memory locations.

By issuing debugger commands at your terminal, you can perform the following operations:

- Start, stop, and resume the program's execution
- Trace the execution path of the program
- Monitor selected locations, variables, or events
- Examine and modify the contents of variables, or force events to occur
- Test the effect of some program modifications without having to edit, recompile, and relink the program

Such techniques allow you to isolate an error in your code much more quickly than you could without the debugger.

Once you have found the error in the program, you can then edit the source code and compile, link, and run the corrected version.

4.2. Getting Started with the Debugger

This section explains how to use the debugger with VSI Fortran programs. The section focuses on basic debugger functions, to get you started quickly. It also provides any debugger information that is specific to VSI Fortran.

For More Information:

- About the OpenVMS Debugger (not specific to the VSI Fortran language), see the *VSI OpenVMS Debugger Manual*.
- On OpenVMS Debugger commands, use online HELP during debugging sessions.

4.2.1. Compiling and Linking a Program to Prepare for Debugging

Before you can use the debugger, you must compile and link your program. The following example shows how to compile and link a VSI Fortran program (consisting of a single compilation unit in the file INVENTORY.F90) prior to using the debugger.

```
$ FORTRAN/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the FORTRAN command line causes the compiler to write the debug symbol records associated with INVENTORY.F90 into the object module INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY.F90 in debugger commands. (If your program has several compilation units, each of the program units that you want to debug must be compiled with the /DEBUG qualifier).

Use the /NOOPTIMIZE qualifier when you compile a program in preparation for debugging. Otherwise, the object code is optimized (to reduce the size of the program and make it run faster), so that the symbolic evaluation of some program locations may be inconsistent with what you might expect from viewing the source code. For example, a variable in an optimized program may not be available. (After debugging the program, recompile it without the /NOOPTIMIZE qualifier.) For a description of the various optimizations performed by the compiler, see Chapter 5.

The /DEBUG qualifier on the LINK command line causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. This qualifier also causes the OpenVMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify the other modules on the LINK command line).

For More Information:

On the effects of specifying the /DEBUG qualifier on the FORTRAN, LINK, and RUN command lines, see Section 2.3.14 and Section 3.4.

4.2.2. Establishing the Debugging Configuration and Interface

Before invoking the debugger, check that the debugging configuration is appropriate for the kind of program you want to debug.

The configuration depends on the current value of the logical name DBG$PROCESS. Before invoking the debugger, issue the DCL command SHOW LOGICAL DBG$PROCESS to determine the current definition of DBG$PROCESS.

The default configuration is appropriate for almost all programs. To request the default debugging configuration. the logical name DBG$PROCESS is undefined or has the value DEFAULT. For example, the following command shows when DBG$PROCESS is undefined:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

To define DBG$PROCESS to have a value of DEFAULT, enter:

```
$ DEFINE DBG$PROCESS DEFAULT
```

To remove (deassign) a logical name definition, use the DEASSIGN command.

If the DECwindows Motif product is installed and running on your workstation, by default the OpenVMS Debugger uses the DECwindows Motif interface. To use the character cell interface on a DECwindows system, define the logical:

```
$ DEFINE/PROCESS DBG$DECW$DISPLAY " "
```

To define this logical name for multiple users, use other logical name tables.

To enable use of the DECwindows interface, deassign the logical:

```
$ DEASSIGN/PROCESS DBG$DECW$DISPLAY
```

The DECwindows interface provides a main window in which portions are updated as the program executes, including the source code, entered commands, and debugger messages. This interface provides pull-down menus and uses the kept debugger (equivalent of DEBUG/KEEP).

The examples in this chapter show the command line (character cell) interface to the OpenVMS Debugger.

The character cell interface to the OpenVMS Debugger provides the following debugging interfaces:

- Screen mode
- Line-oriented mode

Screen mode is activated by pressed PF3 on the keypad (or enter the command SET MODE SCREEN). Screen mode allows the debugger character cell interface to simultaneously display separate groups of data similar to the DECwindows interface. For example, your screen might show the source code (SRC), debugger output (OUT), and debugger command input (PROMPT) displays.

While in screen mode, use the SHOW DISPLAY command to view the predefined displays and the DISPLAY command to define a new display. To view the keypad definitions in screen mode, press PF2 on the keypad.

To leave screen mode and resume line-oriented mode, press PF1 PF3 (or enter the command SET SCREEN NOSCREEN).

For More Information:

On the DECwindows Motif debugger interface (including a source browser) and screen mode, see the *VSI OpenVMS Debugger Manual*.

4.2.3. Invoking the Debugger

After you compile and link your program and establish the appropriate debugging configuration, you can then invoke the debugger. To do so, enter the DCL command RUN, specifying the executable image of your program as the parameter. For example, enter the following command to debug the program INVENTORY:

```
$ RUN INVENTORY
```

```
          OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx  %DEBUG-I-INITIAL,
 language is FORTRAN, module set to INVENTORY
DBG> GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXAMINE N
INVENTORY\N: 4
DBG> EXIT
$
```

The diagnostic message that is displayed at the debugger startup indicates that this debugging session is initialized for a VSI Fortran program and that the name of the main program unit is INVENTORY. In the initial "%DEBUG-I-INITIAL" message, the OpenVMS Debugger term "module" is equivalent to a VSI Fortran "procedure."

When some qualifiers are used to compile (/WARNINGS=ALIGNMENT or most /CHECK keywords), the debugger does not start up in the main program. When this happens, enter GO once to get to the beginning of the main program.

The DBG> prompt indicates that you can now enter debugger commands. At this point, if you enter the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input or an error occurs).

You can specify the DEBUG command /KEEP qualifier to use the kept debugger. The kept debugger allows you to run one (or more) programs with the RUN command, rerun the last program run with a RERUN command, and connect and disconnect to a running process. For example:

```
$ DEBUG /KEEP
          OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx
DBG> RUN SQUARES
%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES
DBG> GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> RERUN
DBG> STEP
 stepped to SQUARES\%LINE 4
 4:       OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
DBG> EXAMINE N
DEBUGEX$MAIN\N: 0
DBG> GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EX N
DEBUGEX$MAIN\N: 4
DBG> EXIT
```

For more information on using the debugger and invoking the debugger for other display modes, see the *VSI OpenVMS Debugger Manual*.

4.2.4. Debugger Commands Used Often

You can use the following debugger commands when debugging any program:

- To get help on debugger commands, enter the HELP command.
- To control program execution, enter the GO, STEP, CALL, or EXIT commands. To control whether the STEP command steps into or over a routine (or performs other functions), you can use the SET STEP command or qualifiers to the STEP command (such as STEP/OVER).
- To look at the contents of a location, enter the EXAMINE command.

- To modify the contents of a location, enter the DEPOSIT command.
- To view source lines, you usually use the TYPE command.
- You can use the debugger SPAWN command (followed by the desired DCL command) to execute a DCL command. For instance, if you have some mail messages you need to read, enter the following:

```
DBG> SPAWN MAIL
```

For More Information:

- About the types of available debugger commands, see Section 4.5.
- On the available debugger commands, see the *VSI OpenVMS Debugger Manual*.

4.2.5. Debugger Breakpoints, Tracepoints, and Watchpoints

The OpenVMS Debugger supports breakpoints, tracepoints, and watchpoints to help you find out what happens at critical points in your program.

Set a **breakpoint** if you want the debugger to stop program execution at a certain point in your program (routine, line number, and so on). After you set a breakpoint and begin program execution, execution stops at the breakpoint, allowing you to look at the contents of program variables to see if they contain the correct values.

Use the following commands to control breakpoints:

- To set a breakpoint, use one of the forms of the SET BREAK commands. You can also specify an action for the breakpoint, such as displaying the value of a variable.
- To view the currently set breakpoints, use the SHOW BREAK command.
- To cancel a breakpoint, use CANCEL BREAK. You can temporarily deactivate a breakpoint with a DEACTIVATE BREAK command, which you can later activate with an ACTIVATE BREAK command.

Set a **tracepoint** to request that the debugger display messages when certain parts of your program execute. You can also specify an action for the tracepoint, such as displaying the value of a variable. Unlike breakpoints, execution continues past the tracepoint. For example, a tracepoint lets you see how many times a routine gets called.

Use the following commands to control tracepoints:

- To set a tracepoint, use one of the forms of the SET TRACE commands. You can also specify an action for the tracepoint, such as displaying the current value of a variable.
- To view the currently set tracepoints, use a SHOW TRACE command.
- To cancel a tracepoint, use a CANCEL TRACE command. You can temporarily deactivate a tracepoint with a DEACTIVATE TRACE command, which you can later activate with an ACTIVATE TRACE command.

Set a **watchpoint** to request that the debugger stop execution when the values of certain variables (or memory locations) change. A breakpoint stops execution when a certain part of program is reached. In contrast, a watchpoint stops execution when a certain value changes.

The following commands are usually used to control watchpoints:

- To set a watchpoint, use one of the forms of the SET WATCH commands. You can also specify an action for the watchpoint, such as displaying the value of a variable.
- To view the currently set watchpoints, use the SHOW WATCH command.
- To cancel a watchpoint, use CANCEL WATCH command. You can temporarily deactivate a watchpoint with a DEACTIVATE WATCH command, which you can later activate with an ACTIVATE WATCH command.

Before you set a breakpoint, tracepoint, or watchpoint, you can define the scope to be used (by using the SET SCOPE or SET MODULE command, for instance) or you can add a pathname prefix before a symbol name.

4.2.6. Ending a Debugging Session

To end a debugging session and return to the DCL level, enter EXIT or press **Ctrl/Z**:

```
DBG> EXIT
$
```

The following message, displayed during a debugging session, indicates that your program has completed normally:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

To continue debugging after seeing this message, enter EXIT and start a new debugging session with the DCL RUN command.

If you specified the DEBUG command with the /KEEP qualifier when you invoked the debugger, you can run the same program again from within the debugging session (RERUN) and perform other functions.

4.2.7. Notes on Debugger Support for VSI Fortran

In general, the debugger supports the data types and operators of VSI Fortran and the other debugger-supported languages. However, the following are language-specific limitations and differences:

- Although the type codes for unsigned integers (BU, WU, LU) are used internally to describe the LOGICAL data types, the debugger treats LOGICAL variables and values as being signed when used in language expressions.
- The debugger prints the numeric values of LOGICAL variables or expressions instead of .TRUE. or .FALSE. Normally, only the low-order bit of a LOGICAL variable or value is significant (0 is .FALSE. and 1 is .TRUE.). However, VSI Fortran does allow all bits in a LOGICAL value to be manipulated, and LOGICAL values can be used in integer expressions. For this reason, it is sometimes necessary to see the entire integer value of a LOGICAL variable or expression, which is what the debugger shows.
- Fortran modules (defined by a MODULE statement) are treated as debugger modules. To see whether a module is known to the debugger, use the SHOW MODULE command. To load the symbolic information about the Fortran module into the debugger, use a SET MODULE command (see Section 4.4.5).
- The default name for a main program unit is *filename*$MAIN, where *filename* is the name of your source file. If *filename* is larger than 26 characters and a name is not specified in a PROGRAM or

BLOCK DATA statement, the name is truncated to 26 characters and $MAIN is appended to form the program name.

- COMPLEX constants such as (1.0, 2.0) are not supported in debugger expressions.
- Floating-point number representation depends on the specified precision (KIND) and the FORTRAN command /FLOAT qualifier used during compilation:
  - Single-precision numbers of type REAL (KIND=4) and COMPLEX (KIND=4) can be represented by the F_float or IEEE S_float format, depending on the FORTRAN command /FLOAT qualifier.
  - Double-precision numbers of type REAL (KIND=8) and COMPLEX (KIND=8) can be represented by the D_float, G_float, or IEEE T_float format, depending on the FORTRAN command /FLOAT qualifier.
  - Extended-precision numbers of type REAL (KIND=16) are always represented in X_float format (based on IEEE).

For More Information:

- About the supported data types and operators of any of the languages, enter the HELP LANGUAGE command at the DBG> prompt.
- On VSI Fortran little endian data type representation, see Chapter 8.
- On the FORTRAN /FLOAT qualifier, see Section 2.3.22.

4.3. Sample 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. Compiler-assigned line numbers have been added in the example so that you can identify the source lines referenced in the explanatory text.

Example 4.1. Sample Program SQUARES

```
1         PROGRAM SQUARES
2         INTEGER (KIND=4) :: INARR(20), OUTARR(20)
3    ! 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    ! 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   ! 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 (array elements) that follow.
2. Enters a loop in which it copies the square of each nonzero integer into another array OUTARR (lines 9 through 13).
3. Prints the number of nonzero elements in the original sequence and the square of each such element (lines 16 through 21).

When you run SQUARES, it produces the following output, regardless of the number of nonzero elements in the data file:

```
$ RUN SQUARES
Number of nonzero elements is   0
```

The 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 use the debugger to find the error in the program in Example 4.1. Comments keyed to the callouts follow the example.

Example 4.2. Sample Debugging Session Using Program SQUARES

```
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES      ❶
$ LINK/DEBUG SQUARES                    ❷
$ SHOW LOGICAL DBG$PROCESS              ❸
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS

$ RUN SQUARES                           ❹
          OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx
%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES

DBG> STEP 5                             ❺
stepped to SQUARES\%LINE 9
     9:         DO 10 I = 1, N

DBG> EXAMINE N,K                        ❻
SQUARES\N:       4
SQUARES\K:       0

DBG> STEP 2                             ❼
stepped to SQUARES\%LINE 11
    11:                 OUTARR(K) = INARR(I)**2

DBG> EXAMINE I,K                        ❽
SQUARES\I:       1
SQUARES\K:       0

DBG> DEPOSIT K = 1                      ❾
DBG> SET TRACE/SILENT %LINE 11 DO (DEPOSIT K = K + 1)   ❿

DBG> GO                                 ⓫
```

```
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
%DEBUG-I-EXITSTATUS, is 'SYSTEM-S-NORMAL, normal successful completion'

DBG> EXIT                                    ⓬
$ EDIT SQUARES.FOR                           ⓭
.
.
.
10:       IF(INARR(I) .NE. 0) THEN
11:           K = K + 1
12:           OUTARR(K) = INARR(I)**2
13:       ENDIF
.
.
.
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES           ⓮
$ LINK/DEBUG SQUARES
$ RUN SQUARES                                ⓯
          OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx
%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES
DBG> SET BREAK %LINE 12 DO (EXAMINE I,K)     ⓰
DBG> SHOW BREAK
breakpoint at SQUARES\%LINE 12
   do (EXAMINE I,K)

DBG> TYPE 7:14
module SQUARES
     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:              K = K + 1
    12:              OUTARR(K) = INARR(I)**2
    13:           ENDIF
    14:         END DO
DBG> GO                                      ⓱
break at SQUARES\%LINE 12     12:              OUTARR(K) = INARR(I)**2
SQUARES\I:        1
SQUARES\K:        1

DBG> GO
break at SQUARES\%LINE 12     12:              OUTARR(K) = INARR(I)**2
SQUARES\I:        2
SQUARES\K:        2

DBG> GO
break at SQUARES\%LINE 12     12:              OUTARR(K) = INARR(I)**2
SQUARES\I:        3
SQUARES\K:        3
DBG> GO
break at SQUARES\%LINE 12     12:              OUTARR(K) = INARR(I)**2
SQUARES\I:        4
SQUARES\K:        4
DBG> GO
```

```
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
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXIT                                    ⓲
$
```

❶ The /DEBUG qualifier on the DCL FORTRAN command directs the compiler to write the symbol information associated with SQUARES into the object module, SQUARES.OBJ, in addition to the code and data for the program.

The /NOOPTIMIZE qualifier disables optimization by the FORTRAN compiler, to ensure that the executable code matches the source code of the program. Debugging optimized code can be confusing because the contents of some program locations might be inconsistent with what you would expect from viewing the source code.

❷ The /DEBUG qualifier on the DCL LINK command causes the linker to include all symbol information that is contained in SQUARES.OBJ in the executable image.

❸ The SHOW LOGICAL DBG$PROCESS command shows that the logical name DBG$PROCESS is undefined, thus the debugger starts in the default configuration.

❹ The DCL command RUN SQUARES starts the debugger, which displays its banner and the debugger prompt, `DBG>`. You can now enter debugger commands. The informational message identifies the source language of the program (FORTRAN) and the name of the main program unit (SQUARES).

After the RUN SQUARES command, execution is initially paused at the start of the main program unit (line 1 of SQUARES, in this example).

❺ You decide to test the values of variables `N` and `K` after the READ statement has been executed and the value 0 has been assigned to `K`.

The command STEP 5 executes 5 source lines of the program. Execution is now paused at 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.

❻ The command EXAMINE N, K displays the current values of `N` and `K`. Their values are correct at this point in the execution of the program.

❼ The command STEP 2 executes the program into the loop (lines 9 to 11) that copies and squares all nonzero elements of INARR into OUTARR

❽ The command EXAMINE I,K displays the current values of `I` and `K`.

`I` has the expected value, 1. But `K` has the value 0 instead of 1, which is the expected value. To fix this error, `K` should be incremented in the loop just before it is used in line 11.

❾ The DEPOSIT command assigns `K` the value it should have now: 1.

❿ The SET TRACE command is now used to patch the program so that the value of `K` is incremented automatically in the loop. The command sets a tracepoint that triggers every time execution reaches line 11:

- The /SILENT qualifier suppresses the "trace at " message that would otherwise appear each time line 11 is executed.

- The DO clause issues the DEPOSIT K = K + 1 command every time the tracepoint is triggered.

⓫ To test the patch, the GO command starts execution from the current location.

The program output shows that the patched program works properly. The EXITSTATUS message shows that the program executed to completion.

⓬ The EXIT command returns control temporarily to DCL level so that you can correct the source file and recompile and relink the program.

⓭ The EXIT command returns control temporarily to DCL level so that you can correct the source file and recompile and relink the program.

⓮ The revised program is compiled and linked.

⓯ The RUN SQUARES (DCL command) starts the debugger using the revised program so that its correct execution can be verified.

⓰ The SET BREAK command sets a breakpoint that triggers every time line 12 is executed. The DO clause displays the values of `I` and `K` automatically when the breakpoint triggers.

⓱ The GO command starts execution.

At the first breakpoint, the value of `K` is 1, indicating that the program is running correctly so far. Each additional GO command shows the current values of `I` and `K`. After two GO commands, `K` is now 3, as expected. However, `I` is 4, because one of the INARR elements was zero so that lines 11 and 12 were not executed (and `K` was not incremented) for that iteration of the DO loop. This confirms that the program is running correctly.

⓲ The EXIT command ends the debugging session, returning control to DCL level.

4.4. Displaying VSI Fortran Variables

You usually display the values of variables by using the debugger EXAMINE command, which accepts numerous qualifiers.

4.4.1. Accessing VSI Fortran Common Block Variables

To display common block variables, enter the EXAMINE command followed by the variable names that make up the common block. For example:

```
DBG> TYPE 1:8
       1:
       2:  PROGRAM TEST
       3:   INTEGER*4 INT4
       4:   CHARACTER(LEN=1) CHR
       5:   COMMON /COM_STRA/ INT4, CHR
       6:   CHR = 'L'
       7:   INT4 = 0
       8:  END PROGRAM TEST
DBG> STEP 3
stepped to TEST\%LINE 8
     8:  END PROGRAM TEST
DBG> EXAMINE CHR, INT4
TEST\CHR:  'L'
TEST\INT4:  0
```

4.4.2. Accessing VSI Fortran Derived-Type Variables

To display derived-type structure variables, enter the EXAMINE command followed by the derived-type variable name, a period (.) (or a %), and the member name. For example:

```
DBG> TYPE 1:6
     1:   PROGRAM TEST
```

```
     2:
     3:       TYPE X
     4:           INTEGER A(5)
     5:       END TYPE X
     6:       TYPE (X) Z
     7:
     8:       Z%A = 1
DBG> STEP 2
stepped to TEST\%LINE 10
    10:   END PROGRAM TEST
DBG> EXAMINE Z.A
TEST\Z.A(1:5)
    (1):         1
    (2):         1
    (3):         1
    (4):         1
    (5):         1
```

4.4.3. Accessing VSI Fortran Record Variables

To display a field in a record structure, enter the EXAMINE command followed by the record name, a period (.), and the field name. To display the entire record structure, enter EXAMINE followed by the record name. For example:

```
DBG> TYPE 1:9
module TEST
   1:   PROGRAM TEST
   2:     STRUCTURE /STRA/
   3:       INTEGER*4 INT4
   4:       CHARACTER(LEN=1)  CHR
   5:     END STRUCTURE
   6:     RECORD /STRA/ REC
   7:
   8:     REC.CHR = 'L'
   9:   END PROGRAM TEST
DBG> STEP 2
stepped to TEST\%LINE 11
    11:   END PROGRAM TEST
DBG> EXAMINE REC.CHR
TEST\REC.CHR:   'L'
DBG> EXAMINE REC.INT4
TEST\REC.INT4:  0
DBG> EXAMINE REC
TEST\REC
    INT4:       0
    CHR:        'L'
```

4.4.4. Accessing VSI Fortran Array Variables

To display one or more array elements, enter the EXAMINE command followed by the array name and subscripts in parentheses, as in Fortran source statements. To display the entire array, enter EXAMINE following by the array name. For example:

```
DBG> TYPE 1:5
module ARRAY1
         1: PROGRAM ARRAY1
         2:  INTEGER (KIND=4)  ARRAY1(6)
```

```
          3:  ARRAY1 = 0
          4:  ARRAY1(1) = 1
          5:  ARRAY1(2) = 2
DBG> STEP 5
stepped to ARRAY1\%LINE 8
DBG> EXAMINE ARRAY1(1:2)
ARRAY\ARRAY1(1):        1
ARRAY\ARRAY1(2):        2
DBG> EXAMINE ARRAY1
ARRAY\ARRAY1(1:6)
     (1):         1
     (2):         2
     (3):         0
     (4):         0
     (5):         0
     (6):         0
```

4.4.5. Accessing VSI Fortran Module Variables

To display a variable defined in a module, enter a SET MODULE command before examining module variables. For example, with a variable named PINTA defined in a module named MOD1, enter the following EXAMINE command to display its value:

```
DBG> SET MODULE MOD1
DBG> TYPE 1:6
     1: PROGRAM USEMODULE
     2:   USE MOD1
     3:   INT4=0
     4:   INT4(1)=1
     5:   PINTA = 4
     6: END PROGRAM USEMODULE
DBG> STEP 4
stepped to USEMODULE\%LINE 6
   6: END PROGRAM USEMODULE
DBG> EXAMINE PINTA
USEMODULE\PINTA:  4
```

4.5. Debugger Command Summary

The following sections list all the debugger commands and any related DCL commands in functional groupings, along with brief descriptions. See the debugger's online help for complete details on commands.

During a debugging session, you can get online HELP on any command and its qualifiers by typing the HELP command followed by the name of the command in question. The HELP command has the following form:

```
HELP command
```

4.5.1. Starting and Terminating a Debugging Session

$ RUN	Invokes the debugger if LINK/DEBUG was used.
$ RUN/[NO]DEBUG	Controls whether the debugger is invoked when the program is executed.

DBG> **Ctrl/Z** or EXIT	Ends a debugging session, executing all exit handlers.
DBG> QUIT	Ends a debugging session without executing any exit handlers declared in the program.
DBG> **Ctrl/C**	Aborts program execution or a debugger command without interrupting the debugging session.
{ DBG> SET | DBG> SHOW }**ABORT_KEY**	Assigns the default **Ctrl/C** abort function to another Ctrl-key sequence or identifies the Ctrl-key sequence currently defined for the abort function.
$ Ctrl/Y DEBUG	The sequence **Ctrl/Y** DEBUG interrupts a program that is running without debugger control and invokes the debugger.
DBG> ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH).
DBG> SPAWN	Creates a subprocess; lets you issue DCL commands without interrupting your debugging context (similar to the DCL command SPAWN).
$ DEBUG/KEEP	Invokes the kept debugger, which allows certain additional commands to be used, including RUN and RERUN.
DBG> RUN *image-name*	When using the kept debugger, runs the specified program.
DBG> RERUN	When using the kept debugger, runs the last program executed again.

4.5.2. Controlling and Monitoring Program Execution

GO	Starts or resumes program execution.
STEP	Executes the program up to the next line, instruction, or specified instruction.
{ SET | SHOW }**STEP**	Establishes or displays the default qualifiers for the STEP command.
{ SET | SHOW | CANCEL | ACTIVATE | DEACTIVATE }**BREAK**	Sets, displays, cancels, activates, or deactivates breakpoints.
{ SET | SHOW | CANCEL | ACTIVATE | DEACTIVATE }**TRACE**	Sets, displays, cancels, activates, or deactivates tracepoints.
{ SET | SHOW | CANCEL | ACTIVATE | DEACTIVATE }**WATCH**	Sets, displays, cancels, activates, or deactivates watchpoints.
SHOW CALLS	Identifies the currently active subroutine calls.
SHOW STACK	Gives additional information about the currently active subroutine calls.
CALL	Calls a subroutine.

4.5.3. Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location
SET MODE [NO]OPERANDS	Controls whether the address and contents of the instruction operands are displayed when you examine an instruction.

DEPOSIT	Changes the value of a variable or the contents of a program location.
EVALUATE	Evaluates a language or address expression.

4.5.4. Controlling Type Selection and Symbolization

{ SET | SHOW | CANCEL }**RADIX**	Establishes the radix for data entry and display, displays the radix, or restores the radix.
{ SET | SHOW | CANCEL }**TYPE**	Establishes the type for program locations that are not associated with a compiler generated type, displays the type, or restores the type.
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT. You can also use SET TYPE or EXAMINE commands to define untyped program locations, such as SET TYPE S_FLOAT, EXAMINE/T_FLOAT, or EXAMINE/X_FLOAT.

4.5.5. Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program.
{ SET | SHOW }**MODULE**	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module.
{ SET | SHOW }**IMAGE**	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image.
SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are set automatically when the debugger interrupts execution.
{ SET | SHOW | CANCEL }**SCOPE**	Establishes, displays, or restores the scope for symbol lookup.
SET MODE [NO]LINE	Controls whether code locations are displayed as line numbers or routine-name + byte offset.
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or as numeric addresses.
SYMBOLIZE	Converts a virtual address to a symbolic address.

4.5.6. Displaying Source Code

TYPE	Displays lines of source code.
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression.
{ SET | SHOW | CANCEL }**SOURCE**	Creates, displays, or cancels a source directory search list.
SEARCH	Searches the source code for the specified string.
{ SET | SHOW }**SEARCH**	Establishes or displays the default qualifiers for the SEARCH command.

SET STEP [NO]SOURCE	Enables or disables the display of source code after a STEP command has been executed or at a breakpoint, tracepoint, or watchpoint.
{ SET | SHOW }**MAX_SOURCE_FILES**	Establishes or displays the maximum number of source files that may be kept open at one time.
{ SET | SHOW }**MARGINS**	Establishes or displays the left and right margin settings for displaying source code.

4.5.7. Using Screen Mode

SET MODE [NO]SCREEN	Enables or disables screen mode.
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command.
DISPLAY	Modifies an existing display.
{ SET | SHOW | CANCEL }**DISPLAY**	Creates, identifies, or deletes a display.
{ SET | SHOW | CANCEL }**WINDOW**	Creates, identifies, or deletes a window definition.
SELECT	Selects a display for a display attribute.
SHOW SELECT	Identifies the displays selected for each of the display attributes.
SCROLL	Scrolls a display.
SAVE	Saves the current contents of a display and writes it to another display.
EXTRACT	Saves a display or the current screen state and writes it to a file.
EXPAND	Expands or contracts a display.
MOVE	Moves a display across the screen.
{ SET | SHOW }**TERMINAL**	Establishes or displays the height and width of the screen.
{ Ctrl/W | DISPLAY/REFRESH }	Refreshes the screen.

4.5.8. Editing Source Code

EDIT	Invokes an editor during a debugging session.
{ SET | SHOW }**EDITOR**	Establishes or identifies the editor invoked by the EDIT command.

4.5.9. Defining Symbols

DEFINE	Defines a symbol as an address, command, value, or process group.
DELETE	Deletes symbol definitions.
{ SET | SHOW }**DEFINE**	Establishes or displays the default qualifier for the DEFINE command.
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined.

4.5.10. Using Keypad Mode

SET MODE [NO]KEYPAD	Enables or disables keypad mode.
DEFINE/KEY	Creates key definitions.
DELETE/KEY	Deletes key definitions.
SET KEY	Establishes the key definition state.
SHOW KEY	Displays key definitions.

4.5.11. Using Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures.
{SET | SHOW} LOG	Specifies or identifies the debugger log file.
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged.
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated.
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed.
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command.
{SET | SHOW} ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures.
@file-spec	Executes a command procedure.

4.5.12. Using Control Structures

IF	Executes a list of commands conditionally.
FOR	Executes a list of commands repetitively.
REPEAT	Executes a list of commands repetitively.
WHILE	Executes a list of commands conditionally, possibly multiple times.
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop.

4.5.13. Additional Commands

SET PROMPT	Specifies the debugger prompt.
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages.
{ SET | SHOW }**LANGUAGE**	Establishes or displays the current language.
{ SET | SHOW } { EVENT_FACILITY }	Establishes or identifies the current run-time facility for language-specific events.
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program.
{ SET | SHOW }**TASK**	Modifies the tasking environment or displays task information.

{ DISABLE | ENABLE | SHOW }**AST**	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled.
SET MODE [NO]SEPARATE	Controls whether a separate window is created on a workstation for debugger input and output (this command has no effect on VT-series terminals).

For More Information:

- On debugger commands and the DECwindows Motif interface, see the *VSI OpenVMS Debugger Manual*.
- On debugger command syntax, enter the online debugger HELP command.

4.6. Locating an Exception

The OpenVMS Debugger supports the SET BREAK/EXCEPTION and SET TRACE/EXCEPTION commands to set a breakpoint or tracepoint when an exception occurs. To allow precise reporting of the exception, compile the program using:

- The FORTRAN command qualifier /SYNCHRONOUS_EXCEPTIONS (Alpha only)
- The FORTRAN command qualifier /CHECK=ALL (in general) or /CHECK=FP_EXCEPTIONS (if you know it is a floating-point exception)
- The FORTRAN command qualifiers /DEBUG and /NOOPTIMIZE

If you use the FORTRAN command qualifier /FLOAT=IEEE_FLOAT to specify IEEE floating-point (S_float and T_float) data, you can also use the /IEEE_MODE qualifier to indicate how exceptions should be handled.

For example, you might use the following commands to create the executable program:

```
$ FORTRAN/DEBUG/NOOPTIMIZE/FLOAT=IEEE_FLOAT/SYNCHRONOUS_EXC/CHECK=ALL TEST ❶
$ LINK/DEBUG TEST
$ RUN TEST

         OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx

%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES
%DEBUG-I-NOTATMAIN, enter GO to get to start of main program
DBG> GO                                        ❷
break at routine TEST$MAIN      1:          REAL (KIND=4) :: A,B
DBG> SET BREAK/EXCEPTION DO (SHOW CALLS)   ❸
DBG> GO                                        ❹

%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000000,
Fmask=00000002, summary=08, PC=00020050, PS=0000001B          ❺
-SYSTEM-F-FLTOVF, arithmetic trap,floating overflow at PC=00020050,PS=0000001B
break on exception preceding TEST$MAIN\%LINE 3+20
    3:         B=A*A
  module name     routine name                      line       rel PC    abs PC
 *TEST$MAIN       TEST$MAIN                            3      00000050  00020050
                                                              00000130  00020130
   SHARE$DEC$FORRTL
```

```
                                                    00000000  000B0A30
                                                    00000130  00020130
                                                    00000000  84F4BAD8

DBG> TYPE 1:3                                  ❻
     1:          REAL (KIND=4) :: A,B
     2:          A=2.5138E20
     3:          B=A*A
DBG> EXIT
$
```

❶ The FORTRAN command line specifies qualifiers that ensure reporting of exceptions and sufficient information for debugging.

The LINK command /DEBUG qualifier requests that debug symbol information be included in the executable image.

❷ The first debugger GO command is needed because run-time checking was requested.

❸ The debugger command SET BREAK/EXCEPTION sets a breakpoint for exceptions. If omitted, the exception is not reported.

❹ The second debugger GO command runs the program.

❺ The "%SYSTEM-F-HPARITH, high performance arithmetic trap" message indicates an exception has occurred. The "-SYSTEM-F-FLTOVF, arithmetic trap, floating overflow" message indicates the type of exception. The remaining display (requested by the SHOW CALLS command) shows the routine and line number where the error occurred.

❻ The TYPE command displays the area of source code associated with the exception.

For More Information:

- On the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier, see Section 2.3.46.
- On the /IEEE_MODE qualifier, see Section 2.3.24.
- On the /FLOAT qualifier, see Section 2.3.22.
- On other FORTRAN command qualifiers, see Section 2.3 or online DCL HELP (HELP FORTRAN).

4.7. Locating Unaligned Data

The OpenVMS Debugger supports the SET BREAK/UNALIGNED command to set a breakpoint when unaligned data is accessed. To allow precise reporting of the source code accessing the unaligned data, compile the program using the FORTRAN command qualifier /SYNCHRONOUS_EXCEPTIONS (Alpha only).

For example, you might use the following commands to create the executable program:

```
$ FORTRAN/DEBUG/NOOPTIMIZE/SYNCHRONOUS_EXCEPT INV_ALIGN ❶
$ LINK/DEBUG INV_ALIGN
$ RUN INV_ALIGN
          OpenVMS DEBUG (IA64 Debug64) Version x.x-xxx
%DEBUG-I-INITIAL, language is FORTRAN, module set to INV_ALIGN

DBG> SET BREAK/UNALIGNED  ❷
DBG> GO
Unaligned data access: virtual address = 0003000A, PC = 000200A0 ❸
break on unaligned data trap preceding INV_ALIGN\OOPS\%LINE 10
```

```
    10:       end
DBG> TYPE 7:9
      7:        subroutine oops(i4)
      8:        integer*4 i4
      9:        i4 = 1
DBG> EXIT
$
```

❶ The FORTRAN command line specifies qualifiers that ensure reporting of exceptions and sufficient information for debugging.

The LINK command /DEBUG qualifier requests that debug symbol information be included in the executable image.

❷ The debugger command SET BREAK/UNALIGNED sets a breakpoint for unaligned data. If omitted, the unaligned data would not be reported.

❸ The “Unaligned data access” message indicate the line causing the unaligned data was just before line 10.

The TYPE command is used to display the lines before line 10. In this case, the integer argument whose address is passed to subroutine OOPS is unaligned, resulting in the unaligned access.

For More Information:

- On unaligned data, see Section 5.3.
- On the FORTRAN command /ALIGNMENT qualifier, see Section 2.3.3.
- On other FORTRAN command qualifiers, see Section 2.3 or online DCL HELP (HELP FORTRAN).

Chapter 5. Performance: Making Programs Run Faster

This chapter describes:

- Section 5.1: Software Environment and Efficient Compilation
- Section 5.2: Analyzing Program Performance
- Section 5.3: Data Alignment Considerations
- Section 5.4: Using Arrays Efficiently
- Section 5.5: Improving Overall I/O Performance
- Section 5.6: Additional Source Code Guidelines for Run-Time Efficiency
- Section 5.7: Optimization Levels: /OPTIMIZE=LEVEL= *n* Qualifier
- Section 5.8: Other Qualifiers Related to Optimization
- Section 5.9: Compiler Directives Related to Performance

5.1. Software Environment and Efficient Compilation

Before you attempt to analyze and improve program performance, you should:

- Obtain and install the latest version of VSI Fortran, along with performance products that can improve application performance, such as the VSI Extended Mathematical Library (VXML).
- If possible, obtain and install the latest version of the OpenVMS operating system and processor firmware for your system.
- Use the FORTRAN command and its qualifiers in a manner that lets the VSI Fortran compiler perform as many optimizations as possible to improve run-time performance.
- Use certain performance capabilities provided by the OpenVMS operating system.

5.1.1. Install the Latest Version of VSI 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 VSI Fortran

  New releases of the VSI Fortran compiler and its associated run-time libraries may provide new features that improve run-time performance. The VSI Fortran run-time libraries are shipped with the OpenVMS operating system.

  If your application will be run on an OpenVMS system other than your program development system, be sure to use the same (or later) version of the OpenVMS operating system on those systems.

You can obtain the appropriate VSI Services software product maintenance contract to automatically receive new versions of VSI Fortran (or the OpenVMS operating system). For information on more recent VSI Fortran releases, contact the VSI Customer Support Center (CSC) if you have the appropriate support contract, or contact your local VSI sales representative or authorized reseller.

- VSI Extended Mathematical Library (VXML) for OpenVMS Alpha Systems

  Calling the VSI Extended Mathematical Library (VXML) routines and installing the VXML product can make certain applications run significantly faster on OpenVMS Alpha systems. Refer to Chapter 15 for information on VXML.

- Performance and Coverage Analyzer (profiler part of DECset)

  You can purchase the Performance and Coverage Analyzer (PCA) product, which performs code profiling. PCA is one of a group of products comprising a development environment available from VSI known as DECset. Other DECset products include the Language-Sensitive Editor (LSE), Source Code Analyzer (SCA), Code Management System (CMS), and the DEC/Test Manager (DTM).

  Use of the Source Code Analyzer (SCA) is supported by using the /ANALYSIS_DATA qualifier (see Section 2.3.4) to produce an analysis data file.

- Other 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.

  Adequate process quotas and pagefile 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 OpenVMS systems, see the *VSI OpenVMS System Manager's Manual*.

5.1.2. Compile Using Multiple Source Files and Appropriate FORTRAN Qualifiers

During the earlier stages of program development, you can use incremental compilation with minimal optimization. For example:

```
$ FORTRAN /OPTIMIZE=LEVEL=1 SUB2
$ FORTRAN /OPTIMIZE=LEVEL=1 SUB3
$ FORTRAN /OPTIMIZE=LEVEL=1 MAIN
$ LINK MAIN SUB2 SUB3
```

During the later stages of program development, you should compile multiple source files together and use an optimization level of at least /OPTIMIZE=LEVEL=4 on the FORTRAN 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 (/OPTIMIZE=LEVEL=4):

```
$ FORTRAN MAIN.F90+SUB2.F90+SUB3.F90
$ LINK MAIN.OBJ
```

Compiling multiple source files using the plus sign (+) separator 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

When compiling all source files together is not feasible (such as for very large programs), consider compiling source files containing related routines together with multiple FORTRAN commands, rather than compiling source files individually.

Table 5.1 shows FORTRAN qualifiers that can improve performance. Most of these qualifiers do not affect the accuracy of the results, while others improve run-time performance but can change some numeric results.

VSI Fortran performs certain optimizations unless you specify the appropriate FORTRAN command qualifiers. Additional optimizations can be enabled or disabled using FORTRAN command qualifiers.

Table 5.1 lists the FORTRAN qualifiers that can directly improve run-time performance.

Table 5.1. FORTRAN Qualifiers Related to Run-Time Performance

Qualifier Names	Description and For More Information
/ALIGNMENT= `keyword`	Controls whether padding bytes are added between data items within common blocks, derived-type data, and Compaq Fortran 77 record structures to make the data items naturally aligned. See Section 5.3.
/ASSUME= NOACCURACY_SENSITIVE	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 (/ASSUME=ACCURACY_SENSITIVE) because of the way intermediate results are rounded. This slight difference in numeric results is acceptable to most programs. See Section 5.8.8.
/ARCHITECTURE= `keyword` (Alpha only)	Specifies the type of Alpha architecture code instructions to be generated for the program unit being compiled; it uses the same options (keywords) as used by the /OPTIMIZE=TUNE qualifier (Alpha only) (which controls instruction scheduling). See Section 2.3.6.
/FAST	Sets the following performance-related qualifiers: /ALIGNMENT=(COMMONS=NATURAL, RECORDS=NATURAL, SEQUENCE) /ARCHITECTURE=HOST, /ASSUME= NOACCURACY, /MATH_LIBRARY=FAST (Alpha only), and /OPTIMIZE=TUNE=HOST (Alpha only). See Section 5.8.3.
/INTEGER_SIZE= `nn`	Controls the sizes of INTEGER and LOGICAL declarations without a kind parameter. See Section 2.3.26.

Qualifier Names	Description and For More Information
/MATH_LIBRARY=FAST (Alpha only)	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. See Section 2.3.30.
/OPTIMIZE=INLINE= `keyword`	Specifies the types of procedures to be inlined. If omitted, /OPTIMIZE=LEVEL= `n` determines the types of procedures inlined. Certain INLINE keywords are relevant only for /OPTIMIZE=LEVEL=1 or higher. See Section 2.3.35.
/OPTIMIZE=LEVEL= `n` (n = 0 to 5)	Controls the optimization level and thus the types of optimization performed. The default optimization level is /OPTIMIZE=LEVEL=4. Use /OPTIMIZE=LEVEL=5 to activate loop transformation optimizations. See Section 5.7.
/OPTIMIZE=LOOPS	Activates a group of loop transformation optimizations (a subset of /OPTIMIZE=LEVEL=5). See Section 5.7.
/OPTIMIZE=PIPELINE	Activates the software pipelining optimization (a subset of /OPTIMIZE=LEVEL=4). See Section 5.7.
/OPTIMIZE=TUNE= `keyword` (Alpha only)	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 /OPTIMIZE=TUNE= `xxxx`, the generated code will run correctly on all implementations of the Alpha architecture. See Section 5.8.6.
/OPTIMIZE=UNROLL= `n`	Specifies the number of times a loop is unrolled (`n`) when specified with optimization level /OPTIMIZE=LEVEL=3 or higher. If you omit /OPTIMIZE=UNROLL= `n`, the optimizer determines how many times loops are unrolled. See Section 5.7.4.1.
/REENTRANCY	Specifies whether code generated for the main program and any Fortran procedures it calls will be relying on threaded or asynchronous reentrancy. See Section 2.3.39.

Table 5.2 lists qualifiers that can slow program performance. Some applications that require floating-point exception handling or rounding need to use the /IEEE_MODE and /ROUNDING_MODE qualifiers. Other applications might need to use the /ASSUME=DUMMY_ALIASES qualifier for compatibility reasons. Other qualifiers listed in Table 5.2 are primarily for troubleshooting or debugging purposes.

Table 5.2. Qualifiers that Slow Run-Time Performance

Qualifier Names	Description and 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, Fortran 90, and Fortran 95 standards but occurs in some older programs. See Section 5.8.9.
/CHECK[= *keyword*]	Generates extra code for various types of checking at run time. This increases the size of the executable image, but may be needed for certain programs to handle arithmetic exceptions. Avoid using /CHECK=ALL except for debugging purposes. See Section 2.3.11.
/IEEE_MODE= `keyword` other than /IEEE_MODE= DENORM_RESULTS (on I64) or /IEEE_MODE=FAST (on Alpha)	On Alpha systems, using /IEEE_MODE=UNDERFLOW_TO_ZERO slows program execution (like /SYNCHRONOUS_EXCEPTIONS (Alpha only)). Using /IEEE_MODE=DENORM_RESULTS slows program execution even more than /IEEE_MODE=UNDERFLOW_TO_ZERO. See Section 2.3.24.
/ROUNDING_MODE=DYNAMIC	Certain rounding modes and changing the rounding mode can slow program execution slightly. See Section 2.3.40.
/SYNCHRONOUS_EXCEPTIONS	Generates extra code to associate an arithmetic exception with the instruction that causes it, slowing program execution. Use this qualifier only when troubleshooting, such as when identifying the source of an exception. See Section 2.3.46.
/OPTIMIZE=LEVEL=0, /OPTIMIZE=LEVEL=1, /OPTIMIZE=LEVEL=2,	Minimizes the optimization level (and types of optimizations). Use during the early stages of program development or when you will use the debugger. See Section 2.3.35 and Section 5.7.

Qualifier Names	Description and For More Information
/OPTIMIZE=LEVEL=3	
/OPTIMIZE=INLINE= NONE, /OPTIMIZE=INLINE= MANUAL	Minimizes the types of inlining done by the optimizer. Use such qualifiers only during the early stages of program development. The type of inlining optimizations are also controlled by the /OPTIMIZE=LEVEL qualifier. See Section 2.3.35 and Section 5.7.

For More Information:

- On compiling multiple files, see Section 2.2.1.
- On minimizing external references, see Section 10.2.1.

5.1.3. Process Environment and Related Influences on Performance

Certain DCL 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. In some cases, inadequate process limits may prolong compilation or program execution. For more information, see Section 1.2.

  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 the MONITOR command.

  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 to SYS$OUTPUT 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:

```
$ DEFINE /USER FOR006 RESULTS.LIS
$ RUN MYPROG
$ TYPE/PAGE RESULTS.LIS
```

For More Information:

About system-wide tuning and suggestions for other performance enhancements on OpenVMS systems, see the *VSI OpenVMS System Manager's Manual*.

5.2. Analyzing Program Performance

This section describes how you can:

- Analyze program performance using timings of program execution using LIB$xxxx_TIMER routines or an equivalent DCL command procedure (Section 5.2.1)
- Analyze program performance using the optional Performance Coverage Analyzer tool (Section 5.2.2)

Before you analyze program performance, make sure any errors you might have encountered during the early stages of program development have been corrected.

5.2.1. Measuring Performance Using LIB$xxxx_TIMER Routines or Command Procedures

You can use LIB$xxxx_TIMER routines or an equivalent DCL command procedure to measure program performance.

Using the LIB$xxxx_TIMER routines allows you to display timing and related statistics at various points in the program as well as at program completion, including elapsed time, actual CPU time, buffered I/O, direct I/O, and page faults. If needed, you can use other routines or system services to obtain and report other information.

You can measure performance for the entire program by using a DCL command procedure (see Section 5.2.1.2). Although using a DCL command procedure does not report statistics at various points in the program, it can provide information for the entire program similar to that provided by the LIB$ xxxx_TIMER routines.

5.2.1.1. The LIB$xxxx_TIMER Routines

Use the following routines together to provide information about program performance at various points in your program:

- LIB$INIT_TIMER stores the current values of specified times and counts for use by LIB$SHOW_TIMER or LIB$STAT_TIMER routines.
- LIB$SHOW_TIMER returns times and counts accumulated since the last call to LIB$INIT_TIMER and displays them on SYS$OUTPUT.
- LIB$STAT_TIMER returns times and counts accumulated since the last call to LIB$INIT_TIMER and stores them in memory.

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, repeat the timings several times to ensure that the results are not misleading. Overhead functions might influence short timings considerably.

You can use the LIB$SHOW_TIMER (or LIB$STAT_TIMER) routine to return elapsed time, CPU time, buffered I/O, direct I/O, and page faults:

- The **elapsed time**, which will be greater than the total charged actual CPU time. Sometimes called "wall clock" time.
- Charged actual **CPU time** is the amount of actual CPU time used by the process.
- **Buffered I/O** occurs when an intermediate buffer is used from the system buffer pool, instead of a process-specific buffer.
- **Direct I/O** is when I/O transfer takes place directly between the process buffer and the device.
- A **page fault** is when a reference to a page occurs that is not in the process working set.

The VSI Fortran program shown in Example 5.1 reports timings for the three different sections of the main program, including accumulative statistics (for a scalar program).

Example 5.1. Measuring Program Performance Using LIB$SHOW_TIMER and LIB$INIT_TIMER

```
!  Example use of LIB$SHOW_TIMER to time an HP Fortran program

 PROGRAM TIMER

   INTEGER TIMER_CONTEXT
   DATA    TIMER_CONTEXT /0/

!  Initialize default timer stats to 0

   CALL LIB$INIT_TIMER

!  Sample first section of code to be timed

   DO I=1,100
     CALL MOM
   ENDDO

!  Display stats

   TYPE *,'Stats for first section'
   CALL LIB$SHOW_TIMER

!  Zero second timer context

   CALL LIB$INIT_TIMER (TIMER_CONTEXT)

!  Sample second section of code to be timed

   DO I=1,1000
     CALL MOM
   ENDDO

!  Display stats

   TYPE *,'Stats for second section'
   CALL LIB$SHOW_TIMER (TIMER_CONTEXT)
   TYPE *,'Accumulated stats for two sections'
   CALL LIB$SHOW_TIMER
```

```
!  Re-Initialize second timer stats to 0

   CALL LIB$INIT_TIMER (TIMER_CONTEXT)

!  Sample Third section of code to be timed

   DO I=1,1000
     CALL MOM
   ENDDO

!  Display stats

   TYPE *,'Stats for third section'
   CALL LIB$SHOW_TIMER (TIMER_CONTEXT)
   TYPE *,'Accumulated stats for all sections'
   CALL LIB$SHOW_TIMER

 END PROGRAM TIMER

!  Sample subroutine performs enough processing so times aren't all 0.0

 SUBROUTINE MOM
   COMMON  BOO(10000)
   DOUBLE PRECISION BOO
   BOO = 0.5    ! Initialize all array elements to 0.5

   DO I=2,10000
      BOO(I)   = 4.0+(BOO(I-1)+1)*BOO(I)*COSD(BOO(I-1)+30.0)
      BOO(I-1) = SIND(BOO(I)**2)
   ENDDO

   RETURN

 END SUBROUTINE MOM
```

The LIB$xxxx_TIMER routines use a single default time when called without an argument. When you call LIB$xxxx_TIMER routines with an INTEGER argument whose initial value is 0 (zero), you enable use of multiple timers.

The LIB$INIT_TIMER routine must be called at the start of the timing. It can be called again at any time to reset (set to zero) the values.

In Example 5.1, LIB$INIT_TIMER is:

- Called once at the start of the program without an argument. This initializes what will become accumulated statistics and starts the collection of the statistics. You can think of this as the first timer.
- Called once at the start of each section with the INTEGER context argument TIMER_CONTEXT. This resets the values for the current section to zero and starts the collection of the statistics. You can think of this as the second timer, which gets reset for each section.

The LIB$SHOW_TIMER routine displays the timer values saved by LIB$INIT_TIMER to SYS$OUTPUT (or to a specified routine). Your program must call LIB$INIT_TIMER before LIB$SHOW_TIMER at least once (to start the timing).

Like LIB$INIT_TIMER:

- Calling LIB$SHOW_TIMER without any arguments displays the default accumulated statistics.
- Calling LIB$SHOW_TIMER with an INTEGER context variable (TIMER_CONTEXT) displays the statistics for the current section.

The free-format source file, TIMER.F90, might be compiled and linked as follows:

```
$ FORTRAN/FLOAT=IEEE_FLOAT TIMER
$ LINK TIMER
```

When the program is run (on a low-end Alpha system), it displays timing statistics for each section of the program as well as accumulated statistics:

```
$ RUN TIMER
Stats for first section
 ELAPSED:    0 00:00:02.36  CPU: 0:00:02.21  BUFIO: 1  DIRIO: 0  FAULTS: 23
Stats for second section
 ELAPSED:    0 00:00:22.31  CPU: 0:00:22.09  BUFIO: 1  DIRIO: 0  FAULTS: 0
Accumulated stats for two sections
 ELAPSED:    0 00:00:24.68  CPU: 0:00:24.30  BUFIO: 5  DIRIO: 0  FAULTS: 27
Stats for third section
 ELAPSED:    0 00:00:22.24  CPU: 0:00:21.98  BUFIO: 1  DIRIO: 0  FAULTS: 0
Accumulated stats for all sections
 ELAPSED:    0 00:00:46.92  CPU: 0:00:46.28  BUFIO: 9  DIRIO: 0  FAULTS: 27

$
```

You might:

- Run the program multiple times and average the results.
- Use different compilation qualifiers to see which combination provides the best performance.

Instead of the LIB$xxxx_TIMER routines (specific to the OpenVMS operating system), you might consider modifying the program to call other routines within the program to measure execution time (but not obtain other process information). For example, you might use VSI Fortran intrinsic procedures, such as SYSTEM_CLOCK, DATE_AND_TIME, and TIME.

For More Information:

- On the LIB$ RTL routines, see the *VSI OpenVMS RTL Library (*LIB$*) Manual*.
- On VSI Fortran intrinsic procedures, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.2.1.2. Using a Command Procedure

Some of the information obtained by using the LIB$ xxxx_TIMER routines can be obtained using a command procedure. You should be aware of the following:

- Using a command procedure does not require source code modification. Using LIB$ xxxx_TIMER routines requires that you modify the source program.
- Using a command procedure can only provide performance timings and statistics for the entire program. Using LIB$ xxxx_TIMER routines provides performance timings and statistics for individual sections of the program and/or the entire program.

Before using a command procedure to measure performance, define a foreign symbol that runs the program to be measured in a subprocess. In the following example, the name of the command procedure is TIMER:

```
$ TIMER :== SPAWN /WAIT /NOLOG @SYS$LOGIN:TIMER
```

The command procedure shown in Example 5.2 uses the F$GETJPI lexical function to measure performance statistics and the F$FAO lexical function to report the statistics. Each output line is saved as a logical name, which can be saved by the parent process if needed.

Example 5.2. Command Procedure that Measures Program Performance

```
$    verify = 'f$verify(0)
$
$! Get initial values for stats (this removes SPAWN overhead or the current
$! process values).
$
$ bio1 = f$getjpi (0, "BUFIO")
$ dio1 = f$getjpi (0, "DIRIO")
$ pgf1 = f$getjpi (0, "PAGEFLTS")
$ vip1 = f$getjpi (0, "VIRTPEAK")
$ wsp1 = f$getjpi (0, "WSPEAK")
$ dsk1 = f$getdvi ("sys$disk:","OPCNT")
$ tim1 = f$time ()
$
$ set noon
$ tik1 = f$getjpi (0, "CPUTIM")
$ set noverify
$
$! User command being timed:
$
$ 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8'
$
$ tik2 = f$getjpi (0, "CPUTIM")
$
$ bio2 = f$getjpi (0, "BUFIO")
$ dio2 = f$getjpi (0, "DIRIO")
$ pgf2 = f$getjpi (0, "PAGEFLTS")
$ vip2 = f$getjpi (0, "VIRTPEAK")
$ wsp2 = f$getjpi (0, "WSPEAK")
$ dsk2 = f$getdvi ("sys$disk:","OPCNT")
$ tim2 = f$time ()
$
$ tim  = f$cvtime("''f$cvtime(tim2,,"TIME")'-''f$cvtime(tim1,,"TIME")'",,"TIME")
$ thun = 'f$cvtime(tim,,"HUNDREDTH")
$ tsec = (f$cvtime(tim,,"HOUR")*3600) + (f$cvtime(tim,,"MINUTE")*60) + -
  f$cvtime(tim,,"SECOND")
$
$ bio  = bio2 - bio1
$ dio  = dio2 - dio1
$ pgf  = pgf2 - pgf1
$ dsk  = dsk2 - dsk1
$ vip  = ""
$ if vip2 .le. vip1 then vip = "*"   ! Asterisk means didn't change (from
 parent)
$ wsp  = ""
$ if wsp2 .le. wsp1 then wsp = "*"
$
$ tiks = tik2 - tik1
$ secs = tiks / 100
$ huns = tiks - (secs*100)
```

```
$ write sys$output ""
$!
$ time$line1 ==  -
 f$fao("Execution (CPU) sec!5UL.!2ZL   Direct I/O  !7UL   Peak working set!7UL!
1AS", -
          secs, huns, dio, wsp2, wsp)
$ write sys$output time$line1
$!
$ time$line2 ==  -
 f$fao("Elapsed (clock) sec!5UL.!2ZL   Buffered I/O!7UL   Peak virtual    !7UL!
1AS", -
                tsec, thun, bio, vip2, vip)
$ write sys$output time$line2
$!
$ time$line3 == -
 f$fao("Process ID         !AS   SYS$DISK I/O!7UL   Page faults     !7UL", -
        f$getjpi(0,"pid"), dsk, pgf)
$ write sys$output time$line3
$ if wsp+vip .nes. "" then write sys$output -
 "                                                        (* peak from parent)"
$ write sys$output ""
$
$! Place these output lines in the job logical name table, so the parent
$! can access them (useful for batch jobs to automate the collection).
$
$ define /job/nolog time$line1 "''time$line1'"
$ define /job/nolog time$line2 "''time$line2'"
$ define /job/nolog time$line3 "''time$line3'"
$
$ verify = f$verify(verify)
```

This example command procedure accepts multiple parameters, which include the RUN command, the name of the executable image to be run, and any parameters to be passed to the executable image.

```
$ TIMER RUN PROG_TEST
$
$! User command being timed:
$
$ RUN PROG_TEST.EXE;

Execution (CPU) sec   45.39   Direct I/O        3   Peak working set   2224
Elapsed (clock) sec   45.96   Buffered I/O     18   Peak virtual      15808
Process ID         20A00999   SYS$DISK I/O      6   Page faults          64
```

If your program displays a lot of text, you can redirect the output from the program. Displaying text increases the buffered I/O count. Redirecting output from the program will change the times reported because of reduced screen I/O.

For More Information:

About system-wide tuning and suggestions for other performance enhancements on OpenVMS systems, see the *VSI OpenVMS System Manager's Manual*.

5.2.2. Performance and Coverage Analyzer (PCA)

To generate profiling information, you can use the optional Performance and Coverage Analyzer (PCA) tool.

Profiling helps you identify areas of code where significant program execution time is spent; it can also identify those parts of an application that are not executed (by a given set of test data).

PCA has two components:

- The **Collector** gathers performance or test coverage data on the running program and writes that data to a performance data file. You can specify the image to be used (image selection) and characteristics of the data collection (measurement and control selection). Data collection characteristics include:
  - Program counter (PC) sampling
  - CPU sampling data
  - Counts of program execution at a location
  - Coverage of program locations
  - Other information
- The **Analyzer** reads and processes the performance data file and displays the collected data graphically in the form of histograms, tables, and annotated source listings.

PCA works with related DECset tools LSE and the Test Manager. PCA provides a callable routine interface, as well as a command-line and DECwindows Motif graphical windowing interface. The following examples demonstrate the character-cell interface.

When compiling a program for which PCA will record and analyze data, specify the /DEBUG qualifier on the FORTRAN command line:

```
$ FORTRAN /DEBUG TEST_PROG.F90
```

On the LINK command line, specify the PCA debugging module PCA$OBJ using the Linker /DEBUG qualifier:

```
$ LINK /DEBUG=SYS$LIBRARY:PCA$OBJ.OBJ TEST_PROG
```

When you run the program, the PCA$OBJ.OBJ debugging module invokes the Collector and is ready to accept your input to run your program under Collector control and gather the performance or coverage data:

```
$ RUN TEST_PROG
PCAC>
```

You can enter Collector commands, such as SET DATAFILE, SET PC_SAMPLING, GO, and EXIT.

To run the Analyzer, type the PCA command and specify the name of a performance data file, such as the following:

```
$ PCA TEST_PROG
PCAA>
```

You can enter the appropriate Analyzer commands to display the data in the performance data file in a graphic representation.

For More Information:

- On the windowing interface for PCA, see the *Guide to Performance and Coverage Analyzer for OpenVMS Systems*.

- On the character-cell interface for PCA, see the *Performance and Coverage Analyzer Command-Line Reference*.

5.3. Data Alignment Considerations

The VSI Fortran compiler aligns most numeric data items on **natural boundaries** to avoid run-time adjustment by software that can slow performance.

A natural boundary is a memory address that is a multiple of the data item's size (data type sizes are described in Table 8.1). 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 VSI Fortran compiler naturally aligns individual data items when it can, certain VSI Fortran statements (such as EQUIVALENCE) can cause data items to become unaligned (see Section 5.3.1).

Although you can use the FORTRAN command /ALIGNMENT qualifier 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.

5.3.1. Causes of Unaligned Data and Ensuring Natural Alignment

Common blocks (COMMON statement), derived-type data, and Compaq Fortran 77 record structures (STRUCTURE and RECORD statements) 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 /ALIGNMENT=COMMONS=STANDARD or /ALIGNMENT=COMMONS=NATURAL) (set by specifying /FAST), depending on the largest data size used.

  For examples and more information, see Section 5.3.3.1.

- 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 avoid specifying the FORTRAN command qualifier /ALIGNMENT= RECORDS=PACKED unless you are sure that the data items in derived-type data structures (and Compaq Fortran 77 record structures) will be naturally aligned.

  If you omit the SEQUENCE statement (and /ALIGNMENT= RECORDS=PACKED), the /ALIGNMENT=RECORDS=NATURAL qualifier ensures all data items are naturally aligned. This is the default.

If you specify the SEQUENCE statement, the /ALIGNMENT= RECORDS=NATURAL qualifier is prevented from adding necessary padding to avoid unaligned data (data items are packed). When you use the SEQUENCE statement, you should specify data declaration order such that all data items are naturally aligned, or add the /ALIGNMENT=RECORDS=SEQUENCE compiler qualifier.

For an example and more information, see Section 5.3.3.2.

- Compaq Fortran 77 record structures (RECORD and STRUCTURE statements)

  Compaq Fortran 77 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 77 record structures, you should avoid specifying the FORTRAN command qualifier /ALIGNMENT=RECORDS= PACKED unless you are sure that the data items in derived-type data and Compaq Fortran 77 record structures will be naturally aligned.

  For an example and more information, see Section 5.3.3.3.

- EQUIVALENCE statements

  EQUIVALENCE statements can force unaligned data or cause data to span natural boundaries. For more information, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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 5.3.3).

- 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 qualifiers 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 77 record structure.

When actual arguments from outside the program unit are not naturally aligned, unaligned data access will occur. VSI 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 77 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 the FORTRAN /ALIGNMENT qualifier to supply needed padding to avoid some array elements being unaligned.

If you specify /ALIGNMENT=RECORDS=PACKED (or equivalent qualifiers), 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 FORTRAN command qualifiers specified. In this case, some elements will be unaligned.

When /ALIGNMENT=RECORDS=NATURAL is in effect (default), 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 FORTRAN command /ALIGNMENT qualifier, see Section 5.3.4.

5.3.2. Checking for Inefficient Unaligned Data

During compilation, the VSI Fortran compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described in Section 5.3.1.

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 5.3.3). 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 5.3.1).

There are two ways unaligned data might be reported:

- During compilation

  During compilation, warning messages are issued for any data items that are known to be unaligned (unless you specify the /WARN=NOALIGNMENTS qualifier).

- During program execution by using the debugger

  On Alpha systems, compile the program with the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier (along with /DEBUG and /NOOPTIMIZE) to request precise reporting of any data that is detected as unaligned.

  Use the debugger (SET BREAK/UNALIGNED) command as described in Section 4.7 to check where the unaligned data is located.

For More Information:

On the /WARNINGS qualifier, see Section 2.3.51.

5.3.3. 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.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.
- If your data includes a mixture of character and numeric data, place the numeric data first.

Using the suggested data declaration guidelines minimizes the need to use the /ALIGNMENT qualifier to add padding bytes to ensure naturally aligned data. In cases where the /ALIGNMENT qualifier is still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

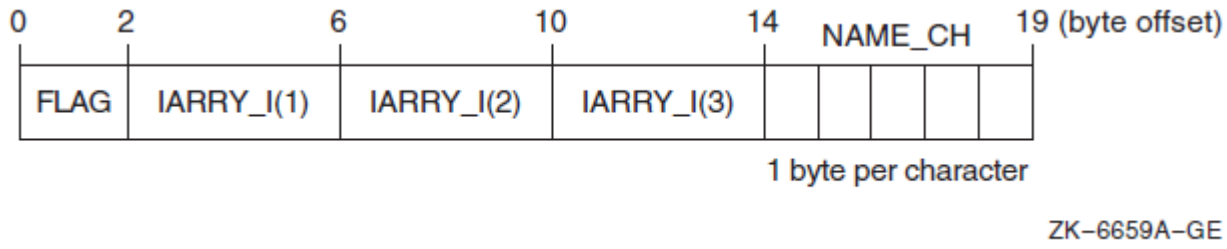
5.3.3.1. Arranging Data Items in Common Blocks

The order of data items in a COMMON statement determines 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
```

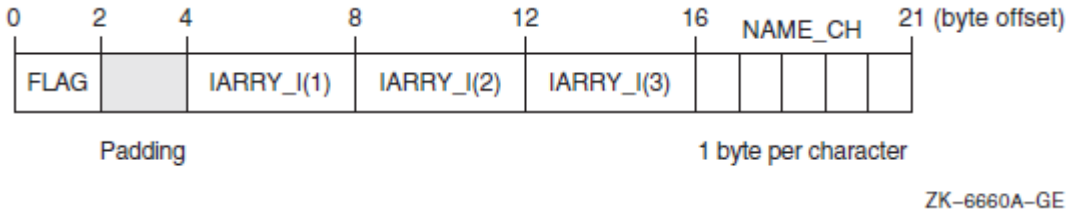
As shown in Figure 5.1, if you omit the appropriate FORTRAN command qualifiers, the common block will contain unaligned data items beginning at the first array element of IARRY_I.

Figure 5.1. Common Block with Unaligned Data



As shown in Figure 5.2, if you compile the program units that use the common block with the /ALIGNMENT=COMMONS=STANDARD qualifier, data items will be naturally aligned.

Figure 5.2. Common Block with Naturally Aligned Data



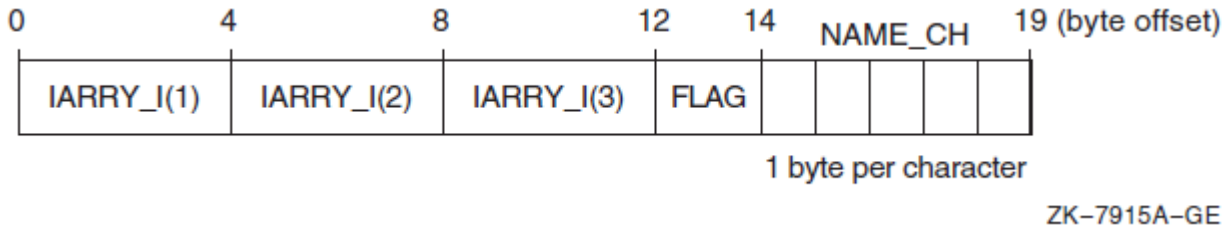
Because the common block X contains data items whose size is 32 bits or smaller, you can specify the /ALIGNMENT=COMMONS qualifier and still have naturally aligned data. If the common block contains data items whose size might be larger than 32 bits (such as REAL (KIND=8) data), specify /ALIGNMENT=COMMONS=NATURAL to ensure naturally aligned data.

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 FORTRAN command /ALIGNMENT=COMMONS=NATURAL (or equivalent) qualifier:

```
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



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.

5.3.3.2. 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 FORTRAN qualifiers. See Section 5.3.4 for information about these exceptions.

VSI 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 FORTRAN qualifiers have no effect on unaligned data, so data declarations must be carefully specified to naturally align data.

The /ALIGNMENT=SEQUENCE qualifier specifically aligns data items in a SEQUENCE derived-type on natural boundaries.

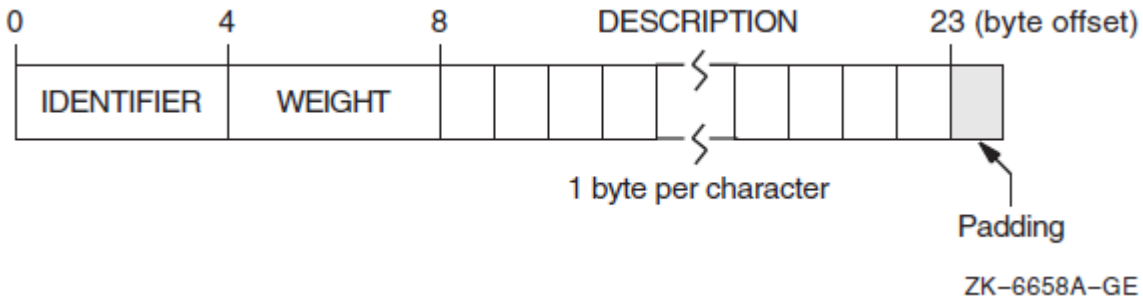
- If you omit the SEQUENCE statement, VSI Fortran adds the padding bytes needed to naturally align data item components, unless you specify the /ALIGNMENT=RECORDS=PACKED qualifier.

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 /ALIGNMENT=RECORDS=NATURAL qualifier (default) is in effect.

Figure 5.4. Derived-Type Naturally Aligned Data (in CATALOG_SPRING())



5.3.3.3. Arranging Data Items in Compaq Fortran 77 Record Structures

VSI Fortran supports record structures provided by Compaq Fortran 77. Compaq Fortran 77 record structures use the RECORD statement and optionally the STRUCTURE statement, which are extensions to the FORTRAN-77, Fortran 90, and Fortran 95 standards. The order of data items in a STRUCTURE statement determines the order in which the data items are stored.

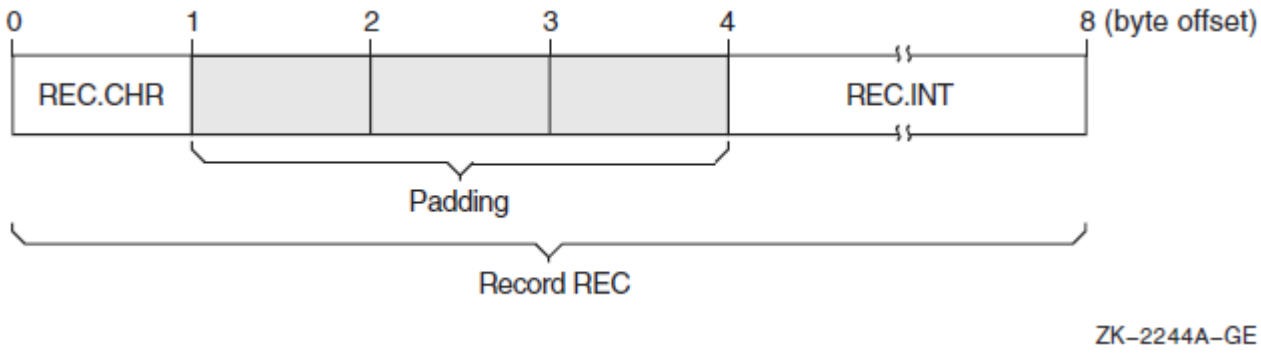
VSI Fortran stores a record in memory as a linear sequence of values, with the record's first element in the first storage location and its last element in the last storage location. Unless you specify the /ALIGNMENT=RECORDS=PACKED qualifier, padding bytes are added if needed to ensure data fields are 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



For More Information:

On data declaration statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.3.4. Qualifiers Controlling Alignment

The following qualifiers control whether the VSI Fortran compiler adds padding (when needed) to naturally align multiple data items in common blocks, derived-type data, and Compaq Fortran 77 record structures:

- Unless you specify /FAST, the default is /ALIGNMENT= COMMONS=PACKED) 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 /ALIGNMENT= COMMONS=STANDARD qualifier requests that data in common blocks be aligned on up to 4-byte boundaries, by adding padding bytes as needed.
- The /ALIGNMENT= COMMONS=NATURAL qualifier requests that data in common blocks be aligned on up to 8-byte boundaries, by adding padding bytes as needed.

  If you specify /FAST, the default is /ALIGNMENT=COMMONS=NATURAL.
- The /ALIGNMENT= COMMONS=NATURAL qualifier is equivalent to specifying /ALIGNMENT= (COMMONS= (NATURAL,NOMULTILANGUAGE), RECORDS=NATURAL).
- The /ALIGNMENT=RECORDS=PACKED qualifier requests that multiple data items in derived-type data and record structures be aligned on byte boundaries instead of being naturally aligned. The default is /ALIGNMENT=RECORDS=NATURAL.

- The /ALIGNMENT= RECORDS=NATURAL qualifier (default) requests that multiple data items in derived-type data without the SEQUENCE statement record structures be naturally aligned, by adding padding bytes as needed.
- The /ALIGNMENT=NOSEQUENCE qualifier controls alignment of derived types with the SEQUENCE attribute. The default /ALIGNMENT=NOSEQUENCE qualifier means that derived types with the SEQUENCE attribute are packed regardless of any other alignment rules. Note that /ALIGNMENT=NONE implies /ALIGNMENT=NOSEQUENCE.

  On the other hand, the /ALIGNMENT=SEQUENCE qualifier means that derived types with the SEQUENCE attribute obey whatever alignment rules are currently in use. Consequently, since /ALIGNMENT= RECORDS is a default value, then /ALIGNMENT=SEQUENCE alone on the command line will cause the fields in these derived types to be naturally aligned. Note that /FAST and /ALIGNMENT=ALL imply /ALIGNMENT= SEQUENCE.
- The /FAST qualifier controls certain defaults, including alignment (sets /ALIGNMENT= COMMONS=NATURAL qualifier).

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* be naturally aligned (/ALIGNMENT= (COMMONS= (PACKED, NOMULTILANGUAGE), RECORDS=NATURAL).

In derived-type data, using the SEQUENCE statement
prevents /ALIGNMENT=RECORDS=NATURAL from adding needed padding bytes to naturally align data items.

For More Information:

On the /ALIGNMENT qualifier, see Section 2.3.3.

5.4. Using Arrays Efficiently

The following sections discuss these topics:

- Accessing arrays efficiently
- Passing arrays efficiently

5.4.1. Accessing Arrays Efficiently

Many of the array access efficiency techniques described in this section are applied automatically by the VSI Fortran loop transformation optimizations (see Section 5.8.1).

Several aspects of array use can improve run-time performance. The following sections describe these aspects.

Array Access

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 90/95 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
```

Multidimensional Arrays

Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the **natural ascending order** (**column major**) for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Writing a whole array 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 Fortran 90/95 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 5.5.3.

Array Intrinsic Procedures

Use the available Fortran 90/95 array intrinsic procedures rather than create your own.

Whenever possible, use Fortran 90/95 array intrinsic procedures instead of creating your own routines to accomplish the same task. VSI Fortran array intrinsic procedures are designed for efficient use with the various VSI Fortran run-time components.

Using the standard-conforming array intrinsics can also make your program more portable.

Noncontiguous Access

With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of 2 (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 2. 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.

5.4.2. Passing Array Arguments Efficiently

In VSI Fortran, there are two general types of array arguments:

- Explicit-shape arrays used with FORTRAN 77.

  These arrays have a fixed rank and extent that are 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 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.

For More Information:

On arrays and their data declaration statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.5. 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** determines 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 may be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement I/O time results in an I/O bottleneck.
- By 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.

VSI offers software solutions to system-wide problems like minimizing device I/O delays (see Section 5.1.1).

5.5.1. 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, VSI Fortran can convert unformatted data in several formats (see Chapter 9).

5.5.2. Write Whole Arrays or Strings

The general guidelines about array use discussed in Section 5.4 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 90/95 array syntax) instead of using implied-DO loops.

5.5.3. 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 5.4). 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.

5.5.4. 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 a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

5.5.5. Defaults for Blocksize and Buffer Count

VSI Fortran provides OPEN statement defaults for BLOCKSIZE and BUFFERCOUNT that generally offer adequate I/O performance. The default for BLOCKSIZE and BUFFERCOUNT is determined by SET RMS_DEFAULT command default values.

Specifying a BUFFERCOUNT of 2 (or 3) allows Record Management Services (RMS) to overlap some I/O operations with CPU operations. For sequential and relative files, specify a BLOCKSIZE of at least 1024 bytes.

Any experiments to improve I/O performance should try to increase the amount of data read by each disk I/O. For large indexed files, you can reduce disk I/O by specifying enough buffers (BUFFERCOUNT) to keep most of the index portion of the file in memory.

For More Information:

- On tuning indexed files and optimal BUFFERCOUNT and BLOCKSIZE values, see the *Guide to OpenVMS File Applications*.
- On specifying BLOCKSIZE and BUFFERCOUNT, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.5.6. Specify RECL

When creating a file, you should consider specifying a RECL value that provides for adequate I/O performance. The RECL value unit differs for unformatted files (4-byte units) and formatted files (1-byte units).

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 qualifier to request 1-byte units (see Section 2.3.7).

When porting unformatted data files from non-VSI systems, see Section 9.6.

For More Information:

- On optimal RECL (record length) values, see the *Guide to OpenVMS File Applications*.
- On specifying RECL, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.5.7. Use the Optimal Record Type

Unless a certain record type is needed for portability reasons (see Section 6.5.3), 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, use variable-length or segmented records.
- For sequential formatted files when records are not fixed in size, use variable-length records, unless you need to use Stream_LF records for data porting compatibility (see Section 6.5.3).

For More Information:

- On VSI Fortran data files and I/O, see Chapter 6.
- On OPEN statement specifiers and defaults, see Section 6.6 and the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.5.8. 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 VSI Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

VSI 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. VSI 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On loop optimizations, see Section 5.7.

5.5.9. Use of Variable Format Expressions

Variable format expressions (a Compaq Fortran 77 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. 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.

5.6.1. Avoid Small or Large Integer and Logical Data Items (Alpha only)

If the target system is an Alpha processor predating EV56, avoid using integer or logical data items whose size is less than 32 bits. On those processors, the smallest unit of efficient single-instruction access is 32 bits, and accessing a 16-bit (or 8-bit) data type can result in a sequence of machine instructions to access the data.

5.6.2. 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 /ASSUME= NOACCURACY_SENSITIVE qualifier (see Section 5.8.8) 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.

5.6.3. 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 (See also Section 5.6.1)
- 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 5.6.2).

5.6.4. 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 VSI Fortran arithmetic operators, from fastest to slowest:

- Addition (+), subtraction (-), and floating-point multiplication (*)
- Integer multiplication (*)
- Division (/)
- Exponentiation (**)

5.6.5. Avoid EQUIVALENCE Statement Use

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 5.7.3)
  - Implied-DO loop collapsing when the control variable is contained in an EQUIVALENCE statement

5.6.6. Use Statement Functions and Internal Subprograms

Whenever the VSI Fortran compiler has access to the use and definition of a subprogram during compilation, it might 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 /OPTIMIZE=LEVEL=4 or higher.

For more information, see Section 5.1.2.

5.6.7. 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 5.8.2 and Section 5.8.4.
- On VSI Fortran statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.7. Optimization Levels: /OPTIMIZE=LEVEL= *n* Qualifier

VSI 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, VSI Fortran increases compile time in favor of decreasing run time. If an operation can be performed, eliminated, or simplified at compile time, VSI 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 /OPTIMIZE=LEVEL=4, but will require more compilation time than if you compile the program at a lower level of optimization.

The size of the 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 VSI Fortran optimization with different /OPTIMIZE=LEVEL= `n` levels. For example, /OPTIMIZE=LEVEL=0 specifies no selectable optimizations (certain optimizations always occur); /OPTIMIZE=LEVEL=5 specifies all levels of optimizations including loop transformation and software pipelining.

Table 5.4. Types of Optimization Performed at Different Levels /OPTIMIZE=LEVEL=n Levels

Optimization Type	n=0	n=1	n=2	n=3	n=4	n=5
Loop transformation						•
Software pipelining					•	•
Automatic inlining					•	•
Loop unrolling				•	•	•
Additional global optimizations				•	•	•
Global optimizations			•	•	•	•
Local (minimal) optimizations		•	•	•	•	•

The default is /OPTIMIZE=LEVEL=4.

In Table 5.4, the following terms are used to describe the levels of optimization (described in detail in Section 5.7.1 to Section 5.7.6):

- **Local (minimal) optimizations** (/OPTIMIZE=LEVEL=1 or higher) occur within the source program unit and include recognition of common subexpressions and the expansion of multiplication and division.
- **Global optimizations** (/OPTIMIZE=LEVEL=2 or higher) include such optimizations as data-flow analysis, code motion, strength reduction, split-lifetime analysis, and instruction scheduling.

- **Additional global optimizations** (/OPTIMIZE=LEVEL=3 or higher) improve speed at the cost of extra code size. These optimizations include loop unrolling and code replication to eliminate branches.

- **Automatic inlining and Software pipelining** (/OPTIMIZE=LEVEL=4 or higher) applies interprocedure analysis and inline expansion of small procedures, usually by using heuristics that limit extra code, and software pipelining.

  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 enables the prefetching of data to reduce the impact of cache misses.

- **Loop transformation** (/OPTIMIZE=LEVEL=5 or higher) includes a group of loop transformation optimizations.

  The loop transformation optimizations apply to array references within loops and can apply to multiple nested loops. These optimizations can improve the performance of the memory system.

5.7.1. Optimizations Performed at All Optimization Levels

The following optimizations occur at any optimization level (0 through 5):

- 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. VSI Fortran space optimizations are as follows:

  - 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, VSI 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, VSI 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 4.

- 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 VSI Fortran RTL. The processing overhead of these calls can be most significant in implied-DO loops.

  If VSI 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 5.5.8). The optimized RTL routine can transfer many elements in one operation.

  VSI 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, VSI 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 5.7.4.1).

5.7.2. Local (Minimal) Optimizations

To enable local optimizations, use /OPTIMIZE=LEVEL=1 or a higher optimization level (LEVEL=2, LEVEL=3, LEVEL=4, LEVEL=5).

To prevent local optimizations, specify /NOOPTIMIZE (/OPTIMIZE=LEVEL=0).

5.7.2.1. Common Subexpression Elimination

If the same subexpressions appear in more than one computation and the values do not change between computations, VSI 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. VSI Fortran eliminates this redundancy by producing the following:

```
t = ((J-1)*25+(I-1)*4
A(t) = B(t)
```

5.7.2.2. 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 VSI 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.

5.7.2.3. Compile-Time Operations

VSI Fortran does as many operations as possible at compile time rather than having them done at run time.

Constant Operations

VSI 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

VSI 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
```

5.7.2.4. Value Propagation

VSI Fortran tracks the values assigned to variables and constants, including those from DATA statements, and traces them to every place they are used. VSI Fortran uses the value itself when it is more efficient to do so.

When compiling subprograms, VSI 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. VSI 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`

5.7.2.5. Dead Store Elimination

If a variable is assigned but never used, VSI 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 VSI Fortran, these programs may show unrealistically good performance results. Realistic results are possible only with program units using their results in output statements.

5.7.2.6. 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, VSI 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

VSI 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, VSI Fortran generates code that uses 64-bit integer and floating-point registers instead of memory locations. Knowing when VSI Fortran uses registers may be helpful when doing certain forms of debugging.

VSI 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 /NOOPTIMIZE (same as /OPTIMIZE=LEVEL=0 or no optimization) was specified.

VSI 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
```

VSI Fortran might 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 stored.

VSI 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.

5.7.2.7. Mixed Real/Complex Operations

In mixed REAL/COMPLEX operations, VSI 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
```

5.7.3. Global Optimizations

To enable global optimizations, use /OPTIMIZE=LEVEL=2 or a higher optimization level (LEVEL=3, LEVEL=4, or LEVEL=5). Using /OPTIMIZE= LEVEL=2 or higher also enables local optimizations (LEVEL=1).

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 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, VSI 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, VSI 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. VSI Fortran limits or may disqualify data items that participate in the following constructs, generally because it cannot fully trace their values.

Data items in the following constructs can make global optimizations less effective:

- 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 90/95 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 example, if a variable in COMMON can change value by means of an OpenVMS AST, you must declare the variable or the COMMON block to which it belongs as volatile.
  - Variables read or written by an AST routine or a condition handler, including those in a common block or module.
  - An address not saved by the %LOC built-in function.

  As requested by the VOLATILE statement, VSI Fortran disqualifies any volatile variables from global data analysis.

- Subroutine calls or external function references

  VSI Fortran cannot trace data flow in a called routine that is not part of the program unit being compiled, unless the same FORTRAN 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

  VSI 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 VSI 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 90/95 modules

  VSI Fortran limits optimizations on variables in Fortran 90/95 modules. Like common blocks, if the variables in Fortran 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

  VSI Fortran limits optimizations on variables indirectly referenced by a %LOC function or variables with the TARGET attribute, because the called routine may dereference the 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. VSI 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.

5.7.4. Additional Global Optimizations

To enable additional global optimizations, use /OPTIMIZE=LEVEL=3 or a higher optimization level (LEVEL=4 or LEVEL=5). Using /OPTIMIZE= LEVEL=3 or higher also enables local optimizations (LEVEL=1) and global optimizations (LEVEL=2).

Additional global optimizations improve speed at the cost of longer compile times and possibly extra code size.

5.7.4.1. Loop Unrolling

At optimization level /OPTIMIZE=LEVEL=3 or above, VSI 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 number of times a loop is unrolled can be determined either by the optimizer or by using the /OPTIMIZE=UNROLL= `n` qualifier, which can specify the limit for loop unrolling. Unless the user specifies a value, the optimizer unrolls a loop four times for most loops or two times for certain loops (large estimated code size or branches out the loop).

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
```

5.7.4.2. 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 (R4 is register 4):

Original Code	**Optimized Code**

`.`
`.`
`.`
`branch to exit1`
`.`
`.`
`.`
`branch to exit1`
`.`
`.`
`.`
`exit1: move 1 into R4`
`return`
	`.`
	`.`
	`.`
	`move 1 into R4`
	`return`
	`.`
	`.`
	`.`
	`move 1 into R4`
	`return`
	`.`
	`.`
	`.`
	`move 1 into R4`
	`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.

5.7.5. Automatic Inlining and Software Pipelining

To enable optimizations that perform automatic inlining and software pipelining, use /OPTIMIZE=LEVEL=4 or a higher optimization level (LEVEL=5). Using /OPTIMIZE=LEVEL=4 also enables local optimizations (LEVEL=1), global optimizations (LEVEL=2), and additional global optimizations (LEVEL=3).

The default is /OPTIMIZE=LEVEL=4 (same as /OPTIMIZE).

5.7.5.1. Interprocedure Analysis

Compiling multiple source files at optimization level /OPTIMIZE=LEVEL=4 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.

5.7.5.2. 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:

- The /OPTIMIZE=LEVEL= `n` qualifier to control the optimization level. For example, specifying /OPTIMIZE=LEVEL=4 or higher enables interprocedure optimizations.

  Different /OPTIMIZE=LEVEL= `n` keywords set different levels of inlining. For example, /OPTIMIZE=LEVEL=4 sets /OPTIMIZE=INLINE=SPEED.

- One of the /OPTIMIZE=INLINE=`xxxxx` keywords to directly control the inlining of procedures (see Section 5.8.5). For example, /OPTIMIZE=INLINE=SPEED inlines more procedures than /OPTIMIZE=INLINE=SIZE.

5.7.5.3. Software Pipelining

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 enables the prefetching of data to reduce the impact of cache misses.

For More Information:

- On VSI Fortran statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On controlling inlining using /OPTIMIZE=INLINE= `keyword`, see Section 5.8.5.
- On software pipelining, see Section 5.8.2.

5.7.6. Loop Transformation

A group of optimizations known as loop transformation optimizations with its associated additional software dependence analysis are enabled by using the /OPTIMIZE=LEVEL=5 qualifier. In certain cases, this improves run-time performance.

The loop transformation optimizations apply to array references within loops and can apply to multiple nested loops. These optimizations can improve the performance of the memory system.

For More Information:

On loop transformations, see Section 5.8.1.

5.8. Other Qualifiers Related to Optimization

In addition to the /OPTIMIZE=LEVEL qualifiers (discussed in Section 5.7), several other FORTRAN command qualifiers and /OPTIMIZE keywords can prevent or facilitate improved optimizations.

5.8.1. Loop Transformation

The loop transformation optimizations are enabled by using the /OPTIMIZE=LOOPS qualifier or the /OPTIMIZE=LEVEL=5 qualifier. 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 /OPTIMIZE=LEVEL=5 with /OPTIMIZE=NOPIPELINE (preferred method)
- Specify /OPTIMIZE=LOOPS with /OPTIMIZE=LEVEL=4, LEVEL=3, or LEVEL=2. This optimization is not performed at optimization levels below LEVEL=2.

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 before entering the loop. For example, most DO loops are counted loops.

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 /OPTIMIZE=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.

For More Information:

On the interaction of command-line options and timing programs compiled with the loop transformation optimizations, see Section 5.7.

5.8.2. Software Pipelining

Software pipelining and additional software dependence analysis are enabled by using the /OPTIMIZE=PIPELINE qualifier or by the /OPTIMIZE=LEVEL=4 qualifier. Software pipelining in certain cases improves run-time performance.

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.

Loop unrolling (enabled at /OPTIMIZE=LEVEL=3 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 also enables the prefetching of data to reduce the impact of cache misses.

On Alpha systems, software pipelining can be more effective when you combine /OPTIMIZE=PIPELINE (or /OPTIMIZE=LEVEL=4) with the appropriate OPTIMIZE=TUNE= `keyword` (Alpha only) for the target Alpha processor generation (see Section 5.8.6).

To specify software pipelining without loop transformation optimizations, do one of the following:

- Specify /OPTIMIZE=LEVEL=4 (preferred method)
- Specify /OPTIMIZE=PIPELINE with /OPTIMIZE=LEVEL=3, or /OPTIMIZE=LEVEL=2. This optimization is not performed at optimization levels below LEVEL=2.

For this version of VSI Fortran, 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 /OPTIMIZE=LEVEL=4 (or /OPTIMIZE=PIPELINE) may not improve performance, as compared to using /OPTIMIZE=(LEVEL=4,NOPIPELINE).

For programs that contain loops that exhaust available registers, longer execution times may result with /OPTIMIZE=LEVEL=4 or /OPTIMIZE=PIPELINE. In cases where performance does not improve, consider compiling with the OPTIMIZE=UNROLL=1 qualifier along with /OPTIMIZE=LEVEL=4 or /OPTIMIZE=PIPELINE, to possibly improve the effects of software pipelining.

For More Information:

On the interaction of command-line options and timing programs compiled with software pipelining, see Section 5.7.

5.8.3. Setting Multiple Qualifiers with the /FAST Qualifier

Specifying the /FAST qualifier sets the following qualifiers:

- /ALIGNMENT=(COMMONS=NATURAL,RECORDS=NATURAL,SEQUENCE) (see Section 5.3)
- /ARCHITECTURE=HOST (see Section 5.8.7)
- /ASSUME=NOACCURACY_SENSITIVE (see Section 5.8.8)
- /MATH_LIBRARY=FAST (Alpha only) (see Section 2.3.30)
- /OPTIMIZE=TUNE=HOST (Alpha only) (see Section 5.8.6)

You can specify individual qualifiers on the command line to override the /FAST defaults. Note that /FAST/ALIGNMENT=COMMONS=PACKED sets /ALIGNMENT=NOSEQUENCE.

5.8.4. Controlling Loop Unrolling

You can specify the number of times a loop is unrolled by using the /OPTIMIZE= UNROLL=`n` qualifier (see Section 2.3.35).

Using /OPTIMIZE=UNROLL= `n` can also influence the run-time results of software pipelining optimizations performed when you specify /OPTIMIZE=LEVEL=5.

Although unrolling loops usually improves run-time performance, the size of the executable program may increase.

For More Information:

On loop unrolling, see Section 5.7.4.1.

5.8.5. Controlling the Inlining of Procedures

To specify the types of procedures to be inlined, use the /OPTIMIZE=INLINE= `keyword` keywords. Also, compile multiple source files together and specify an adequate optimization level, such as /OPTIMIZE=LEVEL=4.

If you omit /OPTIMIZE= INLINE= `keyword`, the optimization level /OPTIMIZE=LEVEL= `n` qualifier used determines the types of procedures that are inlined.

The /OPTIMIZE=INLINE= `keyword` keywords are as follows:

- NONE (same as /OPTIMIZE=NOINLINE) inlines statement functions but not other procedures. This type of inlining occurs if you specify /OPTIMIZE=LEVEL=0, LEVEL=1, LEVEL=2, or LEVEL=3 and omit INLINE= `keyword`.
- MANUAL (same as NONE) inlines statement functions but not other procedures. This type of inlining occurs if you specify /OPTIMIZE=LEVEL=2 or LEVEL=3 and omit INLINE= `keyword`.
- In addition to inlining statement functions, SIZE inlines any procedures that the VSI Fortran optimizer expects will improve run-time performance with no likely significant increase in program size.
- In addition to inlining statement functions, SPEED inlines any procedures that the VSI Fortran optimizer expects will improve run-time performance with a likely significant increase in program size. This type of inlining occurs if you specify /OPTIMIZE=LEVEL=4 or LEVEL=5 and omit /OPTIMIZE=INLINE= `keyword`.
- ALL inlines every call that can possibly be inlined while generating correct code, including the following:
  - Statement functions (always inlined).
  - Any procedures that VSI 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.

For information on the inlining of other procedures (inlined at optimization level /OPTIMIZE=LEVEL=4 or higher), see Section 5.7.5.2.

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 /OPTIMIZE=INLINE=ALL benefits your particular program, time program execution for the same program compiled with and without /OPTIMIZE=INLINE=ALL.

5.8.6. Requesting Optimized Code for a Specific Processor Generation (Alpha only)

You can specify the types of optimized code to be generated by using the /OPTIMIZE=TUNE= `keyword` (Alpha only) keywords. 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 /OPTIMIZE=TUNE= `keyword` (Alpha only) 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 /OPTIMIZE=TUNE= `keyword` (such as using /OPTIMIZE=TUNE=EV4 for an EV5 processor) is usually less than 5%. When using software pipelining (using /OPTIMIZE=LEVEL=5) with /OPTIMIZE=TUNE= `keyword`, the difference can be more than 5%.

The combination of the specified keyword for /OPTIMIZE=TUNE= `keyword` and the type of processor generation used has no effect on producing the expected correct program results.

The /OPTIMIZE=TUNE= `keyword` keywords are as follows:

- GENERIC generates and schedules code that will execute well for all types of Alpha processor generations. This provides generally efficient code for those applications that will be run on systems using all types of processor generations (an alternative to providing multiple versions of the application compiled for each processor generation type).
- HOST generates and schedules code optimized for the type of processor generation in use on the system being used for compilation.
- EV4 generates and schedules code optimized for the EV4 (21064) processor generation.
- EV5 generates and schedules code optimized for the EV5 (21164) processor generation. This processor generation is faster than EV4.
- EV56 generates and schedules code optimized for some 21164 Alpha architecture implementations that use the BWX (Byte/Word manipulation) instruction extensions of the Alpha architecture.
- PCA56 generates and schedules code optimized for 21164PC Alpha architecture implementation that uses BWX (Byte/Word manipulation) and MAX (Multimedia) instructions extensions.
- EV6 generates and schedules code for the 21264 chip implementation that uses the following extensions to the base Alpha instruction set: BWX (Byte/Word manipulation) and MAX (Multimedia) instructions, square root and floating-point convert instructions, and count instructions.
- EV67 generates and schedules code optimized for the EV67 processor generation. This processor generation is faster than EV4, EV5, EV56, PCA56, and EV6.

If you omit /OPTIMIZE=TUNE= `keyword`, if /FAST is specified, then HOST is used; otherwise, GENERIC is used.

5.8.7. Requesting Generated Code for a Specific Processor Generation (Alpha only)

You can specify the types of instructions that will be generated for the program unit being compiled by using the /ARCHITECTURE qualifier. Unlike the /OPTIMIZE=TUNE= *keyword* (Alpha only) option that helps with proper instruction scheduling, the /ARCHITECTURE qualifier specifies the type of Alpha chip instructions that can be used.

Programs compiled with the /ARCHITECTURE=GENERIC option (default) run on all Alpha processors without instruction emulation overhead.

For example, if you specify /ARCHITECTURE=EV6, the code generated will run very fast on EV6 systems, but may run slower on older Alpha processor generations. Because instructions used for the EV6 chip may be present in the program's generated code, code generated for an EV6 system may slow program execution on older Alpha processors when EV6 instructions are emulated by the OpenVMS Alpha Version 7.1 (or later) instruction emulator.

This instruction 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.

The keywords used by /ARCHITECTURE= `keyword` are the same as those used by /OPTIMIZE=TUNE= `keyword`. If you omit /ARCHITECTURE= *keyword*, if /FAST is specified then HOST is used; otherwise, GENERIC is used. For more information on the /ARCHITECTURE qualifier, see Section 2.3.6.

5.8.8. Arithmetic Reordering Optimizations

If you use the /ASSUME=NOACCURACY_SENSITIVE qualifier, VSI 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 (ACCURACY_SENSITIVE) because of the way intermediate results are rounded. However, the NOACCURACY_SENSITIVE results are not categorically less accurate than those gained by the default. In fact, dot product summations using NOACCURACY_SENSITIVE can produce more accurate results than those using ACCURACY_SENSITIVE.

The effect of /ASSUME=NOACCURACY_SENSITIVE is important when VSI Fortran hoists divide operations out of a loop. If NOACCURACY_SENSITIVE is in effect, the unoptimized loop becomes the optimized loop:

Unoptimized Code	**Optimized Code**

`DO I=1,N`
`.`
`.`
`.`
`B(I) = A(I)/V`
`END DO`
	`T = 1/V`
	`DO I=1,N`
	`.`
	`.`
	`.`
	`B(I) = A(I)*T`
	`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.

5.8.9. Dummy Aliasing Assumption

Some programs compiled with VSI Fortran (or Compaq Fortran 77) might have results that differ from the results of other Fortran compilers. Such programs might 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. Alternatively, they may be calling a user-defined procedure with actual arguments that do not match the procedure's dummy arguments in order, number, or type.

This program behavior is prohibited in programs conforming to the Fortran 90 and Fortran 95 standards, but not by VSI Fortran. Other versions of Fortran allow dummy aliases and check for them to ensure correct results. However, VSI Fortran assumes that no dummy aliasing will occur, and it can ignore potential data dependencies from this source in favor of faster execution.

The VSI Fortran default is safe for programs conforming to the Fortran 90 and Fortran 95 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 qualifier 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 5.3 is taken from the DAXPY routine in the Fortran-77 version of the Basic Linear Algebra Subroutines (BLAS).

Example 5.3. Using the /ASSUME=DUMMY_ALIASES Qualifier

```
      SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)

!     Constant times a vector plus a vector.
!     uses unrolled loops for increments equal to 1.

      DOUBLE PRECISION DX(1), DY(1), DA
      INTEGER I,INCX,INCY,IX,IY,M,MP1,N
!
      IF (N.LE.0) RETURN
      IF (DA.EQ.0.0) RETURN
      IF (INCX.EQ.1.AND.INCY.EQ.1) GOTO 20

!     Code for unequal increments or equal increments
!     not equal to 1.
      .
      .
      .
      RETURN
!     Code for both increments equal to 1.
!     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 qualifier 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 qualifier.

Programs calling DAXPY with DA overlapping DY do not conform to the FORTRAN-77, Fortran 90, and Fortran 95 standards. However, they are supported if /ASSUME=DUMMY_ALIASES was used to compile the DAXPY routine.

5.9. Compiler Directives Related to Performance

Certain compiler source directives (cDEC$ prefix) can be used in place of some performance-related compiler options and provide more control of certain optimizations, as discussed in the following sections:

5.9.1. Using the cDEC$ OPTIONS Directive

The cDEC$ OPTIONS directive allows source code control of the alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

Using this directive is an alternative to the compiler option /[NO]ALIGNMENT, which affects the alignment of all fields in record structures and data items in common blocks in the current program unit.

For more information:

See the description of the OPTIONS directive in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.9.2. Using the cDEC$ UNROLL Directive to Control Loop Unrolling

The cDEC$ UNROLL directive allows you to specify the number of times certain counted DO loops will be unrolled. Place the cDEC$ UNROLL directive before the DO loop you want to control the unrolling of.

Using this directive for a specific loop overrides the value specified by the compiler option /OPTIMIZE=UNROLL= for that loop. The value specified by *unroll* affects how many times all loops not controlled by their respective cDEC$ UNROLL directives are unrolled.

For more information:

See the description of the UNROLL directive in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

5.9.3. Using the cDEC$ IVDEP Directive to Control Certain Loop Optimizations

The cDEC$ IVDEP directive allows you to help control certain optimizations related to dependence analysis in a DO loop. Place the cDEC$ IVDEP directive before the DO loop you want to help control the optimizations for. Not all DO loops should use this directive.

The cDEC$ IVDEP directive tells the optimizer to begin dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

For more information:

See the description of the IVDEP directive in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

Chapter 6. VSI Fortran Input/Output

The following topics are addressed in this chapter:

- Section 6.1: Overview
- Section 6.2: Logical I/O Units
- Section 6.3: Types of I/O Statements
- Section 6.4: Forms of I/O Statements
- Section 6.5: Types of Files and File Characteristics
- Section 6.6: Opening Files and the OPEN Statement
- Section 6.7: Obtaining File Information: The INQUIRE Statement
- Section 6.8: Closing a File: The CLOSE Statement
- Section 6.9: Record Operations
- Section 6.10: Output Data Buffering and RMS Journaling

6.1. Overview

This chapter describes VSI Fortran input/output (I/O) as implemented for VSI Fortran. It also provides information about VSI Fortran I/O in relation to the OpenVMS Record Management Services (RMS) and Run-Time Library (RTL).

VSI Fortran assumes that all unformatted data files will be in the same native little endian numeric formats used in memory. If you need to read or write unformatted numeric data (on disk) that has a different numeric format than that used in memory, see Chapter 9.

You can use VSI Fortran I/O statements to communicate between processes on either the same computer or different computers.

For More Information:

- On specifying the native floating-point format used in memory, see Section 2.3.22.
- On supported data types, see Chapter 8.
- On using various VSI and non-VSI data formats for unformatted files, see Chapter 9.
- On interprocess communication, see Chapter 13.
- On porting Compaq Fortran 77 for OpenVMS VAX data, see Appendix B.
- On performing I/O to the same unit with VSI Fortran and Compaq Fortran 77 object files, see Appendix B.
- On using indexed files, see Chapter 12.

6.2. Logical I/O Units

In VSI 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, the following READ statement uses logical unit number 2:

```
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. You can use the LIB$GET_LUN library routine to return a logical unit number not currently in use by your program. If you intend to use LIB$GET_LUN, avoid using logical unit numbers (UNIT) 100 to 119 (reserved for LIB$GET_LUN).

VSI 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 command procedure can set the appropriate logical name or allow the terminal user to type a directory, 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).

ACCEPT, TYPE, and PRINT statements do not refer explicitly to a logical unit (a file or device) from which or to which data is to be transferred; they refer implicitly to a default preconnected logical unit. The ACCEPT statement is normally preconnected to the default input device, and the TYPE and PRINT statements are normally preconnected to the default output device. These defaults can be overridden with appropriate logical name assignments (see Section 6.6.1.2).

READ, WRITE, and REWRITE statements refer explicitly to a specified logical unit from which or to which data is to be transferred. However, to use a preconnected device for READ (SYS$INPUT) and WRITE (SYS$OUTPUT), specify the unit number as an asterisk (*).

Certain unit numbers are **preconnected** to OpenVMS standard devices. Unit number 5 is associated with SYS$INPUT and unit 6 with SYS$OUTPUT. 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, VSI Fortran implicitly opens units 5 and 6 and associates them with their respective operating system standard I/O files.

For More Information:

On the OPEN statement and preconnected files, see Section 6.6.

6.3. Types of I/O Statements

Table 6.1 lists the VSI Fortran I/O statements.

Table 6.1. 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 SYS$OUTPUT (standard output device). Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.
VSI Fortran Extensions	
ACCEPT	Reads input from SYS$INPUT. Unlike READ, ACCEPT only provides formatted sequential output and does not specify a unit number.
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. Certain restrictions apply.
UNLOCK	Releases a lock held on the current record when file sharing was requested when the file was opened (see Section 6.9.2).
TYPE	Writes record output to SYS$OUTPUT (same as PRINT).
DEFINE FILE	Specifies file characteristics for a direct access relative file and connects the unit number to the file (like an OPEN statement). Provided for compatibility with compilers older than 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 and indexed files.
- The BACKSPACE and REWIND statements only apply to sequential files open for sequential access.

- The ENDFILE statement only applies to certain types of sequential files open for sequential access.

The file-related statements (OPEN, INQUIRE, and CLOSE) apply to any relative or sequential file.

6.4. 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 6.9.2, you can access records in any sequential, relative, or indexed file using sequential access. For relative files and fixed-length sequential files, you can also access records using direct access. For indexed files, you can use keyed access.

Table 6.2 shows the main record I/O statements, by category, that can be used in VSI Fortran programs.

Table 6.2. 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, TYPE[1], and REWRITE[1]
List-Directed	READ, WRITE, PRINT, ACCEPT[1], TYPE[1], and REWRITE[1]
Namelist	READ, WRITE, PRINT, ACCEPT[1], TYPE[1], and REWRITE[1]
Unformatted	READ, WRITE, and REWRITE[1]
External file, direct access	
Formatted	READ, WRITE, and REWRITE[1]
Unformatted	READ, WRITE, and REWRITE[1]
External file, keyed access	
Formatted	READ, WRITE, and REWRITE[1]
Unformatted	READ, WRITE, and REWRITE[1]
Internal file[2]	
Formatted	READ, WRITE
List-Directed	READ, WRITE
Unformatted	None

[1]This statement is a VSI extension to the Fortran 90 and Fortran 95 standards.

[2]An internal file is a way to reference character data in a buffer using sequential access (see Section 6.5.2).

6.5. Types of Files and File Characteristics

This section discusses file organization, internal and scratch files, record type, record length, and other file characteristics.

6.5.1. File Organizations

File organization refers to the way records are physically arranged on a storage device.

The default fileorganization is always ORGANIZATION= 'SEQUENTIAL' for an OPEN statement.

VSI Fortran supports three kinds of file organizations: sequential, relative, and indexed sequential. The organization of a file is specified by means of the ORGANIZATION specifier in the OPEN statement.

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 and line printers, as sequential files.

File characteristics, including the file organization and record type, are stored by RMS in the disk file header and can be obtained by using the INQUIRE statement. You can also view the organization of a file using the DCL command DIRECTORY/FULL.

For more information on the INQUIRE statement, see Section 6.7 and the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

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 stored on disk with a fixed-length record type 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).

Relative files allow you to use direct access and detect when a record has been deleted.

When creating a relative file, specify the RECL value to determine the size of the fixed-length cells. Within the cells, you can store variable-length records as long as their size does not exceed the cell size.

Indexed Files

An indexed file consists of two or more separate sections: one section contains the data records and the other sections contain indexes. When an indexed file is created, each index is associated with a specification defining a field within each record, called a key field or simply **key**. A record in an indexed file must contain at least one key, called the **primary key**, which determines the location of the records within the body of the file.

The keys of all records are collected to form one or more structured indexes, through which records are always accessed. The structure of the indexes allows a program to access records in an indexed file either randomly (keyed access) or sequentially (sequential access). With keyed access, you specify a particular key value. With sequential access, you retrieve records with increasing or decreasing key values. You can mix keyed access and sequential access.

Indexed files are supported only on disk devices. When creating an indexed file, specify the RECL value.

For more information on indexed files, see Chapter 12.

6.5.2. Internal Files and Scratch Files

VSI Fortran also supports two other types of files that are not file organizations — namely, internal files and scratch files.

Internal Files

You can use an internal file to reference character data in a buffer when using sequential access. 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 such 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 variable, array element, or substring. 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' on an OPEN statement. By default, the files are created on the user's default disk (SYS$DISK) and are not placed in a directory or given a name that is externally visible (accessible using the DCL command DIRECTORY).

You can create scratch files on a disk other than the default disk by using the FILE specifier in an OPEN statement.

6.5.3. I/O Record Types

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, relative, or indexed files. You can use any of the record types with sequential files.

Records are stored in one of the following record types:

- Fixed-length
- Variable-length

- Segmented
- Stream
- Stream_CR
- Stream_LF

You can use fixed-length and variable-length record types with sequential, relative, or indexed files.

Before you choose a record type, consider whether your application will use formatted or unformatted data. If you will be using formatted data, you can use any record type except segmented. When using unformatted data, you should avoid using the stream, stream_CR, and stream_LF record types.

The segmented record type is unique to VSI Fortran products; it is not used by other OpenVMS-supported languages. It 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 Fortran.

The stream, stream_CR, stream_LF, and segmented record types can only be used with sequential files.

6.5.3.1. Portability Considerations of Record Types

Consider the following portability needs when choosing a record type:

- Data files from VSI Fortran and Compaq Fortran 77 on OpenVMS systems are interchangeable.
- You can use any any record type except segmented with other non-Fortran OpenVMS languages.

VSI Fortran indexed files are portable only to other OpenVMS systems. However, a conversion program can read the records from an indexed file and write them to another file, such as a sequential (or relative) file.

6.5.3.2. Fixed-Length Records

When you create a file that uses the fixed-length record type, you must specify the record size. When you specify fixed-length records, all records in the file must contain the same number of bytes. (The *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] discusses fixed-length records).

A sequential file opened for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

You can obtain the record length (RECL) before opening the file with unformatted data by using a form of the INQUIRE statement (see Section 6.7.3).

6.5.3.3. Variable-Length Records

Variable-length records can contain any number of bytes, up to a specified maximum. These records are prefixed by a count field, indicating the number of bytes in the record. The count field comprises two bytes on a disk device and four bytes on magnetic tape. The value stored in the count field indicates the number of data bytes in the record.

Variable-length records in relative files are actually stored in fixed-length cells, the size of which must be specified by means of the RECL specifier in an OPEN statement (see the *VSI Fortran Reference Manual*

[https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] for details). This RECL value specifies the largest record that can be stored in the file.

The count field of a formatted 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.

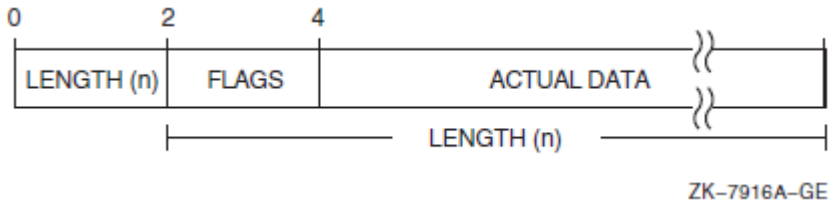
6.5.3.4. Segmented Records

A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequential file. Each variable-length record constitutes a segment. The length of a segmented record is arbitrary.

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. When writing unformatted data to a sequential file using sequential access, the default record type is segmented.

As shown in Figure 6.1, the layout of segmented records consists of control information followed by the user data. On disk, the control information consists of four bytes for compatibility with other VSI Fortran platforms. However, OpenVMS RMS removes the first two length bytes when the record is read, so each record has two control bytes (flags) in memory.

Figure 6.1. 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.

When you wish to access an unformatted sequential file that contains variable-length records, you must specify FORM= ’UNFORMATTED ’ when you open the file. If the unformatted data file was not created using a VSI Fortran product, specify RECORDTYPE= ’VARIABLE ’. If the unformatted data file was created using the segmented record type using a VSI Fortran Fortran product, specify RECORDTYPE= ’SEGMENTED ’.

Otherwise, the first two bytes of each record will be mistakenly interpreted as control information, and errors will probably result.

You can obtain the record length (RECL) before opening the file with unformatted data using a form of the INQUIRE statement (see Section 6.7.3).

6.5.3.5. Stream Records

A stream record is a variable-length record whose length is indicated by explicit record terminators embedded in the data, not by a count.

Stream files use the 2-character sequence consisting of a carriage-return and a line-feed as the record terminator. These terminators are automatically added when you write records to a stream file and removed when you read records.

Stream records resemble the Stream_CR or Stream_LF records shown in Figure 6.2, but use a 2-byte record terminator (carriage-return and line-feed) instead of a 1-byte record terminator.

6.5.3.6. Stream_CR and Stream_LF Records

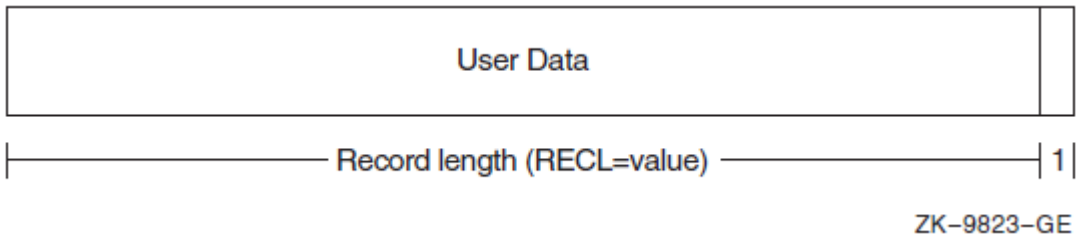
A Stream_CR or Stream_LF record is a variable-length record whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

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.

The layout of Stream_CR and Stream_LF records appears in Figure 6.2.

Figure 6.2. Stream_CR and Stream_LF Records



6.5.4. 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)
- Whether records can span block boundaries (NOSPANBLOCK specifier)
- For an indexed file being created, the key number, its location, and its data type (KEY specifier)

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 qualifier to request 1-byte units (see Section 2.3.7).

For More Information:

- On statement syntax and specifier values (including defaults), see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On file characteristics, see Section 6.6.3 and the OPEN statement in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On I/O performance considerations, see Section 5.5.

6.6. Opening Files and the OPEN Statement

You can choose to open files by:

- Using default values, such as a preconnected unit. In the following example, the PRINT statement is associated with a preconnected unit (SYS$OUTPUT) by default:

  ```
  PRINT *,100
  ```

  Implicitly opening a file by omitting an OPEN statement prevents you from specifying the file connection characteristics and other information provided by the OPEN statement. You might use implicit opening of a file for terminal I/O.

  The following READ statement associates the logical unit 7 with the file FOR007.DAT (because the FILE specifier was omitted) by default:

  ```
  OPEN (UNIT=7,STATUS='OLD')
  READ (7,100)
  ```

- Using default logical names, which allows you to specify which file or files are used at run time. If the following example uses an implicit OPEN, the READ statement causes the logical name FOR007 to be associated with the file FOR007.DAT by default. The TYPE statement causes the logical unit FOR$TYPE to be associated with SYS$OUTPUT by default.

  ```
  READ (7,100)
       .
       .
       .
     TYPE 100
  ```

- Using logical names without an OPEN statement. You can also use DCL commands to set the appropriate logical names to a value that indicates a directory (if needed) and a file name to associate a unit with an external file.
- Specifying a logical name in an OPEN statement. This allows you to specify which file or files are used at run time, but the appropriate logical names must be defined before the program is run. For example:

  ```
  OPEN (UNIT=7, FILE='LOGNAM', STATUS='OLD')
  ```

- Specifying a file specification in an OPEN statement. The file or files are specified at compile time, so the program may need to be recompiled to specify a different file (or the default device and directory changed if these are not specified by the program). For example:

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

If you choose to specify a logical name with the FILE specifier in an OPEN statement, that logical name must be associated with a file specification and the character expression specified for the logical name must contain no punctuation marks.

6.6.1. Preconnected Files and Fortran Logical Names

You can use OpenVMS logical names to associate logical units with file specifications. A logical name is a string up to 255 characters long that you can use as part of a file specification.

Table 6.3 lists the OpenVMS process logical names for standard I/O devices already associated with particular file specifications.

Table 6.3. Predefined System Logical Names

OpenVMS Logical Name	Meaning	Default
SYS$COMMAND	Default command stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job input command file.
SYS$DISK	Default disk device	As specified by user.
SYS$ERROR	Default error stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job log file.
SYS$INPUT	Default input stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch command file.
SYS$OUTPUT	Default output stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch log file.

You can dynamically create a logical name and associate it with a file specification by means of the DCL commands ASSIGN or DEFINE. For example, before program execution, you can define each logical name recognized by your program with a file specification appropriate to your needs without recompiling and relinking the program. For example:

```
$ DEFINE LOGNAM USERD:[SMITH]TEST.DAT;2
```

The preceding command creates the logical name LOGNAM and associates it with the file specification USERD:[SMITH]TEST.DAT;2. As a result, this file specification is used whenever the logical name LOGNAM is encountered during program execution.

The following statement opens the file associated with the current definition of the logical name LOGNAM:

```
OPEN (UNIT=7, FILE='LOGNAM', STATUS='OLD')
```

Logical names provide great flexibility because they can be associated with either a partial or complete file specification (with either a device or a device and a directory), or even another logical name.

6.6.1.1. Preconnected Files

ACCEPT, TYPE, and PRINT statements do not refer explicitly to a logical unit (a file or device) from which or to which data is to be transferred; they refer implicitly to a default preconnected logical unit. ACCEPT refers to the default input device SYS$INPUT. TYPE and PRINT refer to the default output device SYS$OUTPUT. These defaults can be overridden with appropriate logical name assignments (see Section 6.6.1.2).

READ, WRITE, and REWRITE usually refer to explicit unit numbers. If you do not use an OPEN statement to open logical unit 5 or 6 without setting the appropriate logical name (FOR `nnn`), unit number 5 is associated with SYS$INPUT and unit 6 with SYS$OUTPUT.

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, VSI Fortran implicitly opens units 5 and 6 and associates them with their respective operating system standard I/O files if the corresponding logical name is not set.

To redirect I/O to an external disk file instead of these preconnected files, you can either use an OPEN statement to unit 5 and 6 or set the appropriate logical name. If you set the corresponding VSI Fortran logical name, the file specified by that VSI Fortran logical name is used.

The order of precedence when you open a file is:

- When you explicitly open a preconnected file by using an OPEN statement with a file name for that unit, the Fortran logical name and OpenVMS standard I/O device are *not* used. The file is no longer considered preconnected.

  If the file name is not present in the OPEN statement, the unit is still preconnected as shown in Table 6.4.

- If the appropriate Fortran logical name *is* defined, its definition is used instead of the OpenVMS standard I/O logical name.

- If the Fortran logical name *is not* defined, the OpenVMS standard I/O logical name is used.

- For units not associated with a preconnected OpenVMS standard I/O device, if you omit the file name and file type, the system supplies certain defaults, such as a file name and type of FOR `nnn`.DAT (see Section 6.6.1.2.

Table 6.4 shows the I/O statements and their associated Fortran logical names and OpenVMS standard I/O logical names.

Table 6.4. Implicit Fortran Logical Units

Statement	Fortran Logical Name[1]	Equivalent OpenVMS Logical Name
READ (*,f) iolist	FOR$READ	SYS$INPUT
READ f,iolist	FOR$READ	SYS$INPUT
ACCEPT f,iolist	FOR$ACCEPT	SYS$INPUT
WRITE (*,f) iolist	FOR$PRINT	SYS$OUTPUT
PRINT f,iolist	FOR$PRINT	SYS$OUTPUT
TYPE f,iolist	FOR$TYPE	SYS$OUTPUT
READ (5),iolist	FOR005	SYS$INPUT
WRITE (6),iolist	FOR006	SYS$OUTPUT

[1]If the Fortran logical name is defined, it is used; if the Fortran logical name is not defined, the OpenVMS standard I/O logical names are used.

You can change the file specifications associated with these Fortran logical names by using the DCL commands DEFINE or ASSIGN.

6.6.1.2. VSI Fortran Logical Names

VSI Fortran I/O is usually performed by associating a logical unit number with a device or file. OpenVMS logical names provide an additional level of association; a user-specified logical name can be associated with a logical unit number.

VSI Fortran provides predefined logical names in the following form:

FOR*nnn*

The notation *nnn* represents a logical unit number, any non-negative 4-byte integer (maximum value is 2,147,483,647). For example, for logical unit 12, the predefined logical name would be FOR012; for logical unit 1024, the predefined logical name would be FOR1024.

By default, each Fortran logical name is associated with a file named FORnnn.DAT on your default disk under your default directory. For example:

```
WRITE (17,200)
```

If you enter the preceding statement without including an explicit file specification, the data is written to a file named FOR017.DAT on your default disk under your default directory.

You can change the file specification associated with a Fortran logical unit number by using the DCL commands ASSIGN or DEFINE to change the file associated with the corresponding Fortran logical name. For example:

```
$ DEFINE FOR017 USERD:[SMITH]TEST.DAT;2
```

The preceding command associates the Fortran logical name FOR017 (and therefore logical unit 17) with file TEST.DAT;2 on device USERD in directory [SMITH].

You can also associate the Fortran logical names with any of the predefined system logical names, as shown in the following examples:

- The following command associates logical unit 10 with the default output device (for example, the batch output stream):

  ```
  $ DEFINE FOR010 SYS$OUTPUT
  ```

- The following command associates the default command stream with the default input device (for example, the batch input stream):

  ```
  $ DEFINE SYS$COMMAND  SYS$INPUT
  ```

For More Information:

On the DCL commands you can use to assign or deassign logical names, see Appendix D.

6.6.2. Disk Files and File Specifications

Most I/O operations involve a disk file, keyboard, or screen display. You can access the terminal screen or keyboard by using preconnected files, as described in Section 6.6. Otherwise, this chapter discusses disk files.

VSI Fortran recognizes logical names for each logical I/O unit number in the form of FOR nnn, where nnn is the logical I/O unit number, with leading zeros for fewer than three digits. If a file name is not specified in the OPEN statement and the corresponding FOR nnn logical name is not set for that unit number, VSI Fortran generates a file name in the form FOR nnn.DAT, where n is the logical unit number.

Certain VSI Fortran logical names are recognized and preconnected files exist for certain unit numbers. Performing an implied OPEN means that the FILE and DEFAULTFILE specifier values are not specified and a logical name is used, if present.

A complete OpenVMS file specification has the form:

```
node::device:[directory]filename.filetype;version
```

For example:

```
BOSTON::USERD:[SMITH]TEST.DAT;2
```

You can associate a file specification with a logical unit by using a logical name assignment (see Section 6.6.1) or by using an OPEN statement (see Section 6.6.2). If you do not specify such an association or if you omit elements of the file specification, the system supplies default values, as follows:

- If you omit the node, the local computer is used.
- If you omit the device or directory, the current user default device or directory is used.
- If you omit the file name, the system supplies FOR nnn (where nnn is the logical unit number, with leading zeros for one- or two-digit numbers)
- If you omit the file type, the system supplies DAT.
- If you omit the version number, the system supplies either the highest current version number (for input) or the highest current version number plus 1 (for output).

For example, if your default device is USERD and your default directory is SMITH, and you specified the following statements:

```
READ (8,100)
    .
    .
    .
WRITE (9,200)
```

The default input file specification would be:

```
USERD:[SMITH]FOR008.DAT;n
```

The default output file specification would be:

```
USERD:[SMITH]FOR009.DAT;m
```

In these examples, *n* equals the highest current version number of FOR008.DAT and *m* is 1 greater than the highest existing version number of FOR009.DAT.

You can use the FILE and DEFAULTFILE specifiers in an OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. For example:

```
OPEN (UNIT=4, FILE='USERD:[SMITH]TEST.DAT;2', STATUS='OLD')
```

In the preceding example, the existing file TEST.DAT;2 on device USERD in directory SMITH is to be opened on logical unit 4. Neither the default file specification (FOR004.DAT) nor the Fortran logical name FOR004 is used. The value of the FILE specifier can be a character constant, variable, or expression.

VSI Fortran provides the following possible ways of specifying all or part of a file specification (directory and file name), such as DISK2:[PROJECT.DATA]:

- 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 DISK2:[PROJECT.DATA]TESTDATA).
- The DEFAULTFILE specifier (a VSI extension) in an OPEN statement typically specifies a device and/or directory without a file name or a device and/or directory with a file name (such as DISK2:[PROJECT. DATA]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 a logical name to specify a file name or a device and/or directory that contains both a directory and file name (see Section 6.6.1).

In the following interactive example, the file name is supplied by the user and the DEFAULTFILE specifier supplies the default values for the file specification string. The file to be opened is in device and directory DISK4:[PROJ] and is merged with the file name typed by the user into the variable DOC:

```
CHARACTER(LEN=40) DOC
WRITE (6,*)  'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC, DEFAULTFILE='DISK4:[PROJ]',STATUS='OLD')
```

The DEFAULTFILE specification overrides your process default device and directory.

You can also specify a logical name as the value of the FILE specifier, if the logical name is associated with a file specification. If the logical name LOGNAM is defined to be the file specification USERD:[SMITH]TEST.DAT, the logical name can then be used in an OPEN statement, as follows:

```
OPEN (UNIT=19, FILE='LOGNAM', STATUS='OLD')
```

When an I/O statement refers to logical unit 19, the system uses the file specification associated with logical name LOGNAM.

If the value specified for the FILE specifier has no associated file specification, it is regarded as a true file name rather than as a logical name. Suppose LOGNAM had not been previously associated with the file specification by using an ASSIGN or DEFINE command. The OPEN statement would indicate that a file named LOGNAM.DAT is located on the default device, in the default directory.

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 7.1.

For a detailed description of OpenVMS file specifications, see the *Guide to OpenVMS File Applications*.

For More Information:

- On a list of VSI Fortran I/O statements, see Table 6.1.
- On record I/O, see Section 6.9.

- On VSI Fortran I/O statements and specifier values, including defaults, see Table 6.1.
- On statement syntax, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the ERR and IOSTAT specifiers, see Chapter 7.
- On closing files, see Section 6.8.

6.6.3. 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 VSI Fortran run-time system closes the original file and then reopens the specified file.

  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 6.7) 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]) or explicitly specify file attributes with the appropriate OPEN statement specifiers.

The OPEN statement functions and their specifiers are:

- Identify File and Unit

  \UNIT specifies the logical unit number.

  \FILE (or NAME) 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, relative, or indexed).

[1] This specifier is a VSI Fortran extension.

\RECORDTYPE [1] indicates which record type to use.

\FORM indicates whether records are formatted or unformatted. See Section 6.5.4 and Section 6.4.

\CARRIAGECONTROL [1] indicates the terminal control type.

\KEY [1] indicates (when creating an indexed file) the key number, its type, and its location.

\NOSPANBLOCKS [1] indicates that the records should not span block boundaries.

\RECL or RECORDSIZE [1] specifies the record size. See Section 6.5.4.

- Special File Open Routine

  \USEROPEN [1] names the routine that will open the file to establish special context that changes the effect of subsequent VSI Fortran I/O statements (see Chapter 11).

- File Access, Processing, and Position

  \ACCESS indicates the access mode (direct, keyed, or sequential). See Section 6.9.2.

  \ACTION or READONLY [1] indicates whether statements will be used to only read records, only write records, or read *and* write records. See Section 6.9.3.

  \POSITION indicates whether to position the file at the beginning of file, before the end-of-file record, or leave it as is (unchanged). See Section 6.9.4.

  \SHARED [1] indicates that other users can access the same file and activates record locking. See Section 6.9.3.

  \MAXREC [1] specifies the maximum record number for direct access.

  \ASSOCIATEVARIABLE [1] specifies the variable containing next record number for direct access.

- File Allocation

  \INITIALSIZE [1] indicates the allocation unit (in blocks) when creating a file.

  \EXTENDSIZE [1] indicates the allocation unit (in blocks) when allocation additional file space.

- Record Transfer Characteristics

  \BLANK indicates whether to ignore blanks in numeric fields.

  \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 9.

- Error Handling Capabilities

\ERR specifies a label to branch to if an error occurs. See Chapter 7.

- \IOSTAT specifies the integer variable to receive the error (IOSTAT) number if an error occurs. See Chapter 7.
- File Close Action

  \DISPOSE [1] identifies the action to take when the file is closed.

For More Information:

- On specifier syntax and complete information, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the FORM specifier, see Section 6.4.
- On file organizations, see Section 6.5.1.
- On available record types, see Section 6.5.3.
- On the CARRIAGECONTROL specifier, see Section 6.5.4.
- On the RECL (record length) specifier, see Section 6.5.4.
- On shared access to files, see Section 6.9.3.
- On the ERR and IOSTAT specifiers, see Chapter 7.
- On obtaining file information using the INQUIRE statement, see Section 6.7.
- On closing files, see Section 6.8.
- Record I/O transfer, see Section 6.9.6.
- Record advancement, see Section 6.9.5.
- Record positioning, see Section 6.9.4.
- On I/O performance considerations, see Section 5.5.

6.7. Obtaining File Information: The 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

6.7.1. Inquiry by Unit

An inquiry by unit is usually done for an opened (connected) file. An inquiry by unit causes the VSI 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 file specification is 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 Section 6.6.3)
  - 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 qualifier to request 1-byte units (see Section 2.3.7).
- 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)
```

6.7.2. Inquiry by File Name

An inquiry by name causes the VSI 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 full file specification is 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 Section 6.6.3).
  - 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 qualifier to request 1-byte units (see Section 2.3.7).
- 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 full file specification.
- Any other information returned will be file characteristics maintained in the file header or 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)
```

6.7.3. 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 VSI Fortran extension).

The following INQUIRE statement returns the maximum record length of the variable list in integer 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 qualifier to request 1-byte units.

For More Information:

- On the INQUIRE statement and its specifiers, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On record I/O, see Section 6.9.
- On OPEN statement specifiers, see Section 6.6.3.
- On the /ASSUME=BYTERECL qualifier, see Section 2.3.7.

6.8. Closing a File: The 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'. (Other STATUS keyword values include 'SUBMIT' and 'PRINT').

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 VSI Fortran I/O statements, see Table 6.1.
- On changing default I/O characteristics before using VSI Fortran I/O statements, see Chapter 11.
- On OPEN statement specifiers, see Section 6.6.3.
- On statement syntax and specifier values (such as other STATUS values), see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

6.9. 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 VSI extensions.

These statements are described in Section 6.3 and the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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 6.4.

6.9.1. 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 the name of a NAMELIST group.
- For direct access, REC specifies a record number.
- For keyed access:
  - KEYID specifies the key of reference.
  - KEY, KEYNXT, KEYNXTNE, KEYLT, KEYEQ, KEYLE, and KEYGT specify the key value and key match characteristics.

When using nonadvancing I/O, use the ADVANCE, EOR, and SIZE specifiers, as described in Section 6.9.5.

When using the REWRITE statement (a VSI Fortran extension), you can use the UNIT, FMT, ERR, and IOSTAT specifiers.

For More Information

- On specifier syntax and complete information, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On available record types, see Section 6.5.3.
- On the error-related record I/O specifiers ERR, END, and IOSTAT, see Chapter 7.
- On the ADVANCE, EOR, and SIZE specifiers, see Section 6.9.5.
- On record positioning, see Section 6.9.4.
- On record I/O transfer, see Section 6.9.6.
- On record advancement, see Section 6.9.5.

6.9.2. Record Access Modes

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. Access mode is the method a program uses to retrieve and store records in a file. The access mode is specified as part of each I/O statement.

VSI Fortran supports three record access modes:

- Sequential access—transfers records sequentially to or from files (sequential, relative, or indexed) or I/O devices such as terminals.
- Direct access—transfers records selected by record number to and from fixed-length sequential files or relative organization files.

- Keyed access—transfers records to and from indexed files, based on data values (keys) contained in the records, using the current key-of-reference.

Your choice of record access mode is affected by the organization of the file to be accessed. For example, the keyed access mode can be used only with indexed organization files.

Table 6.5 shows all of the valid combinations of access mode and file organization.

Table 6.5. Valid Combinations of File Organization and Record Access Mode

File Organization	Sequential	Direct	Keyed
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

[1]Fixed-length records only.

6.9.2.1. Sequential Access

If you select the sequential access mode for sequential or relative files, records are written to or read from the file starting at the beginning and continuing through the file, one record after another. For indexed files, sequential access can be used to read or write all records according to the direction of the key and the key values. Sequential access to indexed files can also be used with keyed access to read or write a group of records at a specified point in the file.

When you use sequential access for sequential and relative files, a particular record can be retrieved only after all of the records preceding it have been read.

Writing records by means of sequential access also varies according to the file organization:

- For sequential files, new records can be written only at the end of the file.
- For relative files, a new record can be written at any point, replacing the existing record in the specified cell. For example, if two records are read from a relative file and then a record is written, the new record occupies cell 3 of the file.
- For indexed files, records must be written in primary key order, and READ operations refer to the next record meeting the key selection criteria.

6.9.2.2. Direct Access

If you select direct access mode, you 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 access relative files directly. You can also access a sequential disk file directly if it contains fixed-length records. Because direct access uses cell numbers to find records, you can issue successive READ or WRITE statements requesting records that either precede or follow previously requested records. The following statements read record 24, then read record 10.

```
READ (12,REC=24) I
READ (12,REC=10) J
```

6.9.2.3. Keyed Access

If you select keyed access mode, you determine the order in which records are read or written by means of character values or integer values called keys. Each READ statement contains the key that locates the record. The key value in the I/O statement is compared with index entries until the record is located.

When you insert a new record, the values contained in the key fields of the record determine the record's placement in the file; you do not have to indicate a key.

You can use keyed access only for indexed files.

Your program can mix keyed access and sequential access I/O statements on the same file. You can use keyed I/O statements to position the file to a particular record, then use sequential I/O statements to access records with either increasing or decreasing key values (depending on the key chosen).

For More Information

On using indexed files, see Chapter 12.

6.9.3. Shared File Use

With the RMS file-sharing capability, you can allow file access by more than one program at a time or by the same program on more than one logical unit. There are two kinds of file sharing:

- Read sharing occurs when multiple programs are reading a file at the same time.
- Write sharing takes place when at least one program is writing a file and at least one other program is either reading or writing the same file.

All three file organizations – relative, indexed, and sequential – permit read and write access to shared files.

The extent to which file sharing can take place is determined by two factors: the type of device on which the file resides and the explicit information supplied by the user. These factors have the following effects:

- Device type – Sharing is possible only on disk files.
- Explicit file-sharing information supplied by accessing programs – Whether file sharing actually takes place depends on information provided to OpenVMS RMS by each program accessing the file. In VSI Fortran programs, this information is supplied by the ACTION specifier (or VSI extension READONLY specifier) and the SHARED specifier in the OPEN statement.

  Read sharing is accomplished when the OPEN statement specifies the ACTION='READ' (or READONLY) specifier by all programs accessing the file.

  Write sharing is accomplished when the program specifies SHARED with either ACTION='WRITE' or ACTION='READWRITE' (the default is ACTION='READWRITE' unless you specify ACTION='READ' or READONLY).

  Programs that specify ACTION='READ' (or READONLY) or 'SHARED' can access a file simultaneously, with the exception that a file opened for ACTION='READ' (READONLY) cannot be accessed by a program that specifies SHARED.

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, unless the OPEN statement specifies the SHARED specifier.

To allow other users to access the same file once you have opened it, specify the OPEN statement SHARED specifier when you open the file. If you specify the SHARED specifier when opening a file that is already opened by another process, your program will be allowed to access the file.

If you omit the SHARED specifier when opening a file that is already opened by another process, your program will be denied access to the file. If you omit the SHARED specifier and are the first to open that files, file access might be denied to other users later requesting access to the file.

For performance reasons, when writing records to the file, avoid specifying the SHARED qualifier when you are certain that no other processes will access that file. Similarly, unless you will be writing records when specifying the SHARED qualifier, specify ACTION='READ'.

When two or more programs are write sharing a file, each program should use one of the error-processing mechanisms described in Chapter 14.

Use of one of these controls, the RMS record-locking facility, prevents program failure due to a record-locking error.

The RMS record-locking facility, along with the logic of the program, prevents two processes from accessing the same record at the same time. Record locking ensures that a program can add, delete, or update a record without having to check whether the same record is simultaneously being accessed by another process.

When a program opens a relative, sequential, or indexed file specifying SHARED, RMS locks each record as it is accessed. When a record is locked, any program attempting to access it fails with a record-locked error. A subsequent I/O operation on the logical unit unlocks the previously accessed record, so no more than one record on a logical unit is ever locked.

In the case of a WRITE to a sequential or relative organization file opened for shared access, VSI Fortran uses an RMS option that causes the record to be updated if it already exists in the file. This option has the side-effect of momentarily releasing the record lock, if any, and then relocking the target record. There is a small possibility that if another program is trying to access the same record at the same time, it may succeed in locking the record while it is unlocked by the first program, resulting in a record-locked error for the WRITE statement.

Locked records can be explicitly unlocked by means of VSI Fortran's UNLOCK statement. The use of this statement minimizes the amount of time that a record is locked against access by other programs. The UNLOCK statement should be used in programs that retrieve records from a shared file but do not attempt to update them.

For More Information

- On the UNLOCK statement and its syntax, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On record locking for shared files, see the *Guide to OpenVMS File Applications*.
- On how to handle record locking for indexed files, see Section 12.8.
- On specifier syntax and complete information, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On file sharing and record locking, see the *Guide to OpenVMS File Applications*.

6.9.4. 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 a second OPEN statement 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 6.9.5), reading and writing records usually advances the current record position by one record. As discussed in Section 6.9.6, more than one record might be transferred using a single record I/O statement.

6.9.5. 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
```

6.9.6. 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.

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' specifier), 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.

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), an error (OUTSTAOVE) occurs.

- 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 VSI Fortran I/O statements, see Table 6.1.
- On record I/O specifiers, see Section 6.9.1.
- On statement syntax and specifier values, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On improving VSI Fortran I/O performance, see Section 5.5.
- On user-supplied OPEN Procedures (the USEROPEN specifier), see Section 11.4.

6.10. Output Data Buffering and RMS Journaling

When a VSI Fortran output statement is executed, the record data may not be written immediately to the file or device.

To enhance performance, VSI Fortran uses the OpenVMS RMS “write-behind” and “multibuffering” features, which group records together in a memory buffer and delays the actual device write operation until the buffers are full or the file is closed. In most cases, this is desirable (for instance, to minimize disk I/O).

For those applications that depend on data being written to the physical device immediately, “write-behind” and “multibuffering” can result in incomplete data in the case of a power loss or other severe problem that prevents the data being written.

For applications that require guaranteed file consistency for disaster recovery or transactional integrity, the RMS Journaling product is recommended. RMS Journaling provides three types of journaling:

- After-Image journaling, in which journaled transactions allow you to redo record operations.
- Before-Image journaling, in which journaled transactions allow you to undo record operations.
- Recovery Unit (RU) for transactional integrity (multiple operations treated as one transaction)

Both After-Image and Before-Image journaling can be used without modifying the application.

Other applications that do not need the degree of safety provided by RMS journaling can use RMS features to cause data to be written to the file or device more frequently. The simplest method is to use the SYS$FLUSH system service to cause RMS to perform all pending writes immediately to disk. This also has the effect of updating the file's end-of-file pointer so that all of the data written up to that point becomes accessible. An application might choose to call SYS$FLUSH at an interval of every hundred records, for example. The more often SYS$FLUSH is called, the more often the file is updated, but the more performance is affected.

When calling SYS$FLUSH, the RMS Record Access Block (RAB) for the file must be passed as an argument. For files opened by VSI Fortran (or Compaq Fortran 77), the FOR$RAB intrinsic function may be used to obtain the RAB. For example:

```
INTEGER (KIND=4) :: FOR$RAB, IUNIT
    .
    .
    .
IREC_COUNT = 0  DO WHILE (....)
    .
    .
    .
  WRITE (IUNIT) DATA
  IREC_COUNT = IREC_COUNT + 1
  IF (IREC_COUNT .EQ. 100) THEN
     CALL SYS$FLUSH(%VAL(FOR$RAB(IUNIT)))
     IREC_COUNT = 0
     END IF
END DO
```

For More Information:

- On RMS Journaling, see the *RMS Journaling for OpenVMS Manual*.
- On SYS$FLUSH and other RMS features, see Chapter 11, the *Guide to OpenVMS File Applications*, and the *VSI OpenVMS Record Management Services Reference Manual*.
- On using the FOR$RAB intrinsic function, see Section 11.2.3.
- On improving VSI Fortran I/O performance, see Section 5.5.

Chapter 7. Run-Time Errors

This chapter describes:

- Section 7.1: Run-Time Error Overview
- Section 7.2: RTL Default Error Processing
- Section 7.3: Handling Errors
- Section 7.4: List of Run-Time Messages

7.1. Run-Time Error Overview

During execution, your program may encounter errors or exception conditions. These conditions can result from errors that occur during I/O operations, from invalid input data, from argument errors in calls to the mathematical library, from arithmetic errors, or from system-detected errors.

The VSI Fortran Run-Time Library (RTL) provides default processing for error conditions, generates appropriate messages, and takes action to recover from errors whenever possible. However, 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 error (ERR), end-of-record (EOR), and end-of-file (END) branch specifiers in I/O statements.
- To identify Fortran-specific errors based on the value of IOSTAT, use the I/O status specifier (IOSTAT) in I/O statements.
- To tailor error processing to the special requirements of your applications, use the OpenVMS condition-handling facility (including user-written condition handlers). (For information on user-written condition handlers, see Chapter 14).

7.2. RTL Default Error Processing

The RTL contains condition handlers that process a number of errors that may occur during VSI Fortran program execution. A default action is defined for each Fortran-specific error recognized by the RTL. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

Unless you specify the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier when you compile the program, error reporting of exceptions may be inexact; the exception may not be reported until a few instructions after the one that caused the exception. This makes continuation from an exception trap not feasible.

The way in which the RTL actually processes errors depends on several factors:

- The severity of the error
- Whether an I/O error-handling specifier or a condition handler was used

The following FORTRAN command qualifiers are related to handling errors and exceptions:

- The /CHECK=BOUNDS, /CHECK=OVERFLOW, and /CHECK=UNDERFLOW qualifiers generate extra code to catch certain conditions. For example, the /CHECK=OVERFLOW qualifier generates extra code to catch integer overflow conditions.
- The /CHECK=ARG_INFO (I64 only) qualifier controls whether run-time checking of the actual argument list occurs.
- The /CHECK=FP_MODE (I64 only) qualifier controls whether run-time checking of the current state of the processor's floating-point status register (FPSR) occurs.
- The /CHECK=NOFORMAT, /CHECK=NOOUTPUT_CONVERSION, and /CHECK=NOPOWER qualifiers reduce the severity level of the associated run-time error to allow program continuation.
- The /CHECK=NOFP_EXCEPTIONS qualifier and the /CHECK=UNDERFLOW qualifier control the handling and reporting of floating-point arithmetic exceptions at run time.
- The /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier (and certain /IEEE_MODE keywords) influence the reporting of floating-point arithmetic exceptions at run time.
- The /WARNINGS qualifier controls compile-time warning messages, which in some circumstances can help determine the cause of a run-time error.

For More Information:

- On the FORTRAN command qualifier /CHECK, see Section 2.3.11.
- On other /CHECK qualifier keywords, see Section 2.3.11.
- On the FORTRAN qualifiers that control compile-time warning messages, see Section 2.3.51.
- On IEEE floating-point data types and exceptional values, see Section 2.3.24.
- On FORTRAN command qualifiers and their categories, see Table 2.1.
- On VSI Fortran intrinsic data types and their ranges, see Chapter 8.

7.2.1. Run-Time Message Format

The general form of VSI Fortran run-time messages follows:

```
%FOR-severity-mnemonic, message-text
```

The contents of the fields in run-time messages follow:

%	The percent sign identifies the line as a message.
FOR	The facility code for Compaq Fortran 77 and VSI Fortran.
`severity`	A single character that determines message severity. The types of run-time messages are: Fatal (F), Error (E), and Informational (I).
`mnemonic`	A 6- to 9-character name that uniquely identifies that message.
`message_text`	Explains the event or reason why the message appears.

For example, the following message has a severity of Fatal, a mnemonic of ADJARRDIM, and message text of “adjustable array dimension error”:

```
%FOR-F-ADJARRDIM, adjustable array dimension error
```

7.2.2. Run-Time Message Severity Levels

In order of greatest to least severity, the classes of run-time diagnostic messages are as follows:

Severity Code	Description
F	Fatal (severe) This must be corrected. The program cannot complete execution and is terminated when the error is encountered, unless for I/O statements the program uses the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier (see Section 7.3.1 and Section 7.3.2). You can also continue from certain fatal-level messages by using a condition handler.
E	Error This should be corrected. The program may continue execution, but the output from this execution may be incorrect.
I	Informational This should be investigated. The program continues executing, but output from this execution may be incorrect.

The severity depends on the source of the message. In some cases, certain FORTRAN command qualifiers can change the severity level or control whether messages are displayed (such as the /CHECK and /IEEE_MODE keywords).

7.3. Handling Errors

Whenever possible, the VSI Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors.

When no recovery method is specified for a statement and a fatal-level error occurs, a message appears and the program terminates. To prevent program termination, you must include either an appropriate I/O error-handling specifier (see Section 7.3) or a condition handler that performs an unwind (see Chapter 14). The I/O error-handling specifier or condition handler might also handle error-level messages.

You can explicitly supplement or override default actions by using the following VSI Fortran methods:

- To transfer control to error-handling code within the program, use the ERR, EOR, and END branch specifiers in I/O statements (see Section 7.3.1).
- The ERR, EOR, and END branch specifiers transfer control to a part of your program designed to handle specific errors. The error-handling code associated with the ERR branch usually handles multiple types of errors.

  To identify Fortran-specific I/O errors based on the value of VSI Fortran RTL error codes, use the I/O status specifier (IOSTAT) in I/O statements (or call the ERRSNS subroutine) (see Section 7.3.2).

When a fatal error occurs during program execution, the RTL default action is to print an error message and terminate the program. You can establish an OpenVMS condition handler that performs an unwind for certain fatal errors.

These error-processing methods are complementary; you can use all of them within the same program. However, before attempting to write a condition handler, you should be familiar with the OpenVMS condition-handling facility (CHF) and with the condition-handling description in Chapter 14.

Using the END, EOR, or ERR branch specifiers in I/O statements prevent signaling (display) of errors, including secondary return values for file system errors, such as RMS errors. Using these specifiers prevent the transfer of control to a condition handler.

There are certain file system errors where no handler (condition handler or vector exception handler) exists. To obtain the secondary file system errors in these cases, remove the END, EOR, and ERR specifiers, recompile, relink, and rerun the program.

You do not need to remove the ERR or IOSTAT specifiers if you use a vectored exception handler (established using SYS$SETEXV), which will receive control instead of the ERR and IOSTAT specifiers. The ERRSNS subroutine allows you to obtain secondary return values for file system errors (see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

7.3.1. Using the ERR, EOR, and END Branch Specifiers

When an error, end-of-record, or end-of-file condition occurs during program execution, the RTL default action is to display a message and terminate execution of the program.

You can use the ERR, EOR, and END specifiers in I/O statements to override this default by transferring control to a specified point in 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. Note that end-of-file and end-of-record are not considered error conditions by the Fortran language standard.

If an END, EOR, or ERR branch specifier is present, and the corresponding condition occurs, no message is displayed and execution continues at the statement designated in the appropriate specifier.

For example, consider the following READ statement:

```
READ (8, 50, END=400) X,Y,Z
```

If an end-of-file condition occurs during execution of this statement, the contents of variables X, Y, and Z become undefined, and control is transferred to the statement at label 400. You can also add an ERR specifier to transfer control if an error condition occurs. Note that an end-of-file or end-of-record condition does not cause an ERR specifier branch to be taken.

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 specify ERR as a keyword to such I/O statements as 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 statement 999.

For More Information:

- On the IOSTAT specifier, see Section 7.3.2).
- On detailed descriptions of errors processed by the RTL, see Table C.1 or online FORTRAN HELP.
- On user-written condition handlers, see Chapter 14.

7.3.2. 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. It can supplement or replace the END, EOR, and ERR transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses printing of an error message and causes the specified integer variable, array element, or scalar field reference to be defined as one of the following:

- A value of –2 if an end-of-record condition occurs (nonadvancing reads).
- A value of –1 if an end-of-file condition occurs.
- A value of 0 if neither an error condition nor an end-of-file condition occurs.
- A positive integer value if an error condition occurs (this value is one of the Fortran-specific IOSTAT numbers listed in Table 7.1).

Following 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.

Your program can include the $FORIOSDEF library module from the FORSYSDEF library (automatically searched during compilation) to obtain symbolic definitions for the values of IOSTAT.

The values of the IOSTAT symbols from the $FORIOSDEF library module are not the same as the values of the Fortran condition symbols from the $FORDEF library module.

The symbolic names in the $FORIOSDEF library module have a form similar to the Fortran condition symbols:

Fortran Condition Symbol ($FORDEF)	**IOSTAT Symbolic Name ($FORIOSDEF)**
FOR$_mnemonic	FOR$IOS_mnemonic

Example 7.1 uses the ERR and IOSTAT specifiers to handle an OPEN statement error (in the FILE specifier). Condition symbols are included from the $FORIOSDEF library module (in FORSYSDEF).

Example 7.1. Handling OPEN Statement File Name Errors

```
     CHARACTER(LEN=40) :: FILNM      ! Typed file specification
     INCLUDE '($FORIOSDEF)'          ! Include condition symbol definitions

     DO I=1,4                        ! Allow four tries
       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 records
        .

       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

! After four attempts or a Ctrl/Z on the READ statement, allow program
 restart

     WRITE (6,*) 'File not found. Type DIRECTORY to find file and run
 again'
   END PROGRAM
```

For More Information:

- On user-written condition handlers, see Chapter 14.
- On the END, EOR, or ERR branch specifiers, see Section 7.3.1.
- On detailed descriptions of errors processed by the RTL, see Table C.1 or online FORTRAN HELP.
- On the ERRSNS subroutine, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On including library modules from text libraries, see Section 10.8.1.

7.4. List of Run-Time Messages

Table 7.1 lists the Fortran-specific errors processed by the RTL. For each error, the table shows the Fortran-specific message mnemonic (follows either FOR$_ or FOR$IOS_ for condition symbols), the

Fortran-specific message number, the severity (fatal, error, or informational), the message text. For more detailed descriptions of errors processed by the RTL, see Table C.1.

Table 7.1. Summary of Run-Time Errors

Mnemonic	Number[1]	Severity	Message Text
NOTFORSPE[2]	1	F	not a Fortran-specific error
BUG_CHECK	8	F	internal consistency check failure
SYNERRNAM	17	F	syntax error in NAMELIST input
TOOMANVAL	18	F	too many values for NAMELIST variable
INVREFVAR	19	F	invalid reference to variable in NAMELIST input
REWERR	20	F	REWIND error
DUPFILSPE	21	F	duplicate file specifications
INPRECTOO	22	F	input record too long
BACERR	23	F	BACKSPACE error
ENDDURREA	24	F	end-of-file during read
RECNUMOUT	25	F	record number outside range
OPEDEFREQ	26	F	OPEN or DEFINE FILE required
TOOMANREC	27	F	too many records in I/O statement
CLOERR	28	F	CLOSE error
FILNOTFOU	29	F	file not found
OPEFAI	30	F	open failure
MIXFILACC	31	F	mixed file access modes
INVLOGUNI	32	F	invalid logical unit number
ENDFILERR	33	F	ENDFILE error
UNIALROPE	34	F	unit already open
SEGRECFOR	35	F	segmented record format error
ATTACCNON	36	F	attempt to access non-existent record
INCRECLEN	37	F	inconsistent record length
ERRDURWRI	38	F	error during write
ERRDURREA	39	F	error during read
RECIO_OPE	40	F	recursive I/O operation
INSVIRMEM	41	F	insufficient virtual memory
NO_SUCDEV	42	F	no such device
FILNAMSPE	43	F	file name specification error
INCRECTYP	44	F	inconsistent record type
KEYVALERR	45	F	keyword value error in OPEN statement
INCOPECLO	46	F	inconsistent OPEN/CLOSE parameters
WRIREAFIL	47	F	write to READONLY file
INVARGFOR	48	F	invalid argument to Fortran Run-Time Library
INVKEYSPE	49	F	invalid key specification

Mnemonic	Number[1]	Severity	Message Text
INCKEYCHG	50	F	inconsistent key change or duplicate key
INCFILORG	51	F	inconsistent file organization
SPERECLOC	52	F	specified record locked
NO_CURREC	53	F	no current record
REWRITERR	54	F	REWRITE error
DELERR	55	F	DELETE error
UNLERR	56	F	UNLOCK error
FINERR	57	F	FIND error
FMYSYN	58	I	format syntax error at or near `xxx`
LISIO_SYN[3]	59	F	list-directed I/O syntax error
INFFORLOO	60	F	infinite format loop
FORVARMIS [3]	61	F or I[4]	format/variable-type mismatch
SYNERRFOR	62	F	syntax error in format
OUTCONERR[34]	63	E or I[4]	output conversion error
INPCONERR[3]	64	F	input conversion error
FLTINV	65	E	floating invalid
OUTSTAOVE	66	F	output statement overflows record
INPSTAREQ	67	F	input statement requires too much data
VFEVALERR[3]	68	F	variable format expression value error
INTOVF	70	F	integer overflow
INTDIV	71	F	integer divide by zero
FLTOVF	72	E	floating overflow
FLTDIV	73	E	floating divide by zero
FLTUND	74	E	floating underflow
SUBRNG	77	F	subscript out of range
WRONUMARG	80	F	wrong number of arguments
INVARGMAT	81	F	invalid argument to math library
UNDEXP[5]	82	F	undefined exponentiation
LOGZERNEG [5]	83	F	logarithm of zero or negative value
SQUROONEG [5]	84	F	square root of negative value
SIGLOSMAT [5]	87	F	significance lost in math library
FLOOVEMAT [5]	88	F	floating overflow in math library
FLOUNDMAT	89	E	floating underflow in math library
ADJARRDIM[6]	93	F	adjustable array dimension error
INVMATKEY	94	F	invalid key match specifier for key direction
FLOCONFAI	95	E	floating point conversion failed
FLTINE	140	E	floating inexact
ROPRAND	144	F	reserved operand

Mnemonic	Number[1]	Severity	Message Text
ASSERTERR	145	F	assertion error
NULPTRERR	146	F	null pointer error
STKOVF	147	F	stack overflow
STRLENERR	148	F	string length error
SUBSTRERR	149	F	substring error
RANGEERR	150	F	range error
INVREALLOC	151	F	allocatable array is already allocated
RESACQFAI	152	F	unresolved contention for VSI Fortran RTL global resource
INVDEALLOC	153	F	allocatable array is not allocated
INVDEALLOC2	173	F	A pointer passed to DEALLOCATE points to an array that cannot be deallocated
SHORTDATEARG	175	F	DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8
SHORTTIMEARG	176	F	TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10
SHORTZONEARG	177	F	ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5
DIV	178	F	divide by zero
ARRSIZEOVF	179	F	cannot allocate array — overflow on array size calculation
UNFIO_FMT	256	F	unformatted I/O to unit open for formatted transfers
FMTIO_UNF	257	F	formatted I/O to unit open for unformatted transfers
DIRIO_KEY	258	F	direct-access I/O to unit open for keyed access
SEQIO_DIR	259	F	sequential-access I/O to unit open for direct access
KEYIO_DIR	260	F	keyed-access I/O to unit open for direct access
OPERREQDIS	264	F	operation requires file to be on disk or tape
OPEREQSEQ	265	F	operation requires sequential file organization and access
ENDRECDUR	268	F	end of record during read
FLOINEEXC	296	I	`nn` floating inexact traps
FLOINCEXC	297	I	`nn` floating invalid traps
FLOOVREXC	298	I	`nn` floating overflow traps
FLODIV0EXC	299	I	`nn` divide-by-zero traps
FLOUNDEXC	300	I	`nn` floating underflow traps

[1]Although most error numbers are returned as IOSTAT values, the following are not: 1, 24 (END specifier), 41, 58, 70–75, 77, 80-89, 95, 140–150, 151, 153, 173, 175–177, 179, 266, 268 (EOR specifier), 297, 298, 299, 300. You can use condition symbols (FOR$_mnemonic or FOR$IOS_mnemonic) to obtain the number (see Section 7.3.2). Some of these error numbers are returned as STAT= values in either the ALLOCATE (41, 151, 179) or DEALLOCATE (41, 153, 173) Fortran statement.

[2]The FOR$_NOTFORSPE error (number 1) indicates an error not reportable through any other message. If you call ERRSNS, an error of this kind returns a value of 1. Use the fifth argument of the call to ERRSNS (condval) to obtain the unique system condition value that identifies the error.

[3]For error numbers 59, 61, 63, 64, and 68, the ERR transfer is taken after completion of the I/O statement. The resulting file status and record position are the same as if no error had occurred. Other I/O errors take the ERR transfer as soon as the error is detected, and file status and record position are undefined.

[4]For errors 61 and 63, the severity depends on the /CHECK keywords in effect during compilation (see Section 2.3.11). If no ERR address is defined for error number 63, the program continues and the entire overflowed field is filled with asterisks to indicate the error in the output record.

[5]Function return values for error numbers 82, 83, 84, 87, 88, and 89 can be modified by means of user-written condition handlers. (See Chapter 14 for information about user-written condition handlers.)

[6]If error number 93 (FOR$_ADJARRDIM) occurs and a user-written condition handler causes execution to continue, any reference to the array in question may cause an access violation.

The message mnemonic shown in the first column is part of the condition status code symbols signaled by the RTL I/O support routines. You can define these symbolic values in your program by including the library module $FORDEF or $FORIOSDEF from the system-supplied default library FORSYSDEF.TLB:

- The symbolic values defined in library module $FORDEF have the form FOR$_ `mnemonic`.
- The condition symbols defined in $FORIOSDEF have the form FOR$IOS_ `mnemonic`.

If you will be using the IOSTAT specifier for error handling, you should include the $FORIOSDEF library module (instead of $FORDEF) from the FORSYSDEF.TLB library (see Section 7.3.2).

The standard VSI Fortran error numbers that are generally compatible with other versions of VSI Fortran are shown in the second column. Most of these error values are returned to IOSTAT variables when an I/O error is detected.

The codes in the third column indicate the severity of the error conditions (see Section 7.2.2).

For more detailed descriptions of errors processed by the RTL, see Table C.1 or type the following DCL command to obtain a list of mnemonics (such as ADJARRDIM):

```
$ HELP FORTRAN ERROR RUN_TIME
```

For More Information:

- On user-written condition handlers, see Chapter 14.
- On the END, EOR, or ERR branch specifiers, see Section 7.3.1.
- On the IOSTAT specifier, see Section 7.3.2.
- On detailed descriptions of errors processed by the RTL, see Table C.1 or online FORTRAN HELP.
- On the ERRSNS subroutine, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the Alpha architecture, see the *Alpha Architecture Reference Manual*.
- On locating exceptions within the debugger, see Section 4.6.

Chapter 8. Data Types and Representation

This chapter describes:

- Section 8.1: Summary of Data Types and Characteristics
- Section 8.2: Integer Data Representations
- Section 8.3: Logical Data Representations
- Section 8.4: Native Floating-Point Representations and IEEE Exceptional Values
- Section 8.5: Character Representation
- Section 8.6: 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.

VSI 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 9.
- On VSI Fortran I/O, see Chapter 6.

8.1. Summary of Data Types and Characteristics

Table 8.1 lists the intrinsic data types provided by VSI Fortran, the storage required, and valid numeric ranges.

Table 8.1. VSI 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 FORTRAN command qualifiers (see Section 8.2.1).
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)
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]

Data Type	Bytes	Description
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 FORTRAN command qualifiers (see Section 8.3).
LOGICAL (KIND=1) LOGICAL*1	1 (8 bits)	Logical values .TRUE. or .FALSE.
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]
REAL	4 or 8	Real floating-point numbers whose size is controlled by a kind type parameter or, if a kind type parameter (or size specifier) is omitted, by using the FORTRAN command /REAL_SIZE qualifier. To control the floating-point format used in memory, use the FORTRAN command /FLOAT qualifier.
REAL (KIND=4) REAL*4	4 (32 bits)	Single-precision real floating-point values in IEEE S_float or VAX F_float formats. IEEE S_float normalized values range from 1.17549435E–38 to 3.40282347E38. Values between 1.17549429E–38 and 1.40129846E–45 are denormalized[3]. VAX F_float values range from 0.293873588E–38 to 1.7014117E38.
DOUBLE PRECISION REAL (KIND=8) REAL*8	8 (64 bits)	Double-precision real floating-point values in IEEE T_float, VAX G_float, or VAX D_float formats. IEEE T_float normalized values range from 2.2250738585072013D–308 to 1.7976931348623158D308. Values between 2.2250738585072008D–308 and 4.94065645841246544D–324 are denormalized[3]. VAX G_float values range from 0.5562684646268004D–308 to 0.8988465674311578540 7D308. VAX D_float values range from 0.2938735877055719D–38 to 1.70141183460469229D38. You can change the data size of DOUBLE PRECISION declarations from REAL (KIND=8) to REAL (KIND=16) with the FORTRAN command /DOUBLE_SIZE qualifier.

Data Type	Bytes	Description
REAL (KIND=16) REAL*16	16 (128 bits)	Extended-precision real floating-point values in IEEE-like X_float format ranging from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932. The smallest normalized number is 3.362103143112093506262677817321753Q-4932.
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 FORTRAN command /REAL_SIZE qualifier. To control the floating-point format used in memory, use the FORTRAN command /FLOAT qualifier.
COMPLEX (KIND=4) COMPLEX*8	8 (64 bits)	Single-precision complex floating-point values in a pair of floating-point parts: real and imaginary. Use the IEEE S_float or VAX F_float format. IEEE S_float real and imaginary parts range from 1.17549435E–38 to 3.40282347E38. Values between 1.17549429E–38 and 1.40129846E–45 are denormalized[3]. VAX F_float real and imaginary parts range from 0.293873588E–38 to 1.7014117E38.
DOUBLE COMPLEX COMPLEX (KIND=8) COMPLEX*16	16 (128 bits)	Double-precision complex floating-point values in a pair of parts: real and imaginary. Use the IEEE T_float, VAX G_float, or VAX D_float format. IEEE T_float format 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]. VAX G_float format real and imaginary parts each range from 0.5562684646268004D–308 to 0.8988465674311578 5407D308. VAX D_float format real and imaginary parts each range from 0.2938735877055719D–38 to 1.70141183460469229D38.
COMPLEX (KIND=16) COMPLEX*32	32 (256 bits)	Extended-precision complex floating-point values in a pair of IEEE 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.

Data Type	Bytes	Description
HOLLERITH	1 byte (8 bits) per Hollerith character	Hollerith constants.

[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.

[3]You cannot write a constant for a denormalized number. For more information on floating-point underflow, see Section 2.3.24.

In addition to the intrinsic numeric data types, you can define nondecimal (binary, octal, or hexadecimal) constants as explained in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

8.2. 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).

8.2.1. Integer Declarations and FORTRAN Command Qualifiers

The default size used for an INTEGER data declaration without a kind parameter (or size specifier) 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). VSI Fortran provides intrinsic INTEGER kinds of 1, 2, 4, and 8. Each INTEGER kind number corresponds to the 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 90 standard.

- Use the FORTRAN command /INTEGER_SIZE= `nn` qualifier to control the size of all default (without a kind parameter or size specifier) INTEGER and LOGICAL declarations (see Section 2.3.26).

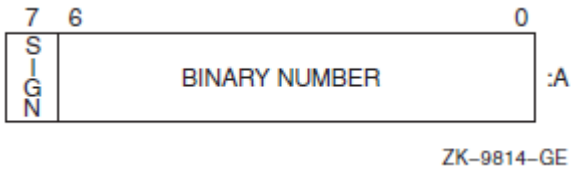
8.2.2. INTEGER (KIND=1) or INTEGER*1 Representation

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 8.1.

Figure 8.1. INTEGER (KIND=1) or INTEGER*1 Representation



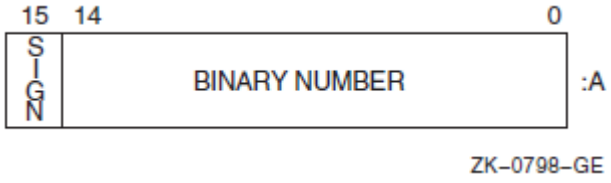
8.2.3. INTEGER (KIND=2) or INTEGER*2 Representation

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 8.2.

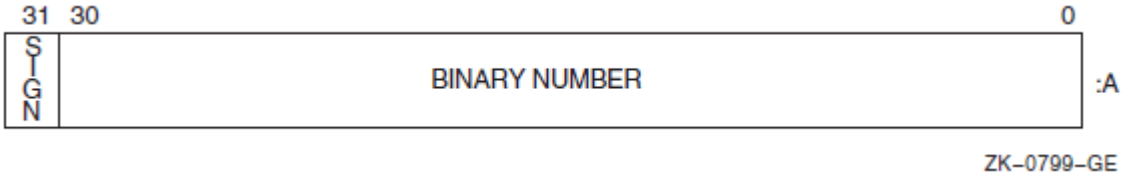
Figure 8.2. INTEGER (KIND=2) or INTEGER*2 Representation



8.2.4. 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 8.3.

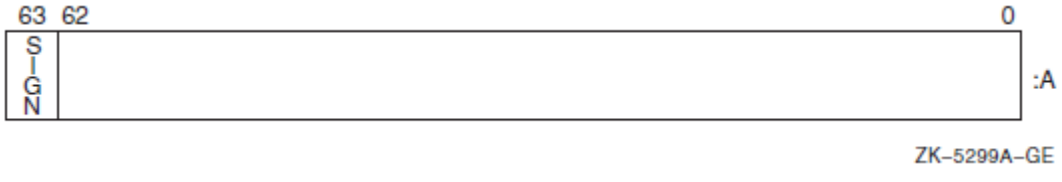
Figure 8.3. INTEGER (KIND=4) or INTEGER*4 Representation



8.2.5. 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 8.4.

Figure 8.4. INTEGER (KIND=8) or INTEGER*8 Representation



For More Information:

- On defining constants and assigning values to variables, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

- On intrinsic functions related to the various data types, such as KIND and SELECTED_INT_KIND, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the FORTRAN command qualifiers that control the size of default INTEGER declarations, see Section 2.3.26.

8.3. 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). VSI 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.

  You can also use a size specifier, such as LOGICAL*4, but be aware this is an extension to the Fortran 90 standard.
- Use the FORTRAN command /INTEGER_SIZE= `nn` qualifier to control the size of default (without a kind parameter or size specifier) LOGICAL and INTEGER declarations (see Section 2.3.26).

To improve 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.

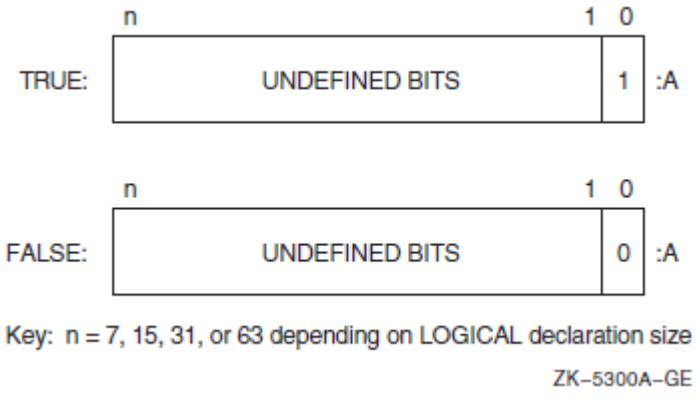
Logical (intrinsic) values can also be stored in the following sizes of contiguous bytes starting on an arbitrary byte boundary:

- 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 90 standard). 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 8.5.

Figure 8.5. LOGICAL Representations



For More Information:

- On defining constants and assigning values to variables, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On intrinsic functions related to the various data types, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the FORTRAN command qualifiers that control the size of default LOGICAL declarations, see Section 2.3.26.

8.4. Native Floating-Point Representations and IEEE Exceptional Values

Floating-point numbers are stored on OpenVMS systems in one of the following IEEE or VAX little endian binary floating-point formats, as follows:

- REAL (KIND=4) or REAL*4 declarations are stored in little endian IEEE S_float format or VAX F_float format.
- REAL (KIND=8) or REAL*8 declarations are stored in little endian IEEE T_float format, VAX G_float format, or VAX D_float format.
- REAL (KIND=16) or REAL*16 declarations always are stored in little endian Alpha IEEE-like X_float 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 or VAX F_float format using two REAL (KIND=4) or REAL*4 values.
- COMPLEX (KIND=8) or COMPLEX*16 declarations are stored in IEEE T_float, VAX G_float, or VAX D_float format using two REAL (KIND=8) or REAL*8 values.
- COMPLEX (KIND=16) or COMPLEX*32 declarations are stored in IEEE X_float format using two REAL (KIND=16) or REAL*16 values.

To specify the floating-point format used in memory, use the FORTRAN command /FLOAT qualifier. If you do not specify the /FLOAT qualifier, the following formats are used:

- VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) data
- VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) data
- X_float for REAL (KIND=16)

To change the default size for REAL and COMPLEX declarations, use the FORTRAN command /REAL_SIZE qualifier.

To change the default size for DOUBLE PRECISION declarations, use the FORTRAN command /DOUBLE_SIZE qualifier.

All floating-point formats represent fractions using sign-magnitude notation, with the binary radix point to the left of the most significant bit for F_floating, D_floating, and G_floating, and to the right for

S_floating and T_floating. Fractions are assumed to be normalized, and therefore the most significant bit is not stored. This is called “hidden bit normalization”.

With IEEE data, 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).

With VAX data, the hidden bit is assumed to be 1.

For an explanation of the representation of NaN, Infinity, and related IEEE exceptional values, see Section 8.4.8.

8.4.1. REAL, COMPLEX, and DOUBLE PRECISION Declarations and FORTRAN Qualifiers

The default size for REAL, COMPLEX, and DOUBLE PRECISION 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). To change the default REAL (and COMPLEX) size to (KIND=8), use the FORTRAN command /REAL_SIZE qualifier (see Section 2.3.37).
- For COMPLEX data declarations without a kind parameter (or size specifier), the default data size is COMPLEX (KIND=4) (same as COMPLEX*8). To change the default COMPLEX (and REAL) size to (KIND=8), use the FORTRAN command /REAL_SIZE qualifier (see Section 2.3.37).
- For a DOUBLE PRECISION data declarations, the default size is REAL (KIND=8) (same as REAL*8). To change the default DOUBLE PRECISION size to REAL (KIND=16), use the FORTRAN command /DOUBLE_SIZE qualifier (see Section 2.3.17).

You can explicitly declare the length of a REAL or a COMPLEX declaration using a kind parameter or specify DOUBLE PRECISION or DOUBLE COMPLEX.

Intrinsic REAL kinds are 4 (single precision) and 8 (double precision); intrinsic COMPLEX kinds are also 4 (single precision) and 8 (double precision). For example, REAL (KIND=4) requests single-precision floating-point data.

You can also use a size specifier, such as REAL*4, but be aware this is an extension to the Fortran 90 standard.

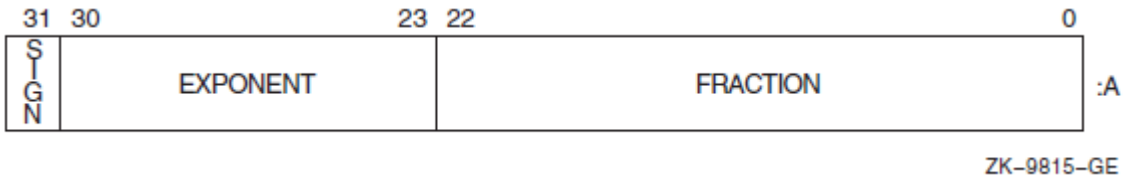
8.4.2. REAL (KIND=4) or REAL*4 Representations

REAL (KIND=4) or REAL*4 data can be in IEEE S_float or VAX F_float formats. This is the default data size for REAL declarations.

8.4.2.1. IEEE S_float 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 8.6.

Figure 8.6. IEEE S_float REAL (KIND=4) or REAL*4 Representation



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.

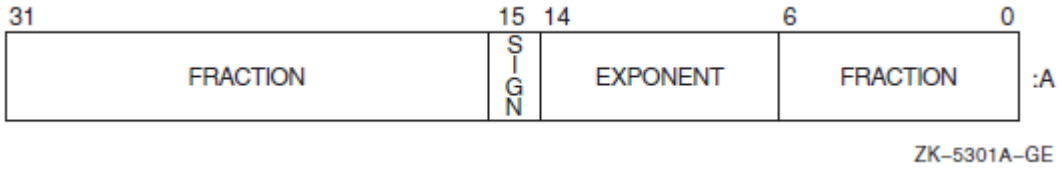
The value of data is in the approximate range: 1.17549435E–38 (normalized) to 3.40282347E38. The IEEE denormalized limit is 1.40129846E–45.

The precision is approximately one part in 2**23, typically seven decimal digits.

8.4.2.2. VAX F_float Representation

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 8.7.

Figure 8.7. 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.

The value of data is in the approximate range: 0.293873588E–38 to 1.7014117E38.

The precision is approximately one part in 2**23, typically seven decimal digits.

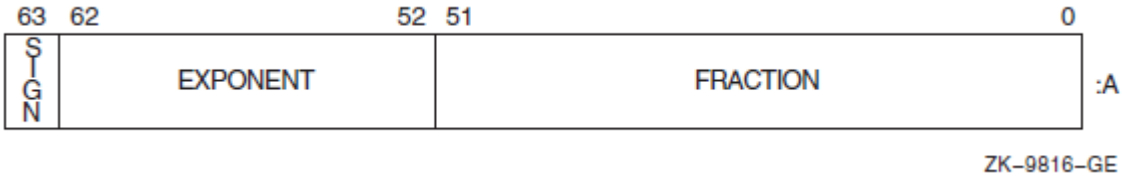
8.4.3. REAL (KIND=8) or REAL*8 Representations

REAL (KIND=8) or REAL*8 data can be in IEEE T_float, VAX G_float, or VAX D_float formats.

8.4.3.1. IEEE T_float 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 8.8.

Figure 8.8. IEEE T_float REAL (KIND=8) or REAL*8 Representation



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.

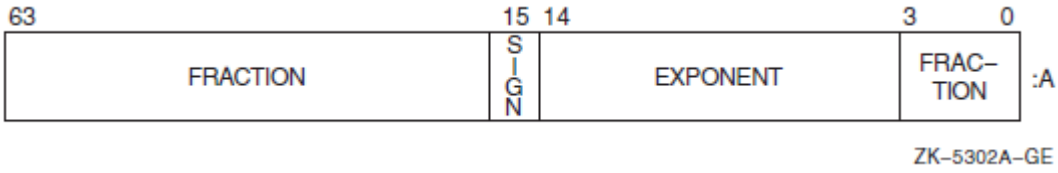
The value of data is in the approximate range: 2.2250738585072013D-308 (normalized) to 1.7976931348623158D308. The IEEE denormalized limit is 4.94065645841246544D-324.

The precision is approximately one part in 2**52, typically 15 decimal digits.

8.4.3.2. VAX G_float Representation

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 8.9.

Figure 8.9. 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.

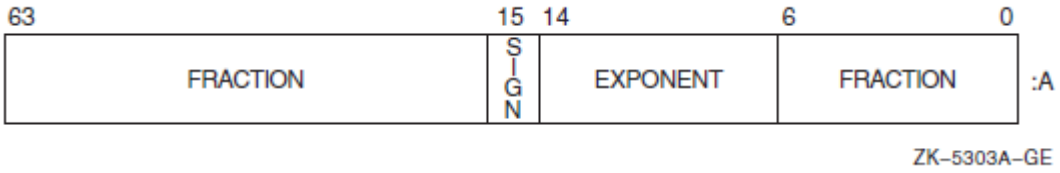
The value of data is in the approximate range: 0.5562684646268004D–308 to 0.8988465674311578540 7D308.

The precision of G_float data is approximately one part in 2**52, typically 15 decimal digits.

8.4.3.3. VAX D_float Representation

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 8.10.

Figure 8.10. 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.

The value of data is in the approximate range: 0.2938735877055719D–38 to 1.70141183460469229D38.

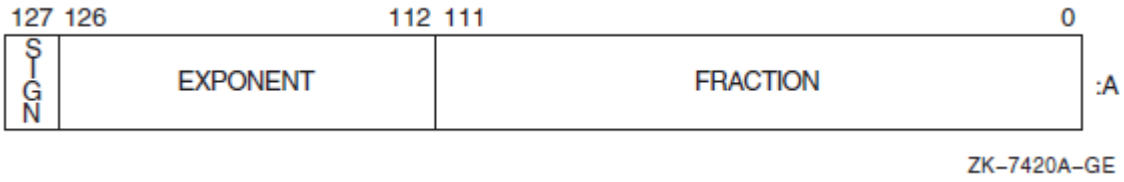
The precision is approximately one part in 2**55, typically 16 decimal digits. Calculations with D_float data on Alpha systems use G_float precision (53-bit instead of 56-bit fraction).

For information about how D_float and G_float calculations are done on I64 systems, see Appendix A.

8.4.4. REAL (KIND=16) or REAL*16 X_float Representation

REAL*16 data occupies 16 contiguous bytes stored in X_float format. Bits are labeled from the right, 0 through 127, as shown in Figure 8.11.

Figure 8.11. X_float REAL (KIND=16) Floating-Point Data Representation



The form of REAL (KIND=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: 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932. The smallest normalized number is 3.362103143112093506262677817321753Q-4932.

In contrast to 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 precision is approximately one part in 2**112 or typically 33 decimal digits.

8.4.5. COMPLEX (KIND=4) or COMPLEX*8 Representations

COMPLEX (KIND=4) or COMPLEX*4 data can be in IEEE S_float or VAX F_float formats. This is the default data size for COMPLEX declarations.

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 or VAX F_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,

The limits (and underflow characteristics for S_float numbers) 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.

Figure 8.12 shows the IEEE S_float representation of COMPLEX (KIND=4) or COMPLEX*8 numbers.

Figure 8.12. IEEE S_float COMPLEX (KIND=4) or COMPLEX*8 Representation

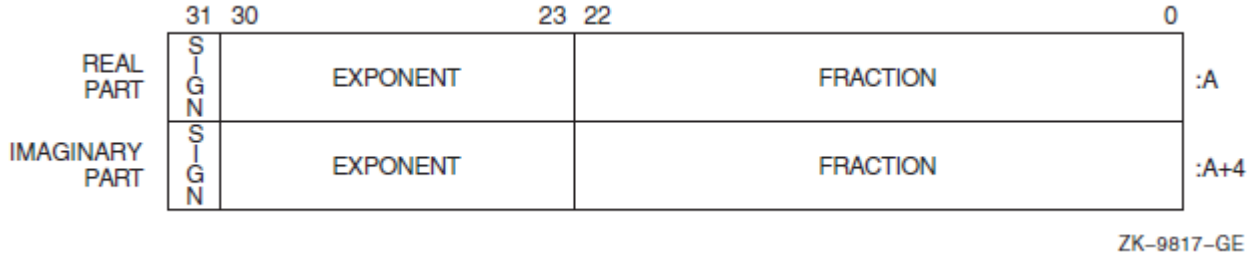
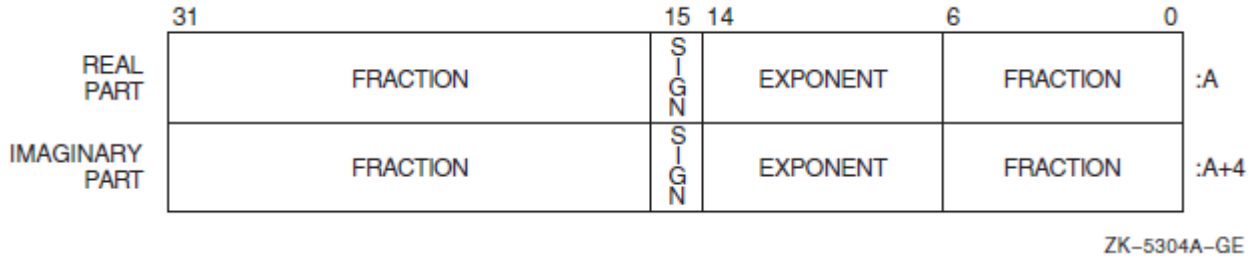


Figure 8.13 shows the VAX F_float representation of COMPLEX (KIND=4) or COMPLEX*8 numbers.

Figure 8.13. VAX F_float COMPLEX (KIND=4) or COMPLEX*8 Representation



8.4.6. COMPLEX (KIND=8) or COMPLEX*16 Representations

COMPLEX (KIND=8) or COMPLEX*16 data can be in IEEE T_float, VAX G_float, or VAX D_float formats.

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, VAX G_float, or VAX D_float formats.

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.

Like REAL (KIND=8) or REAL*8 numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers. The limits (and underflow characteristics for T_float numbers) apply to the two separate real and imaginary parts of a COMPLEX (KIND=8) or COMPLEX*16 number.

Figure 8.14 shows the IEEE T_float COMPLEX (KIND=8) or COMPLEX*16 representation.

Figure 8.14. IEEE T_float COMPLEX (KIND=8) or COMPLEX*16 Representation

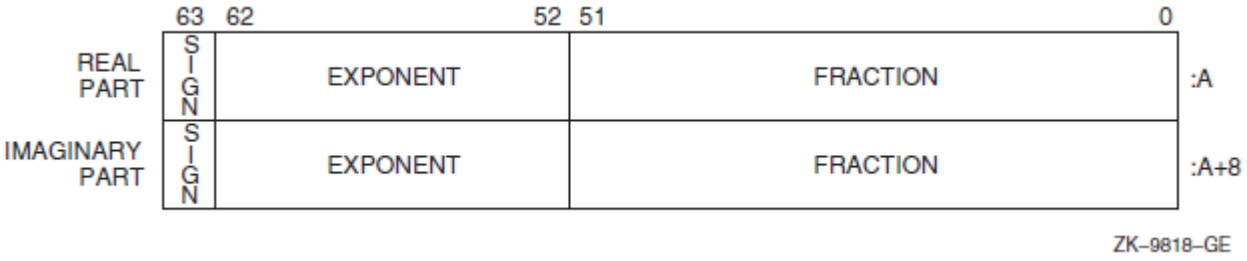


Figure 8.15 (G_float) and Figure 8.16 (D_float) show the representation of the VAX COMPLEX (KIND=8) or COMPLEX*16 numbers.

Figure 8.15. VAX G_float COMPLEX (KIND=8) or COMPLEX*16 Representation

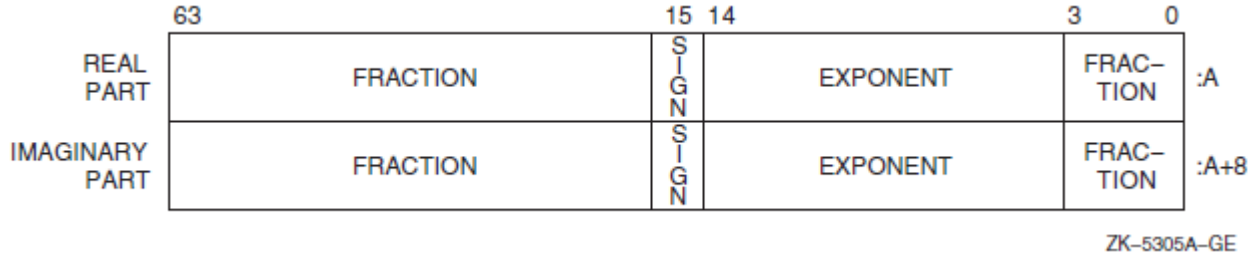
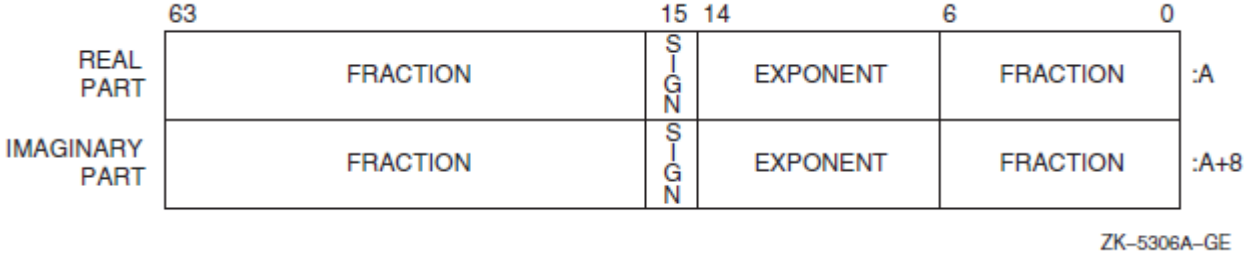


Figure 8.16. VAX D_float COMPLEX (KIND=8) or COMPLEX*16 Representation

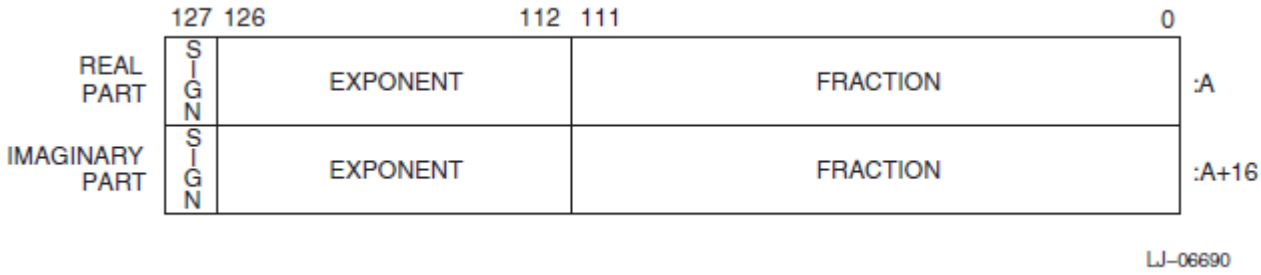


8.4.7. COMPLEX (KIND=16) or COMPLEX*32 Representation

Intrinsic COMPLEX (KIND=16) or COMPLEX*32 (extended precision) data is 32 contiguous bytes containing a pair of REAL*16 values stored in 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 8.17.

Figure 8.17. 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 9.
- On defining constants and assigning values to variables, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On intrinsic functions related to the various data types, such as SELECTED_REAL_KIND, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On OpenVMS VAX floating-point data types (provided for those converting OpenVMS data), see Section B.8.
- On the FORTRAN command qualifiers that control the size of REAL and COMPLEX declarations (without a kind parameter or size specifier), see Section 2.3.37.
- On the FORTRAN command qualifiers that control the size of DOUBLE PRECISION declarations, see Section 2.3.17.
- On IEEE binary floating-point, see the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985).

8.4.8. Exceptional IEEE Floating-Point Representations

Exceptional values usually result from a computation and include plus infinity, minus infinity, NaN, and denormalized numbers. Exceptional values and the representation of zero are associated only with IEEE

S_float, T_float, and X_float formats (/FLOAT=IEEE_FLOAT qualifier), not with VAX floating-point formats. (VAX floating-point representation of minus (–) zero causes a reserved operand fault on VAX systems).

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 floating-point data type, the values of *min* or *max* are listed in Section 8.4.2 (single precision), Section 8.4.3 (double precision), and Section 8.4.4 (extended precision).

  For native VAX floating-point data types, finite numbers do not include reserved operand or dirty zero values.

  Special bit patterns that are not finite numbers represent exceptional values.

- **Infinity**—An IEEE floating-point bit pattern that represents plus or minus infinity. VSI 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. VSI 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. It 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 /IEEE_MODE qualifier keyword. For example, if an exceptional value is used in a calculation, an unrecoverable exception can occur unless you specify the appropriate /IEEE_MODE qualifier keyword. Denormalized numbers can be processed as is, set equal to zero with program continuation or a program stop, and generate warning messages (see Section 2.3.24).

Table 8.2 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 8.2. IEEE 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'
Signaling NaN (+)	From Z '7F800001' to Z '7FBFFFFF'
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'
Signaling NaN (+)	From Z '7FF0000000000001' to Z '7FF7FFFFFFFFFFFF'
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'
Signaling NaN (+)	From Z '7FFF0000000000000000000000000001' to Z '7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF'

VSI 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 such as x=1.0/0 and comparing x to the calculated number).

The appropriate FORTRAN command /IEEE_MODE qualifier keyword 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 VSI Fortran allows continuation for arithmetic exception, you can use the ISNAN intrinsic function. Example 8.1 shows how to use the ISNAN intrinsic function to test whether a REAL*8 (DOUBLE PRECISION) value contains a NaN.

Example 8.1. Testing for a NaN 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
```

To allow continuation when a NaN (or other exceptional value) is encountered in a calculation, this program might be compiled with /FLOAT=IEEE_FLOAT and /IEEE_MODE=UNDERFLOW_TO_ZERO (or /IEEE_MODE=DENORM_RESULTS) qualifiers:

```
$ FORTRAN/FLOAT=IEEE_FLOAT/IEEE_MODE=UNDERFLOW_TO_ZERO ISNAN
$ LINK ISNAN
$ RUN ISNAN
->  Variable F contains a NaN value <-
```

To enable additional run-time message reporting with traceback information (source line correlation), use the FORTRAN command qualifier /CHECK=FP_EXCEPTIONS.

The FP_CLASS intrinsic function is also available to check for exceptional values (see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

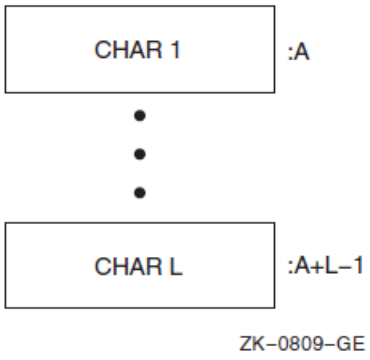
For More Information:

- On using the FORTRAN command /IEEE_MODE qualifier keywords to control arithmetic exception handling, see Section 2.3.24.
- On floating-point architecture, see the *Intel Itanium Architecture Software Developer's Manual*.
- On Alpha 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).

8.5. Character Representation

A character string is a contiguous sequence of bytes in memory, as shown in Figure 8.18.

Figure 8.18. CHARACTER Data Representation



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 and assigning values to variables, using substring expressions and concatenation, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On intrinsic functions related to the various data types, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

8.6. 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 8.18).

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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On intrinsic functions related to the various data types, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

Chapter 9. Converting Unformatted Numeric Data

This chapter describes:

- Section 9.1: Overview of Converting Unformatted Numeric Data
- Section 9.2: Endian Order of Numeric Formats
- Section 9.3: Native and Supported Nonnative Numeric Formats
- Section 9.4: Limitations of Numeric Conversion
- Section 9.5: Methods of Specifying the Unformatted Numeric Format
- Section 9.6: Additional Information on Nonnative Data

9.1. Overview of Converting Unformatted Numeric Data

You specify the floating-point format in memory with the /FLOAT qualifier. VSI Fortran supports the following little endian floating-point formats in memory (default is /FLOAT=G_FLOAT):

Floating-Point Size	**/FLOAT=IEEE_FLOAT**	**/FLOAT=D_FLOAT**	**/FLOAT=G_FLOAT**
KIND=4	S_float	F_float	F_float
KIND=8	T_float	D_float	G_float
KIND=16	X_float	X_float	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.

For example, if your program primarily uses IEEE little endian floating-point data, specify /FLOAT=IEEE_FLOAT to specify use of the S_float, T_float, and X_float formats in memory. If your program needs to read a data file containing a different format (VAX or big endian), you need to specify which VAX or big endian floating-point format to use for conversion into the native IEEE memory format for that file.

Converting unformatted data is generally faster than converting formatted data and is less likely to lose precision for floating-point numbers.

9.2. Endian Order of Numeric Formats

Data storage in different computers use a convention of either **little endian** or **big endian** storage. The storage convention generally applies to numeric values that span multiple bytes, as follows:

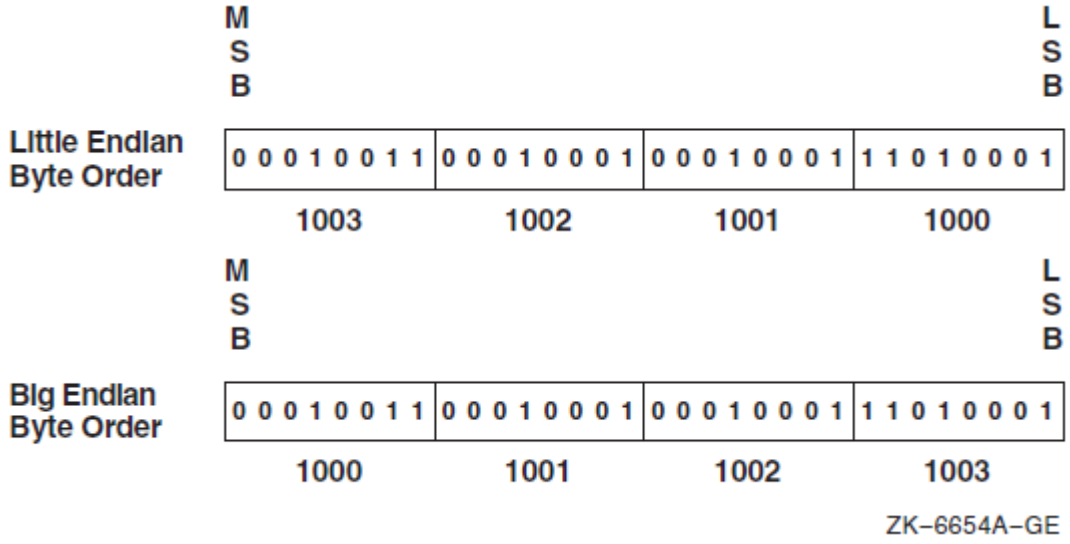
- **Little endian** storage occurs when:
  - The least significant bit (LSB) value is in the byte with the lowest address.

- 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 9.1 shows the difference between the two byte-ordering schemes.

Figure 9.1. Little and Big Endian Storage of an INTEGER Value



Moving data files between big endian and little endian computers requires that the data be converted.

9.3. Native and Supported Nonnative Numeric Formats

VSI 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 Section 7.2).

Supported native and nonnative floating-point formats include:

- Standard IEEE little endian floating-point formats and little endian integers. This format is found on OpenVMS I64 systems, OpenVMS Alpha systems, UNIX systems, and Microsoft Windows operating systems (for IBM-compatible PC systems).

- Standard IEEE big endian floating-point formats and big endian integers found on most UNIX systems.
- VAX little endian floating-point formats and little endian integers supported by Compaq Fortran 77 and VSI Fortran for OpenVMS I64 and Alpha systems and Compaq Fortran 77 for OpenVMS VAX systems. (OpenVMS VAX systems use a different 16-byte REAL format).
- 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 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 INTEGER and LOGICAL declarations without a kind parameter or size specifier, use a FORTRAN command qualifier (see Section 8.2.1).
- The following floating-point sizes and formats are available:
  - Single-precision 4-byte REAL and 8-byte COMPLEX declarations (KIND=4) in either IEEE S_float or VAX F_float formats. This is the default size for all REAL or COMPLEX declarations without a kind parameter or size specifier.
  - Double-precision 8-byte REAL and 16-byte COMPLEX declarations (KIND=8) in IEEE T_float, VAX G_float, or VAX D_float formats. This is the default size for all DOUBLE PRECISION declarations.
  - Extended-precision 16-byte REAL declarations and 32-byte COMPLEX declarations (KIND=16) in IEEE-like X_float format.

  You can specify the real or complex data length by using an explicit data declaration (kind parameter or size specifier). You can change the default size for REAL, COMPLEX, and DOUBLE PRECISION declarations by using FORTRAN command qualifiers (/REAL_SIZE or /DOUBLE_SIZE).

Table 9.1 lists the keywords for the supported unformatted file data formats. Use the appropriate keyword after the /CONVERT qualifier (such as /CONVERT=CRAY) or as an logical name value (see Section 9.5).

Table 9.1. Unformatted Numeric Formats, Keywords, and Supported Data Types

Recognized Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate INTEGER size (one, two, or four bytes) and big endian IEEE floating-point formats for REAL and COMPLEX single- and double-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.

Recognized Keyword	Description
IBM	Big endian integer data of the appropriate INTEGER size (one, two, or four bytes) and big endian IBM 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 native little endian proprietary floating-point formats: F_float for REAL (KIND=4) and COMPLEX (KIND=4) D_float for REAL (KIND=8) and COMPLEX (KIND=8) IEEE-like X_float for REAL (KIND=16) and COMPLEX (KIND=16) For information on these native OpenVMS formats, see Table 8.1 and Section 8.4.
FGX	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following native little endian proprietary floating-point formats: F_float for REAL (KIND=4) and COMPLEX (KIND=4) G_float for REAL (KIND=8) and COMPLEX (KIND=8) IEEE-like X_float for REAL (KIND=16) and COMPLEX (KIND=16) For information on these native OpenVMS formats, see Table 8.1 and Section 8.4.
LITTLE_ENDIAN	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-like X_float for REAL (KIND=16) and COMPLEX (KIND=16) For information on these native OpenVMS formats, see Table 8.1 and Section 8.4.
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 proprietary floating-point formats: F_float for REAL (KIND=4) and COMPLEX (KIND=4) D_float for REAL (KIND=8) and COMPLEX (KIND=8) H_float for REAL (KIND=16) For information on the F_float and D_float formats, see Table 8.1 and Section 8.4. For information on the H_float format (available only on VAX systems), see Section B.8.
VAXG	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats:

Recognized Keyword	Description
	F_float format for REAL (KIND=4) and COMPLEX (KIND=4) G_float format for REAL (KIND=8) and COMPLEX (KIND=8) H_float format for REAL (KIND=16) For information on the F_float and D_float formats, see Table 8.1 and Section 8.4. For information on the H_float format (available only on VAX systems), see Section B.8.

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 ranges and the format of native IEEE floating-point data types, see Table 8.1 and Section 8.4.
- On ranges and the format of VAX floating-point data types, see Section B.8.
- On specifying the size of INTEGER declarations (without a kind) using an FORTRAN command qualifier, see Section 8.2.1.
- On specifying the size of LOGICAL declarations (without a kind) using an FORTRAN command qualifier, see Section 8.3.
- On specifying the size of REAL or COMPLEX declarations (without a kind) using an FORTRAN command qualifier, see Section 8.4.1.
- On data declarations and other VSI Fortran language information, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

9.4. Limitations of Numeric Conversion

The VSI Fortran floating-point conversion solution is not expected to fulfill all floating-point conversion needs.

Data (variables) in record structures (specified in a STRUCTURE statement) and data components of derived types (TYPE statement) are not converted. When 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 they 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. To convert floating-point formats for individual fields, consider using the CVT$CONVERT_FLOAT routine (see Example B.1).

With EQUIVALENCE statements, the data type of the variable named in the I/O statement is used.

9.5. Methods of Specifying the Unformatted Numeric Format

The methods you can use to specify the type of numeric floating-point format are as follows:

- Set a logical name for a specific unit number before the file is opened. The logical name is named FOR$CONVERT`nnn`, where `nnn` is the unit number.
- Set a logical name for a specific file name extension before the file is opened. The logical name is named FOR$CONVERT.`ext` (or FOR$CONVERT_ext), where `ext` is the file name extension (suffix).
- Add the CONVERT specifier to the OPEN statement for a specific unit number.
- Compiling 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.
- Compiling the program with the FORTRAN command /CONVERT= `keyword` qualifier. 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 logical name for a specific unit number
2. Check for a FOR$CONVERT.`ext` logical name and then for a FOR$CONVERT_ext logical name (if the former logical name is not found)
3. Check for 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 FORTRAN command /CONVERT=(keyword) qualifier 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 or VAX format) or otherwise translated by the application program.

Any keyword listed in Table 9.1 can be used with any of these methods.

9.5.1. Logical Name FOR$CONVERT`nnn` Method

You can use the logical name 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 logical name before you open that unit number. For unit numbers that contain fewer than three digits, use leading zeros.

For example, to specify the numeric format for unit 9, set logical name FOR$CONVERT009 to the appropriate value (such as BIG_ENDIAN) before you run the program. For unit 125, set the logical name FOR$CONVERT125 before you run the program.

When you open the file, the logical name is always used, since this method takes precedence over the FORTRAN command qualifier methods. For instance, you might use this method to specify that different unformatted numeric formats for different unit numbers (perhaps in a command procedure that sets the logical name 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 and T_float) when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29. The FORTRAN command qualifier /FLOAT specifies the IEEE floating-point format in memory and the LINK command creates the executable program:

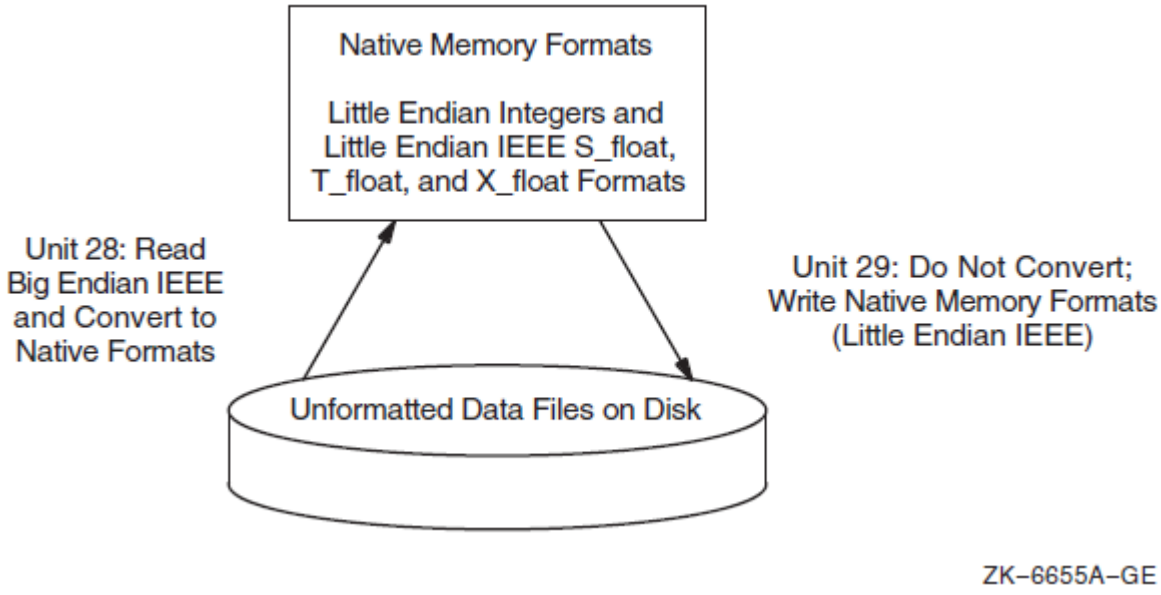
```
$ FORTRAN/FLOAT=IEEE_FLOAT CONV_IEEE
$ LINK CONV_IEEE
```

Without requiring source code modification of recompilation of this program, the following DCL command sequence sets the appropriate logical name and then runs the program CONV_IEEE.EXE:

```
$ DEFINE FOR$CONVERT028 BIG_ENDIAN
$ DEFINE FOR$CONVERT029 NATIVE
$ RUN CONV_IEEE
```

Figure 9.2 shows the data formats used on disk and in memory when the example file conv_ieee.exe is run after the logical names are set with DCL commands.

Figure 9.2. Sample Unformatted File Conversion



For information on the DCL commands you can use to define and deassign logical names, see Appendix D.

9.5.2. Logical Name FOR$CONVERT.ext (and FOR$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 logical name 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. You would `DEFINE FOR $CONVERT.DAT BIG_ENDIAN`.

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 logical names FOR$CONVERT.DATA and FOR$CONVERT`nnn` (for that unit number) are not defined.

The FOR$CONVERT`nnn` method takes precedence over this method. When the appropriate logical name is set when you open the file, the FOR$CONVERT.`ext` logical name is used if a FOR$CONVERT`nnn` logical name is not set for the unit number.

The FOR$CONVERT`nnn` and FOR$CONVERT.`ext` (or FOR$CONVERT_`ext`) logical name 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).

See Example 9.1.

Example 9.1. Example Showing the Use of the FOR$CONVERT.ext Method

```
Source program:

program p
   integer i
   real    r
   open( file='convert_in.txt',  unit=10)
   open( file='convert_out.big', unit=11, form="unformatted")
   open( file='convert_out.dat', unit=12, form="unformatted")
   do i = 1, 10
      read(10,*) r     ! In text
      type *,r
      write(11) r      ! In BIG_ENDIAN
      write(12) r      ! In NATIVE
   enddo
   close(10)
   close(11)
   close(12)
   end
```

Assume the following data in the file `convert_in.txt`:

```
1
2
3
4
5
6
7
8
9
10
```

Define the following symbols:

```
$ def for$convert.big big_endian
$ def for$convert.dat native
```

Then after you run the compiled source program, you get the following output:

```
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.00000
```

And the following data in the big endian output file `convert_out.big`:

```
?
@
@@
@
@
@À
@à
A
A
A
```

And the following data in the native output file `convert_out.dat`:

```
@
A
@A
A
A
ÀA
àA
B
B
 B
```

9.5.3. 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 /CONVERT= `keyword` method (see Section 9.5.5), but has a lower precedence than the logical name method.

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 logical name FOR$CONVERT020 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 logical name method (see Section 9.5.1), 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 method in use.

9.5.4. OPTIONS Statement /CONVERT= `keyword` Method

You can only specify one numeric file format for all unit numbers using this method, unless you also use the logical name or 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 could use one source program and compile it using different FORTRAN commands to create multiple executable programs that each read a certain format.

The logical name or OPEN CONVERT specifier methods take precedence over this method. For instance, you might use the logical name or OPEN CONVERT specifier method to specify each unit number that will use a format other than that specified using the FORTRAN command qualifier method. This method takes precedence over the FORTRAN command /CONVERT qualifier method.

You can use OPTIONS statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding FORTRAN command qualifiers. For example, to use G_float as both the memory format and as the unformatted file format, specify the following OPTIONS statements:

```
OPTIONS /FLOAT=G_FLOAT
OPTIONS /CONVERT=NATIVE
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format using the FORTRAN/CONVERT= `keyword` method alone, unless you use it in combination with the logical name method 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

9.5.5. FORTRAN Command /CONVERT= `keyword` Qualifier Method

You can specify only one numeric format for all unit numbers by using the FORTRAN command qualifier method, unless you also use the logical name method or CONVERT specifier method. You specify the numeric format at compile time and must compile all routines under the same /CONVERT=

`keyword` qualifier (or the equivalent OPTIONS statement). If needed, you can use one source program and compile it using different FORTRAN 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 logical name or OPEN CONVERT specifier method to specify each unit number that will use a format other than that specified using the FORTRAN command qualifier method.

For example, the following DCL 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, S_float and T_float little endian IEEE floating-point format). The created file, VAXD_CONVERT.EXE, is then run:

```
$ FORTRAN/FLOAT=IEEE_FLOAT /CONVERT=VAXD/OBJECT=VAXD_CONVERT.OBJ FILE.F90
$ LINK VAXD_CONVERT
$ RUN VAXD_CONVERT
```

Because this method affects all unit numbers, you cannot read or write data in different formats if you only use the FORTRAN /CONVERT= `keyword` method. To specify a different format for a particular unit number, use the FORTRAN /CONVERT= `keyword` method in combination with the logical name method or the OPEN statement CONVERT specifier method.

9.6. Additional Information on Nonnative Data

The following notes apply 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 2.3.7) of unformatted files. While formatted files are specified in units of characters (bytes), unformatted files are specified in longword units (unless /ASSUME=BYTERECL is specified) for Compaq Fortran 77, VSI 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.”

- Certain vendors apply different OPEN statement defaults to determine the record type. The default record type (RECORDTYPE) with VSI Fortran depends on the values for the ACCESS and FORM specifiers for the OPEN statement, as described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

- On VSI Fortran file characteristics, see Section 6.5.

- On VSI Fortran record types, see Section 6.5.3.

Chapter 10. Using VSI Fortran in the Common Language Environment

This chapter describes:

- Section 10.1: Overview
- Section 10.2: VSI Fortran Procedures and Argument Passing
- Section 10.3: Argument-Passing Mechanisms and Built-In Functions
- Section 10.4: Using the cDEC$ ALIAS and cDEC$ ATTRIBUTES Directives
- Section 10.5: OpenVMS Procedure-Calling Standard
- Section 10.6: OpenVMS System Routines
- Section 10.7: Calling Routines: General Considerations
- Section 10.8: Calling OpenVMS System Services
- Section 10.9: Calling Between Compaq Fortran 77 and VSI Fortran
- Section 10.10: Calling Between VSI Fortran and VSI C

10.1. Overview

VSI Fortran provides you with a variety of mechanisms for gaining access to procedures and system services external to your VSI Fortran programs. By including CALL statements or function references in your source program, you can use procedures such as mathematical functions, OpenVMS system services, and routines written in such languages as Compaq Fortran 77 and VSI C.

The VSI Fortran compiler operates within the OpenVMS common language environment, which defines certain calling procedures and guidelines. These guidelines allow you to use VSI Fortran to call OpenVMS system or library routines and routines written in different languages (usually called mixed-language programming).

This chapter provides information on the OpenVMS procedure-calling standard and how to access OpenVMS system services.

For More Information:

About calling and using the RMS (Record Management Services) system services, see Chapter 11.

10.2. VSI 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 VSI Fortran language, including statement functions and defined assignment statements not described in this manual, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.2.1. 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:

- Better compile-time argument checking and fewer run-time errors
- In some cases, faster run-time performance
- Ease of locating problems in source files since the features help to make the interface self-documenting
- Allows use of some language features that require an explicit interface, such as array function return values.
- When passing certain types of arguments between VSI 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 the passed array descriptor. The array descriptor is generated when an appropriate explicit interface is used for certain types of array arguments.

For More Information:

On VSI Fortran array descriptors, see Section 10.2.7.

10.2.2. Types of VSI 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 10.2.3).

  For subprograms with no explicit interface, declare the subprogram name as external 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.

10.2.3. Using Procedure Interface Blocks

Procedure interface blocks allow you to specify an explicit interface for a subprogram as well as to 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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.
  - 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 1.4.

10.2.4. Passing Arguments and Function Return Values

VSI Fortran uses the same argument-passing conventions as Compaq Fortran 77 on OpenVMS Alpha systems for non-pointer scalar variables and explicit-shape and assumed-size arrays.

When calling VSI Fortran subprograms, be aware that VSI 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 (arguments contain an address).

  An argument contains the address of the data being passed, not the data itself (unless explicitly specified otherwise, such as with the cDEC$ ATTRIBUTES directive or the %VAL built-in function).

  Assumed-shape arrays and deferred-shape arrays are passed by array descriptor.

Character data is passed by character descriptor.

Arguments omitted by adding an extra comma (,) are passed as a zero by immediate value (see Section 10.8.4). Such arguments include OPTIONAL arguments that are not passed.

Any argument specified using the %DESCR built-in function is also passed by descriptor (see Section 10.3).

- 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 by a character descriptor (except for character constant actual arguments when /BY_REF_CALL is used).

  Dummy arguments for character data can use an assumed length.

  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 VSI Fortran, use the CHAR intrinsic function (described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

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 must be in the same order, unless argument keywords are used.

  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 VSI Fortran intrinsic data types are character, integer, logical, real, and complex.

    o convert data from one data type to another, use the appropriate intrinsic procedures described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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 (see Section 10.2.6).

  - 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** (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.

Other rules which apply to passing arrays and pointers are described in Section 10.2.5, Section 10.2.6, and in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

- The means by which the argument is passed and received (passing mechanism) must match.

  By default, VSI Fortran arguments are passed by reference. (See Table 10.1 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 VSI 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 VSI extensions:

  - cDEC$ ATTRIBUTES directive (see Section 10.4.2)
  - Built-in functions (see Section 10.3)

- 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 actual or dummy arguments.

For More Information:

- On passing arguments, function return values, and the contents of registers on OpenVMS systems, see the *VSI OpenVMS Calling Standard*.
- On intrinsic data types, see Chapter 8 and the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On intrinsic procedures and attributes available for array use, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On explicit interfaces and when they are required, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On a VSI Fortran example program that uses an external subprogram and a module that contains a procedure interface block, see Example 1.3.

10.2.5. 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-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.

Certain combinations of actual and dummy array arguments are disallowed.

For More Information:

- On the types of arrays and passing array arguments, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On explicit interfaces and when they are required, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On array descriptors, see Section 10.2.7.

10.2.6. 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 (and must be conformable)
- Dummy arguments have the TARGET attribute
- Both actual and dummy arguments have neither attribute

If you specify an actual argument of type POINTER and a dummy argument of type POINTER, the dummy argument receives the correct pointer value if you specify (in the code containing the actual argument) an appropriate explicit interface that defines the dummy argument with the POINTER attribute and follows certain rules.

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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.2.7. VSI Fortran Array Descriptor Format

When using an explicit interface (by association or procedure interface block), VSI Fortran will generate a descriptor for the following types of dummy argument data structures:

- Pointers to arrays (array pointers)
- Assumed-shape arrays

To allow calling between Compaq Fortran 77 and VSI 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 VSI Fortran, an array descriptor is not used.

When calling between VSI Fortran and a non-Fortran language (such as C), you can specify an appropriate explicit interface or use an implicit interface.

However, for cases where the called routine needs the information in the array descriptor, declare the routine with an assumed-shape argument and an explicit interface.

The array descriptor used by VSI Fortran is the OpenVMS Noncontiguous Array Descriptor as described in the *VSI OpenVMS Calling Standard*. In the DSC$B_AFLAGS byte, bit DSC$V_FL_UNALLOC specifies whether storage has or has not been set for this array. If this bit is set, the array has not yet been allocated.

For example, for 32-bit address access, consider the following array 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 (using 32-bit addresses) would contain the following values:

- length (DSC$W_LENGTH) contains 4.
- dtype (DSC$B_DTYPE) contains DSC$K_DTYPE_L.
- class (DSC$B_CLASS) contains DSC$K_CLASS_NCA.
- pointer (DSC$A_POINTER) contains the address of A (9,1).
- scale (DSC$B_SCALE) contains 0.
- digits (DSC$B_DIGITS) contains 0.
- aflags (DSC$B_AFLAGS) contains 0, since A is allocated, the V_FL_UNALLOC bit is clear.
- dimen (DSC$B_DIMEN) contains 2.
- arsize (DSC$L_ARSIZE) contains 60.
- DSC$A_A0 contains the address of A(0,0).
- DSC$L_S `i` contains the stride of dimension `i`.
- DSC$L_L `i` contains the lower bound of dimension `i`.
- DSC$L_U `i` contains the upper bound of dimension `i`.

For information about the Noncontiguous Array Descriptor when 64-bit addressing is requested (cDEC$ ATTRIBUTES ADDRESS64 directive), see the *VSI OpenVMS Calling Standard*.

10.3. Argument-Passing Mechanisms and Built-In Functions

The OpenVMS procedure-calling standard defines three mechanisms by which arguments are passed to procedures:

- By immediate value: The argument list entry contains the value.
- By reference: The argument list entry contains the address of the value.
- By descriptor: The argument list entry contains the address of a descriptor of the value.

By default, VSI Fortran uses the reference and descriptor mechanisms to pass arguments, depending on each argument's data type:

- The reference mechanism is used to pass all actual arguments that are numeric: logical, integer, real, and complex.
- The descriptor mechanism is used to pass all actual arguments that are character, Fortran 90 pointers, assumed-shape arrays, and deferred-shape arrays (except for character constant actual arguments when the VSI Fortran program was compiled with /BY_REF_CALL).

When a VSI Fortran program needs to call a routine written in a different language (or in some cases a Fortran 90 subprogram), there may be a need to use a form other the VSI Fortran default mechanisms. For example, OpenVMS system services may require that certain numeric arguments be passed by immediate value instead of by reference.

For cases where you cannot use the default passing mechanisms, VSI Fortran provides three built-in functions for passing arguments. It also provides a built-in function for computing addresses for use in argument lists. These built-in functions are:

- %VAL, %REF, %DESCR: Argument list built-in functions
- %LOC: General usage built-in function

Except for the %LOC built-in function, which can be used in any arithmetic expression, these functions can appear only as unparenthesized arguments in argument lists. The three argument list built-in functions and %LOC built-in function are rarely used to call a procedure written in VSI Fortran.

The use of these functions in system service calls is described in Section 10.8.4. The sections that follow describe their use in general.

Instead of using the VSI Fortran built-in functions, you can use the cDEC$ ATTRIBUTES directive to change the VSI Fortran default passing mechanisms (see Section 10.4.2).

10.3.1. Passing Arguments by Descriptor – %DESCR Function

The %DESCR function passes its argument by descriptor. It has the following form:

```
%DESCR(arg)
```

The argument generated by the compiler is the address of a descriptor of the argument (arg). The argument value can be any Fortran 90 expression. The argument value must *not* be a derived type, record

name, record array name, or record array element. The compiler can generate OpenVMS descriptors for all Fortran data types.

In VSI Fortran, the descriptor mechanism is the default for passing character arguments, Fortran 90 pointers, assumed-shape arrays, and deferred-shape arrays. This is because the subprogram may need to know the length or other information about the character, pointer, or array argument. VSI Fortran always generates code to refer to character dummy arguments through the addresses in their character descriptors.

For More Information:

On VSI Fortran array descriptors, see Section 10.2.7.

10.3.2. Passing Addresses – %LOC Function

The %LOC built-in function computes the address of a storage element as an INTEGER (KIND=8) (64-bit) value. With 64-bit addressing (cDEC$ ATTRIBUTE ADDRESS64 directive specified), all 64-bits are used. With 32-bit addressing (cDEC$ ATTRIBUTE ADDRESS64 directive omitted), only the lower 32 bits are used. You can then use this value in an arithmetic expression. It has the following form:

```
%LOC(arg)
```

The %LOC function is particularly useful for certain system services or 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 declaring volatile data structures, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On optimization and declaring volatile data, see Section 5.7.3.
- On an example that uses the %LOC function, see Example 10.4.

10.3.3. Passing Arguments by Immediate Value – %VAL Function

The %VAL function passes the argument list entry as a 64-bit immediate value. It has the following form:

```
%VAL(arg)
```

The argument-list entry generated by the compiler is the value of the argument (arg). The argument value can be a constant, variable, array element, or expression of type INTEGER, LOGICAL, REAL (KIND=4), REAL (KIND=8), COMPLEX (KIND=4), or COMPLEX (KIND=8).

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.

If the value is a byte, word, or longword, it is sign-extended to a quadword (eight bytes).

To produce a zero-extended value rather than a sign-extended value, use the ZEXT intrinsic function.

For More Information:

- On intrinsic procedures, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On an example of passing integer data by value (using %VAL) and by reference (default) to a C function, see Section 10.10.7.
- On the ZEXT intrinsic function, see *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the %VAL built-in function, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.3.4. 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 generated by the compiler 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 VSI Fortran, passing by reference is the default mechanism for numeric values, so the %REF call is usually not needed.

10.3.5. Examples of Argument Passing Built-in Functions

The following examples demonstrate the use of the argument list built-in functions.

1. In this example, the first constant is passed by reference. The second constant is passed by immediate value:

   ```
   CALL SUB(2,%VAL(2))
   ```

2. In this example, the first character variable is passed by character descriptor. The second character variable is passed by reference:

   ```
   CHARACTER(LEN=10) A,B
   CALL SUB(A,%REF(B))
   ```

3. In this example, the first array is passed by reference. The second array is passed by descriptor:

   ```
   INTEGER IARY(20), JARY(20)
   CALL SUB(IARY,%DESCR(JARY))
   ```

10.4. 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 allows you to 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 allows you to 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.

10.4.1. The cDEC$ ALIAS Directive

VSI Fortran now supports the cDEC$ ALIAS directive in the same manner as Compaq Fortran 77. Use this 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 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. The default for the VSI Fortran compiler is to force the name into uppercase.

If *external-name* is a symbolic name, the symbolic name (in uppercase) is used as the external name for the specified internal name. Any other declaration of the specified symbolic name is ignored for the purposes of the ALIAS directive.

For example, in 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. Use single quotation marks (character constant) to specify a case-sensitive name.

This feature can be useful when porting code to systems where 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.

10.4.2. The 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 VSI Fortran routines written in C or Assembler.

The cDEC$ ATTRIBUTES directive takes the following form:

```
cDEC$ ATTRIBUTES att [,att]... :: object [,object]...
```

In this form:

att is one of the keywords listed in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]. For example, C, ALIAS, REFERENCE, VALUE, EXTERN, VARYING, and ADDRESS64.

c is the letter or character (c, C, *, !) that introduces the directive (see *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

object is the name of a data object used as an argument or procedure. Only one object is allowed when using the C and ALIAS properties.

The *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] explains the valid combinations of properties with the various types of objects.

The ATTRIBUTES properties are described in the following sections:

- C Property, Section 10.4.2.1
- ALIAS Property, Section 10.4.2.2
- REFERENCE and VALUE Properties, Section 10.4.2.3
- EXTERN and VARYING Properties, Section 10.4.2.4
- ADDRESS64 Property, Section 10.4.2.5

10.4.2.1. C Property

The C property provides a convenient way for VSI 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.

The C property affects how arguments are passed, as described in Table 10.1.

Table 10.1. 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 (large derived type variables may be passed by reference)
Scalar, with VALUE specified	Passed by value	Passed by value
Scalar, with REFERENCE specified	Passed by reference	Passed by reference
String	Passed by character descriptor	Passes the first character of the string, padded to a full integer
String, with VALUE specified	Error	Passes the first character of the string, padded to a full integer
String, with REFERENCE specified	Passed by reference	Passed by reference
Arrays, including pointers to arrays	Always passed by reference	Always passed by reference

If C is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran conventions pass arguments by reference.

Character arguments are passed as follows:

- If C is specified without REFERENCE for the arguments, the first character of the string is passed (padded with zeros out to INTEGER*4 length).
- If C is specified with REFERENCE for the argument (or if only REFERENCE is specified), the starting address of the string is passed with no descriptor.

Example 10.1 shows VSI Fortran code that calls the C function PNST by using the cDEC$ ATTRIBUTES C directive and C language passing conventions.

Example 10.1. 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 10.2 shows the C function called PNST that is called by the example program shown in Example 10.1

Example 10.2. 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 10.1 and Example 10.2) might be compiled, linked, and run as follows:

```
$ CC PASS_INT_CDEC_C.C
$ FORTRAN PASS_INT_CDEC.F90
$ LINK/EXECUTABLE=PASS_CDEC PASS_INT_CDEC, PASS_INT_CDEC_C
$ RUN PASS_CDEC
99==99 99==          99
```

10.4.2.2. ALIAS Property

You can specify the ALIAS property as cDEC$ ALIAS or as cDEC$ ATTRIBUTES ALIAS; they are equivalent, except that using cDEC$ ALIAS allows symbol names (see Section 10.4.1).

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 10.4.1).

10.4.2.3. 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.

Character values, substrings, and arrays cannot be passed by value. When REFERENCE is specified for a character argument, the string is passed with no descriptor.

VALUE is the default if the C property is specified in the subprogram definition (for scalar data only).

Consider the following free-form example, which passes an integer by value:

```
interface
  subroutine foo (a)
   !DEC$ ATTRIBUTES value :: a
      integer a
  end subroutine foo
end interface
```

This subroutine can be invoked from VSI Fortran using the name foo:

```
 integer i
 i = 1
 call foo(i)

 end program
```

This is the actual subroutine code:

```
  subroutine foo (i)
   !DEC$ ATTRIBUTES value :: i
      integer i
      i = i + 1
      .
      .
  end subroutine foo
```

10.4.2.4. 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.4.2.5. ADDRESS64 Property

Specifies that the object has a 64-bit address. This property can be specified for any variable or dummy argument, including ALLOCATABLE and deferred-shape arrays. However, variables with this property cannot be data-initialized.

It can also be specified for COMMON blocks or for variables in a COMMON block. If specified for a COMMON block variable, the COMMON block implicitly has the ADDRESS64 property.

ADDRESS64 is not compatible with the AUTOMATIC attribute.

For More Information:

- On requirements and the use of 64-bit virtual addresses, see the online release notes.
- On the memory layout of process and system memory, see the *OpenVMS Alpha Guide to 64-Bit Addressing and VLM Features*.
- On passing arguments, argument types, function return values, and the contents of registers on OpenVMS systems, see the *VSI OpenVMS Calling Standard*.
- On VSI Fortran intrinsic data types, see Chapter 8.
- On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the VSI C language, see the *VSI C Language Reference Manual*.
- On the CC command, see the *VSI C User's Guide for OpenVMS Systems*.

10.5. OpenVMS Procedure-Calling Standard

Programs compiled by the VSI Fortran compiler conform to the standard defined for OpenVMS procedure calls (see the *OpenVMS Programming Interfaces: Calling a System Routine* and *VSI OpenVMS Calling Standard*). This standard prescribes how registers and the system-maintained call stack can be used, how function values are returned, how arguments are passed, and how procedures receive and return control.

When writing routines that can be called from VSI Fortran programs, you should give special consideration to the argument list descriptions in Section 10.5.3.

10.5.1. Register and Stack Usage

For information about register and stack usage on I64, see the *VSI OpenVMS Calling Standard*.

10.5.1.1. Register and Stack Usage on Alpha

The Alpha architecture provides 32 general purpose integer registers (R0-R31) and 32 floating-point registers (F0-F31), each 64 bits in length. The *OpenVMS Programming Interfaces: Calling a System Routine* defines the use of these registers, as listed in Table 10.2.

Table 10.2. OpenVMS Alpha Register Usage

Register	Use
R0	Function value return registers; also see F0, F1
R1	Conventional scratch register
R2-R15	Conventional saved registers
R16-R21	Argument registers (one register per argument, additional arguments are placed on the stack)
R22-R24	Conventional scratch registers
R25	Argument information (AI); contains argument count and argument type
R26	Return address (RA) register
R27	Procedure value (PV) register
R28	Volatile scratch register
R29	Frame pointer (FP)
R30	Stack pointer (SP)
R31	Read As Zero/Sink (RZ) register
PC	Program counter (PC), a special register that addresses the instruction stream, which is not accessible as an integer register
F0, F1	Function value return registers (F1 is used for the imaginary part of COMPLEX)
F2-F9	Conventional saved registers
F10-F15	Conventional scratch registers
F16-F21	Argument registers (one per argument, additional arguments are placed on the stack)
F22-F30	Conventional scratch registers
F31	Read As Zero/Sink (RZ) register

A **stack** is defined as a LIFO (last-in/first-out) temporary storage area that the system allocates for every user process.

Each time you call a routine, the system places information on the stack in the form of procedure context structures, as described in the *VSI OpenVMS Calling Standard*.

10.5.2. Return Values of Procedures

A procedure is a VSI Fortran subprogram that performs one or more computations for other programs. Procedures can be either functions or subroutines. Both functions and subroutines can return values by storing them in variables specified in the argument list or in common blocks.

A function, unlike a subroutine, can also return a value to the calling program by assigning the value to the function's name. The method that function procedures use to return values depends on the data type of the value, as summarized in Table 10.3.

Table 10.3. OpenVMS Alpha Function Return Values

Data Type	Return Method
{Logical | Integer}	R0
REAL (KIND=4)	F0

Data Type	Return Method
REAL (KIND=8)	F0
REAL (KIND=16)	F0 and F1
COMPLEX (KIND=4) (COMPLEX*8)	F0 (real part), F1 (imaginary part)
COMPLEX (KIND=8) (COMPLEX*16)	F0 (real part), F1 (imaginary part)
COMPLEX (KIND=16) (COMPLEX*32)	In addition to the arguments, an entry is added to the beginning of the argument list. This additional entry contains the address of the character string descriptor. At run time, before the call, the calling program allocates enough storage to contain the result and places the storage address in the descriptor.
Character	In addition to the arguments, an entry is added to the beginning of the argument list. This additional entry contains the address of the character string descriptor. At run time, before the call, the calling program allocates enough storage to contain the result and places the storage address in the descriptor.
Pointers	R0 contains the address of the array descriptor (see Section 10.2.6).
Assumed-shape arrays, Deferred-shape arrays	R0 contains the address of the array descriptor (see Section 10.2.5).

For More Information:

- On VSI Fortran array descriptors, see Section 10.2.7.
- On defining and invoking subprograms, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.5.3. Argument Lists

Use an argument list to pass information to a routine and receive results.

The *VSI OpenVMS Calling Standard* defines an argument list as an **argument item sequence**, consisting of the first six arguments occupying six integer and six floating-point registers (R16-R21 and F16-F21), with additional arguments placed on the stack. The argument information is contained in R25 (AI register). The stack pointer is contained in R30. For more details on argument lists, see the *VSI OpenVMS Calling Standard*.

Memory for VSI Fortran argument lists and for OpenVMS Alpha descriptors is allocated dynamically on the stack.

OpenVMS Alpha descriptors are generated from the use of the %DESCR function or by passing CHARACTER data, Fortran 90 pointers, and certain types of arrays (see Section 10.2.7).

Omitted arguments – for example, CALL X(A, ,B) – are represented by an argument passed by value that has a value of zero. This is a VSI extension to the Fortran 90 standard.

Fortran optional arguments (OPTIONAL attribute) are also represented by an argument passed by value that has a value of zero.

For More Information:

On using Fortran language standards to specify arguments, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.6. OpenVMS System Routines

System routines are OpenVMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that VSI Fortran supports the data structures required to call the routine (in a FORSYSDEF library module) or you define them yourself.

The system routines used most often are OpenVMS Run-Time Library routines and system services. System routines are documented in detail in the *VMS Run-Time Library Routines Volume* and the *OpenVMS System Services Reference Manual*.

10.6.1. OpenVMS Run-Time Library Routines

The OpenVMS Run-Time Library provides commonly-used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular OpenVMS Run-Time Library facility. Table 10.4 lists all of the language-independent Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 10.4. Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
CVT$	Library routines that handle floating-point data conversion
DTK$	DECtalk routines that are used to control VSI's DECtalk device
LIB$	Library routines that: Obtain records from devices Manipulate strings Convert data types for I/O Allocate resources Obtain system information Signal exceptions Establish condition handlers Enable detection of hardware exceptions Process cross-reference data
MATH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations
OTS$	General-purpose routines that perform such tasks as data type conversions as part of a compiler's generated code
PPL$	Parallel processing routines that help you implement concurrent programs on single-CPU and multiprocessor systems
SMG$	Screen-management routines that are used in designing, composing, and keeping track of complex images on a video screen

Facility Prefix	Types of Tasks Performed
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings

10.6.2. OpenVMS System Services Routines

System services are system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the OpenVMS Run-Time Library routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). However, these services are logically divided into groups that perform similar tasks. Table 10.5 describes these groups.

Table 10.5. System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of ASTs
Change Mode	Changes the access mode of particular routines
Condition Handling	Designates condition handlers for special purposes
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters
Input/Output	Performs I/O directly, without going through OpenVMS RMS
Lock Management	Enables processes to coordinate access to shareable system resources
Logical Names	Provides methods of accessing and maintaining pairs of character string logical names and equivalence names
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data
Process Control	Creates, deletes, and controls execution of processes
Process Information	Returns information about processes
Security	Enhances the security of OpenVMS systems
Timer and Time Conversion	Schedules events, and obtains and formats binary time values

10.7. Calling Routines: General Considerations

The basic steps for calling routines are the same whether you are calling a routine written in VSI Fortran, a routine written in some other OpenVMS language, a system service, or a VSI Fortran RTL routine routine.

To call a subroutine, use the CALL statement.

To call a function, reference the function name in an expression or as an argument in another routine call.

In any case, you must specify the name of the routine being called and all non-optional arguments required for that routine. Make sure the data types and passing mechanisms for the actual arguments you are passing coincide with those declared in the routine. (See Table 10.6 for information on OpenVMS

data types or *OpenVMS Programming Interfaces: Calling a System Routine* for data types needed for mixed language programming.)

If you do not want to specify a value for a required parameter, you can pass a null argument by inserting a comma (,) as a placeholder in the argument list. If the routine requires any passing mechanism other than the default, you must specify the passing mechanism in the CALL statement or the function call.

Example 10.3 illustrates calling an OpenVMS RTL LIB$ routine. This example uses the LIB$GET_VM RTL routine and the Compaq Fortran 77 POINTER statement to allocate memory for an array, and uses VSI extensions (POINTER statement and LIB$ routine) to allocate virtual memory.

Example 10.3. Use of LIB$GET_VM and POINTER

```
! Program accepts an integer and displays square root values

  INTEGER (KIND=4) N
  READ (5,*) N            ! Request typed integer value
  CALL MAT(N)
  END

! Subroutine MAT uses the typed integer value to display the square
! root values of numbers from 1 to N (the typed number)

  SUBROUTINE MAT(N)
  REAL I(1000)           ! Fixed 1000 dimension allows bounds checking
  INTEGER SIZE,STATUS
  POINTER (P,I)          ! HP Fortran 77 POINTER statement establishes
                         ! P as the pointer and array variable I as the
                         ! pointee. P will receive memory base address
                         ! from LIB$GET_VM.

  SIZE=SIZEOF(I(1)) * N  ! Array I contains values calculated in loop
                         ! below.
                         ! Intrinsic SIZEOF returns size of memory
                         ! to allocate.

  STATUS = LIB$GET_VM(SIZE,P)    ! Allocate memory
  IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

  DO J=1,N
     I(J) = SQRT(FLOAT(J))   ! Intrinsic FLOAT converts integer to REAL.
  ENDDO
  TYPE *, (I(J),J=1,N)       ! Display calculated values

  STATUS = LIB$FREE_VM(SIZE,P)   ! Deallocate memory
  IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

  END SUBROUTINE MAT
```

The following commands show how to compile, link, and run the program and how it displays the square root of numbers from 1 to 4 during execution:

```
$ FORTRAN SQUARE_ROOT
$ LINK SQUARE_ROOT
$ RUN SQUARE_ROOT

4
```

```
 1.000000       1.414214       1.732051       2.000000
```

The call to LIB$GET_VM as a function reference allocates memory and returns the starting address in variable P. The return status (variable STATUS) is tested to determine whether an error should be signaled using a subroutine call to LIB$SIGNAL, passing the value of the variable status (not its address) using the %VAL built-in function.

For More Information:

- On intrinsic procedures (such as FLOAT, SQRT, and SIZEOF) and the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the %VAL built-in function, see Section 10.3.3.
- On an example of memory allocation using Fortran 90 standard-conforming allocatable arrays, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

10.8. Calling OpenVMS System Services

You can invoke system services in a VSI Fortran program with a function reference or a subroutine CALL statement that specifies the system service you want to use. To specify a system service, use the form:

```
SYS$service-name(arg,...,arg)
```

You pass arguments to the system services according to the requirements of the particular service you are calling; the service may require an immediate value, an address, the address of a descriptor, or the address of a data structure. Section 10.8.4 describes the VSI Fortran syntax rules for each of these cases. See the *VSI OpenVMS System Services Reference Manual* for a full definition of individual services.

The basic steps for calling system services are the same as those for calling any external routine. However, when calling system services (or Run-Time Library routines), additional information is often required. The sections that follow describe these requirements.

10.8.1. Obtaining Values for System Symbols

OpenVMS uses symbolic names to identify the following values or codes for system services:

- Return status values are used for testing the success of system service calls.
- Condition values are used for error recovery procedures (see Chapter 14).
- Function codes are the symbolic values used as input arguments to system service calls.

The values chosen determine the specific action performed by the service.

The *VSI OpenVMS System Services Reference Manual* describes the symbols that are used with each system service. The *VSI OpenVMS I/O User's Reference Manual* describes the symbols that are used with I/O-related services.

The VSI Fortran symbolic definition library FORSYSDEF contains VSI Fortran source definitions for related groups of system symbols. Each related group of system symbols is stored in a separate text library module; for example, the library module $IODEF in FORSYSDEF contains PARAMETER statements that define the I/O function codes.

The library modules in FORSYSDEF correspond to the symbolic definition macros that OpenVMS MACRO programmers use to define system symbols. The library modules have the same names as the macros and contain VSI Fortran source code, which is functionally equivalent to the MACRO source code. To determine whether you need to include other symbol definitions for the system service you want to use, refer to the documentation for that particular system service. If the documentation states that values are defined in a macro, you must include those symbol definitions in your program.

For example, the description for the *flags* argument in the SYS$MGBLSC (Map Global Section) system service states that “Symbolic names for the flag bits are defined by the $SECDEF macro.” Therefore, when you call SYS$MGBLSC, you must include the definitions provided in the $SECDEF macro.

Library module $SYSSRVNAM in FORSYSDEF contains declarations for all system-service names. It contains the necessary INTEGER and EXTERNAL declarations for the system-service names. (The library module, once extracted from FORSYSDEF.TLB, also contains comments describing the arguments for each of the system services.) Also, library module $SSDEF contains system-service return status codes and is generally required whenever you access any of the services.

The library modules in FORSYSDEF contain definitions for constants, bit masks, and data structures. See Section 10.8.4.4 for a description of how to create data structure arguments in VSI Fortran. Refer to Appendix E for a list of library modules that are in FORSYSDEF.

You can access the library modules in the FORSYSDEF library with the INCLUDE statement, using the following format:

```
INCLUDE '(library-module-name)'
```

The notation `library-module-name` represents the name of a library module contained in FORSYSDEF. The library FORSYSDEF is searched if the specified library module was not found in a previously searched library.

10.8.2. Calling System Services by Function Reference

In most cases, you should check the return status after calling a system service. Therefore, you should invoke system services by function reference rather than by issuing a call to a subroutine.

For example:

```
INCLUDE '($SSDEF)'
INCLUDE '($SYSSRVNAM)'
INTEGER (KIND=2) CHANNEL
   .
   .
   .
MBX_STATUS = SYS$CREMBX(,CHANNEL,,,,,'MAILBOX')
IF (MBX_STATUS .NE. SS$_NORMAL) GO TO 100
```

In this example, the system service referenced is the Create Mailbox system service. An INTEGER (KIND=2) variable (CHANNEL) is declared to receive the channel number.

The function reference allows a return status value to be stored in the variable MBX_STATUS, which can then be checked for correct completion on return. If the function's return status is not

SS$_NORMAL, failure is indicated and control is transferred to statement 100. At that point, some form of error processing can be undertaken.

You can also test the return status of a system service as a logical value. The status codes are defined so that when they are tested as logical values, successful codes have the value true and error codes have the value false. The last line in the preceding example could then be changed to the following:

```
IF (.NOT. MBX_STATUS) GO TO 100
```

Refer to the *VSI OpenVMS System Services Reference Manual* for information concerning return status codes. The return status codes are included in the description of each system service.

10.8.3. Calling System Services as Subroutines

Subroutine calls to system services are made like other subroutine calls. For example, to call the Create Mailbox system service, issue a call to SYS$CREMBX, passing the appropriate arguments to it, as follows:

```
CALL SYS$CREMBX(,CHANNEL,,,,,'MAILBOX')
```

This call corresponds to the function reference described in Section 10.8.2. The main difference is that the status code returned by the system service is not tested. For this reason, you should avoid this method and use a function reference when calling system services whenever the service could fail for any reason.

10.8.4. Passing Arguments to System Services

The description of each system service in the *VSI OpenVMS System Services Reference Manual* specifies the argument-passing method for each argument. Four methods are supported:

- By immediate value
- By address: this is the VSI Fortran default and is termed “by reference”
- By descriptor: this is the VSI Fortran default for CHARACTER arguments, Fortran 90 pointers, and certain types of arrays (see Section 10.2.7)
- By data structure

These methods are discussed separately in Section 10.8.4.1 through Section 10.8.4.4.

You can determine the arguments required by a system service from the service description in the *VSI OpenVMS System Services Reference Manual*. Each system service description indicates the service name, the number of arguments required, and the positional dependency of each argument.

Table 10.6 lists the VSI Fortran declarations that you can use to pass any of the standard OpenVMS data types as arguments. OpenVMS data types are defined in *OpenVMS Programming Interfaces: Calling a System Routine*.

Instead of using record structure declarations for most standard OpenVMS data types, you can consider using derived-type structures if you use the SEQUENCE statement and are careful of alignment. However, RMS control block structures must be declared as record structures (STRUCTURE statement).

Table 10.6. VSI Fortran Implementation of OpenVMS Data Types

OpenVMS Data Type	VSI Fortran Declaration
access_bit_names	INTEGER (KIND=4) (2,32) or STRUCTURE /access_bit_names/ INTEGER (KIND=4) access_name_len INTEGER (KIND=4) access_name_buf END STRUCTURE !access_bit_names RECORD /access_bit_names/ my_names(32)
access_mode	BYTE or INTEGER (KIND=1)
address	INTEGER (KIND=4)
address_range	INTEGER (KIND=4) (2) or STRUCTURE /address_range/ INTEGER (KIND=4) low_address INTEGER (KIND=4) high_address END STRUCTURE
arg_list	INTEGER (KIND=4) (n)
ast_procedure	EXTERNAL
boolean	LOGICAL (KIND=4)
byte_signed	BYTE or INTEGER (KIND=1)
byte_unsigned	BYTE or INTEGER (KIND=1) [1]
channel	INTEGER (KIND=2)
char_string	CHARACTER (LEN= *n*) [2]
complex_number	COMPLEX (KIND=4)[3] COMPLEX (KIND=8) [3]
cond_value	INTEGER (KIND=4)
context	INTEGER (KIND=4)
date_time	INTEGER (KIND=8)
device_name	CHARACTER (LEN= *n*)
ef_cluster_name	CHARACTER (LEN= *n*)
ef_number	INTEGER (KIND=4)
exit_handler_block	STRUCTURE /exhblock/ INTEGER (KIND=4) flink

OpenVMS Data Type	VSI Fortran Declaration
	INTEGER (KIND=4) exit_handler_addr BYTE (3) %FILL BYTE arg_count INTEGER (KIND=4) cond_value ! . ! .(optional arguments … ! . one argument per longword) ! END STRUCTURE !cntrlblk RECORD /exhblock/ myexh_block
fab	INCLUDE '($FABDEF)' RECORD /FABDEF/ myfab
file_protection	INTEGER (KIND=4)
floating_point	REAL (KIND=4)[3] REAL (KIND=8)[3] DOUBLE PRECISION[3] REAL (KIND=16)[3]
function_code	INTEGER (KIND=4)
identifier	INTEGER (KIND=4)
invo_context_blk	INCLUDE '($LIBICB)' RECORD /INVO_CONTEXT_BLK/ invo_context_blk
invo_handle	INTEGER (KIND=4)
io_status_block	STRUCTURE /iosb/ INTEGER (KIND=2) iostat, ! return status INTEGER (KIND=2) term_offset, ! Loc. of terminator INTEGER (KIND=2) terminator, ! terminator value INTEGER (KIND=2) term_size ! terminator size END STRUCTURE RECORD /iosb/ my_iosb
item_list_2	STRUCTURE /itmlst/ UNION MAP

OpenVMS Data Type	VSI Fortran Declaration
	INTEGER (KIND=2) buflen,code INTEGER (KIND=4) bufadr END MAP MAP INTEGER (KIND=4) end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2 (n) (Allocate *n* records, where *n* is the number item codes plus an extra element for the end-of-list item.)
item_list_3	STRUCTURE /itmlst/ UNION MAP INTEGER (KIND=2) buflen,code INTEGER (KIND=4) bufadr,retlenadr END MAP MAP INTEGER (KIND=4) end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_3 (n) (Allocate *n* records where *n* is the number item codes plus an extra element for the end-of-list item.)
item_list_pair	STRUCTURE /itmlist_pair/ UNION MAP INTEGER (KIND=4) code

OpenVMS Data Type	VSI Fortran Declaration
	INTEGER (KIND=4) value END MAP MAP INTEGER (KIND=4) end_list /0/ END MAP END UNION END STRUCTURE !itmlst_pair RECORD /itmlst_pair/ my_itmlst_pair (n) (Allocate *n* records where *n* is the number item codes plus an extra element for the end-of-list item.)
item_quota_list	STRUCTURE /item_quota_list/ MAP BYTE quota_name INTEGER (KIND=4) quota_value END MAP MAP BYTE end_quota_list END MAP END STRUCTURE !item_quota_list
lock_id	INTEGER (KIND=4)
lock_status_block	STRUCTURE/lksb/ INTEGER (KIND=2) cond_value INTEGER (KIND=2) unused INTEGER (KIND=4) lock_id BYTE(16) END STRUCTURE !lksb RECORD /lksb/ my_lksb
lock_value_block	BYTE(16)
logical_name	CHARACTER (LEN= *n*)

OpenVMS Data Type	**VSI Fortran Declaration**
longword_signed	INTEGER (KIND=4)
longword_unsigned	INTEGER (KIND=4) [1]
mask_byte	INTEGER (KIND=1) ! (or BYTE)
mask_longword	INTEGER (KIND=4)
mask_quadword	INTEGER (KIND=8)
mask_word	INTEGER (KIND=2)
mechanism_args	INCLUDE '($CHFDEF)' RECORD /CHFDEF2/ mechargs ! (For more information, see Section 14.6).
null_arg	%VAL (0) ! (or an unspecified optional argument)
octaword_signed	INTEGER (KIND=4) (4)
octaword_unsigned	INTEGER (KIND=4) (4)[1]
page_protection	INTEGER (KIND=4)
procedure	INTEGER (KIND=4)
process_id	INTEGER (KIND=4)
process_name	CHARACTER (LEN= *n*)
quadword_signed	INTEGER (KIND=8)
quadword_unsigned	INTEGER (KIND=8)[1]
rights_holder	STRUCTURE /rights_holder/ INTEGER (KIND=4) rights_id INTEGER (KIND=4) rights_mask END STRUCTURE !rights_holder RECORD /rights_holder/ my_rights_holder
rights_id	INTEGER (KIND=4)
rab	INCLUDE '($RABDEF)' RECORD /RABDEF/ myrab
section_id	INTEGER (KIND=4) (2) or INTEGER (KIND=8)
section_name	CHARACTER (LEN= *n*)
system_access_id	INTEGER (KIND=4) (2) or INTEGER (KIND=8)
time_name	CHARACTER (LEN=23)
transaction_id	INTEGER (KIND=4)(4)
uic	INTEGER (KIND=4)
user_arg	Any longword quantity
varying_arg	Any appropriate type
vector_byte_signed	BYTE (n)

OpenVMS Data Type	VSI Fortran Declaration
vector_byte_unsigned	BYTE (n)[1]
vector_longword_signed	INTEGER(KIND=4) (n)
vector_longword_unsigned	INTEGER(KIND=4) (n)[1]
vector_quadword_signed	INTEGER(KIND=8) (n)
vector_quadword_unsigned	INTEGER (KIND=8) (n)[1]
vector_word_signed	INTEGER (KIND=2) (n)
vector_word_unsigned	INTEGER (KIND=2) (n)[1]
word_signed	INTEGER (KIND=2) (n)
word_unsigned	INTEGER (KIND=2) (n)[1]

[1]Unsigned data types are not directly supported by VSI Fortran. However, in most cases you can substitute the signed equivalent, provided you do not exceed the range of the signed data structure.

[2]Where *n* can range from 1 to 65535.

[3]The format used by floating-point (KIND=4) and (KIND=8) data in memory is determined by the FORTRAN command qualifier /FLOAT.

Many arguments to system services are optional. However, if you omit an optional argument, you must include a comma (,) to indicate the absence of that argument. For example, the SYS$TRNLNM system service takes five arguments. If you omit the first and the last two arguments, you must include commas to indicate their existence, as follows:

```
ISTAT = SYS$TRNLNM(,'LNM$FILE_DEV','LOGNAM',,)
```

An invalid reference results if you specify the arguments as follows:

```
STAT = SYS$TRNLNM('LOGNAM',LENGTH,BUFFA)
```

This reference provides only three arguments, not the required five.

When you omit an optional argument by including an extra comma, the compiler supplies a default value of zero (passed by immediate value).

10.8.4.1. Immediate Value Arguments

Value arguments are typically used when the description of the system service specifies that the argument is a “number,” “mask,” “mode,” “value,” “code,” or “indicator.” You must use either the cDEC$ ATTRIBUTES VALUE (see Section 10.4.2.3) or the %VAL built-in function (see Section 10.3.3) whenever this method is required.

Immediate value arguments are used for input arguments only.

10.8.4.2. Address Arguments

Use address arguments when the description of the system service specifies that the argument is “the address of.” (However, refer to Section 10.8.4.3 to determine what to do when “the address of a descriptor” is specified.) In VSI Fortran, this argument-passing method is called “by reference.”

Because this method is the VSI Fortran default for passing numeric arguments, you need to specify the cDEC$ ATTRIBUTES REFERENCE directive (see Section 10.4.2.3) or the %REF built-in function only when the data type of the argument is character.

The argument description also gives the hardware data type required.

For arguments described as “address of an entry mask” or “address of a routine,” declare the argument value as an external procedure. For example, if a system service requires the address of a routine and you want to specify the routine HANDLER3, include the following statement in the declarations portion of your program:

```
EXTERNAL HANDLER3
```

This specification defines the address of the routine for use as an input argument.

Address arguments are used for both input and output:

- For input arguments that refer to byte, word, longword, or quadword values, you can specify either constants or variables. If you specify a variable, you must declare it to be equal to or longer than the data type required. Table 10.7 lists the variable data type requirements for both input and output arguments.

- For output arguments you must declare a variable of exactly the length required to avoid including extraneous data. If, for example, the system returns a byte value in a word-length variable, the leftmost eight bits of the variable are not overwritten on output. The variable, therefore, might not contain the data you expect.

  To store output produced by system services, you must allocate sufficient space to contain the output. You make this allocation by declaring variables of the proper size. For an illustration, refer to the Translate Logical Name system service example in Section 10.8.4.3. This service returns the length of the equivalent name string as a 2-byte value.

  If the output is a quadword value, you must declare a variable of the proper dimensions. For example, to use the Get Time system service (SYS$GETTIM), which returns the time as a quadword binary value, declare the following:

  ```
  INCLUDE '($SYSSRVNAM)'
  INTEGER (KIND=8) SYSTIM
       .
       .
       .
  ISTAT = SYS$GETTIM(SYSTIM)
  ```

  The type declaration INTEGER (KIND=8) SYSTIM establishes a variable consisting of one quadword, which is then used to store the time value.

Table 10.7. Variable Data Type Requirements

OpenVMS Type Required	Input Argument Declaration	Output Argument Declaration
Byte	BYTE, INTEGER (KIND=1), INTEGER (KIND=2), INTEGER (KIND=4)	BYTE, INTEGER (KIND=1)
Word	INTEGER (KIND=2), INTEGER (KIND=4)	INTEGER (KIND=2)
Longword	INTEGER (KIND=4)	INTEGER (KIND=4)
Quadword	INTEGER (KIND=8) or INTEGER (KIND=4) (2) or properly dimensioned array	INTEGER (KIND=8) or INTEGER (KIND=4) (2) or properly dimensioned array
Indicator	LOGICAL	

OpenVMS Type Required	Input Argument Declaration	Output Argument Declaration
Character string descriptor	CHARACTER (LEN= *n*)	CHARACTER (LEN= *n*)
Entry mask or routine	EXTERNAL	

10.8.4.3. Descriptor Arguments

Descriptor arguments are used for input and output of character strings. Use a descriptor argument when the argument description specifies “address of a descriptor.” Because this method is the VSI Fortran default for character arguments, you need to specify the %DESCR built-in function only when the data type of the argument is not character.

On input, a character constant, variable, array element, or expression is passed to the system service by descriptor. On output, two items are needed:

- The character variable or array element to hold the output string
- An INTEGER (KIND=2) variable that is set to the actual length of the output string

In the following example of the Translate Logical Name system service (SYS$TRNLNM), the logical name LOGNAM is translated to its associated name or file specification. The output string and string length are stored in the variables EQV_BUFFER and W_NAMELEN, respectively:

```
INCLUDE '($LNMDEF)'
INCLUDE '($SYSSRVNAM)'

STRUCTURE /LIST/
  INTEGER (KIND=2) BUF_LEN/255/
  INTEGER (KIND=2) ITEM_CODE/LNM$_STRING/
  INTEGER (KIND=4) TRANS_LOG
  INTEGER (KIND=4) TRANS_LEN
  INTEGER (KIND=4) END_ENTRY/0/
END STRUCTURE    !LIST

CHARACTER (LEN=255) EQV_BUFFER
INTEGER (KIND=2)  W_NAMELEN

RECORD/LIST/ ITEM_LIST
ITEM_LIST.TRANS_LOG = %LOC(EQV_BUFFER)
ITEM_LIST.TRANS_LEN = %LOC(W_NAMELEN)

ISTAT = SYS$TRNLNM(, 'LNM$FILE_DEV', 'FOR$SRC', ,ITEM_LIST)
IF (ISTAT) PRINT *, EQV_BUFFER(:W_NAMELEN)
END
```

10.8.4.4. Data Structure Arguments

Data structure arguments are used when the argument description specifies “address of a list,” “address of a control block,” or “address of a vector.” The data structures required for these arguments are constructed in VSI Fortran with structure declaration blocks and the RECORD statement. The storage declared by a RECORD statement is allocated in exactly the order given in the structure declaration, with no space between adjacent items. For example, the item list required for the SYS$GETJPI (or SYS$GETJPIW) system service requires a sequence of items of two words and two longwords each. By declaring each item as part of a structure, you ensure that the fields and items are allocated contiguously:

```
STRUCTURE /GETJPI_STR/
        INTEGER (KIND=2) BUFLEN, ITMCOD
        INTEGER (KIND=4) BUFADR, RETLEN
END STRUCTURE
...
RECORD /GETJPI_STR/ LIST(5)
```

If a given field is provided as input to the system service, the calling program must fill the field before the system service is called. You can accomplish this with data initialization (for fields with values that are known at compile time) and with assignment statements (for fields that must be computed).

When the data structure description requires a field that must be filled with an address value, use the %LOC built-in function to generate the desired address (see Section 10.3.2). When the description requires a field that must be filled with a symbolic value (system-service function code), you can define the value of the symbol by the method described in Section 10.8.1.

10.8.4.5. Examples of Passing Arguments

Example 10.4 shows a complete subroutine that uses a data structure argument to the SYS$GETJPIW system service.

Example 10.4. Subroutine Using a Data Structure Argument

```
!  Subroutine to obtain absolute and incremental values of process
 parameters:
!  CPU time, Buffered I/O count, Direct I/O count, Page faults.

      SUBROUTINE PROCESS_INFO(ABS_VALUES, INCR_VALUES)

!  Declare the arguments and working storage

      INTEGER (KIND=4) ABS_VALUES(4), INCR_VALUES(4), LCL_VALUES(4)
      INTEGER (KIND=4) STATUS, I

!  Declare the SYS$GETJPIW item list data structure in a structure
 declaration

      STRUCTURE /GETJPI_STR/
              INTEGER (KIND=2)  BUFLEN /4/, ITMCOD /0/
              INTEGER (KIND=4)  BUFADR, RETLEN /0/
      END STRUCTURE

!  Create a record with the fields defined in the structure declaration

      RECORD /GETJPI_STR/ LIST(5)

!  Declare the structure of an I/O Status Block

      STRUCTURE /IOSB_STR/
           INTEGER (KIND=4) STATUS, RESERVED
      END STRUCTURE

!  Declare the I/O Status Block

      RECORD /IOSB_STR/ IOSB

!  Assign all static values in the item list
```

```
      LIST(1).ITMCOD = JPI$_CPUTIM
      LIST(2).ITMCOD = JPI$_BUFIO
      LIST(3).ITMCOD = JPI$_DIRIO
      LIST(4).ITMCOD = JPI$_PAGEFLTS

!  Assign all item fields requiring addresses

      LIST(1).BUFADR = %LOC(LCL_VALUES(1))
      LIST(2).BUFADR = %LOC(LCL_VALUES(2))
      LIST(3).BUFADR = %LOC(LCL_VALUES(3))
      LIST(4).BUFADR = %LOC(LCL_VALUES(4))

      STATUS = SYS$GETJPIW(,,,LIST,IOSB,,)  ! Perform system service call

      IF (STATUS) STATUS = IOSB.STATUS      ! Check completion status
      IF (.NOT. STATUS) CALL EXIT (STATUS)

!  Assign the new values to the arguments

      DO I=1,4
         INCR_VALUES(I) = LCL_VALUES(I) - ABS_VALUES(I)
         ABS_VALUES(I)  = LCL_VALUES(I)
      END DO
      RETURN

      END SUBROUTINE PROCESS_INFO
```

Example 10.5 is an example of the typical use of an I/O system service. The program invokes SYS$QIOW to enable Ctrl/C trapping. When the program runs, it prints an informational message whenever it is interrupted by a **Ctrl/C**, and then it continues execution.

Example 10.5. Ctrl/C Trapping Example

```
   PROGRAM TRAPC
   INCLUDE '($SYSSRVNAM)'
   INTEGER (KIND=2) TT_CHAN
   COMMON TT_CHAN
   CHARACTER (LEN=40) LINE

!  Assign the I/O channel.  If unsuccessful stop; otherwise
!  initialize the trap routine.

   ISTAT = SYS$ASSIGN ('TT',TT_CHAN,,)
   IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
   CALL ENABLE_CTRLC

!     Read a line of input and echo it

 10 READ (5,'(A)',END=999) LINE
    TYPE *, 'LINE READ: ', LINE
    GO TO 10
999 END

    SUBROUTINE ENABLE_CTRLC
    INTEGER (KIND=2) TT_CHAN
    COMMON TT_CHAN
    EXTERNAL CTRLC_ROUT
```

```
!   Include I/O symbols

    INCLUDE '($IODEF)'
    INCLUDE '($SYSSRVNAM)'

!     Enable Ctrl/C trapping and specify CTRLC_ROUT
!     as routine to be called when Ctrl/C occurs

    ISTAT = SYS$QIOW( ,%VAL(TT_CHAN), %VAL(IO$_SETMODE .OR.
                      IO$M_CTRLCAST),&
        ,,,CTRLC_ROUT,,%VAL(3),,,)
    IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
    RETURN
    END SUBROUTINE ENABLE_CTRLC

    SUBROUTINE CTRLC_ROUT
    PRINT *, 'Ctrl/C pressed'
    CALL ENABLE_CTRLC
    RETURN

    END SUBROUTINE CTRLC_ROUT
```

For more examples of calling system services and RTL routines, see Appendix F.

10.9. Calling Between Compaq Fortran 77 and VSI Fortran

On OpenVMS Alpha systems, you can call a Compaq Fortran 77 subprogram from VSI Fortran or call a VSI Fortran subprogram from Compaq Fortran 77 (with very few exceptions). A Compaq Fortran 77 procedure and a VSI Fortran procedure can also perform I/O to the same unit number.

10.9.1. Argument Passing and Function Return Values

The recommended rules for passing arguments and function return values between Compaq Fortran 77 and VSI Fortran procedures are as follows:

- If possible, express the following VSI Fortran features with the Compaq Fortran 77 language:
  - Function references
  - CALL statements
  - Function definitions
  - Subroutine definitions

  Avoid using VSI Fortran language features not available in Compaq Fortran 77. Since VSI 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 VSI Fortran have equivalent Compaq Fortran 77 data types. The following VSI Fortran features should not be used between VSI Fortran and Compaq Fortran 77 procedures, because they are not supported by Compaq Fortran 77:

- Derived-type (user-defined) data, which has no equivalents in Compaq Fortran 77.

  VSI Fortran record structures are supported by Compaq Fortran 77 and VSI Fortran as an extension to the Fortran 90 standard. Thus, you can use Compaq Fortran 77 record structures in both VSI Fortran and Compaq Fortran 77.

- VSI Fortran data with the POINTER attribute, which has no equivalents in Compaq Fortran 77. The pointer data type supported by Compaq Fortran 77 is not equivalent to Fortran 90 pointer data.

  Because VSI Fortran supports the pointer data type supported by Compaq Fortran 77, you can use Compaq Fortran 77 pointer data types in both VSI Fortran and Compaq Fortran 77. (In some cases, you can create Compaq Fortran 77 pointer data in a VSI Fortran procedure using the %LOC function).

  VSI 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 VSI Fortran array with the POINTER attribute (see Section 10.2.7).

- VSI Fortran assumed-shape arrays.

  VSI 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 VSI Fortran assumed-shape array (see Section 10.2.7).

For more information on how VSI Fortran handles arguments and function return values, see Section 10.2.4.

- Make sure the sizes of INTEGER, LOGICAL, REAL, and COMPLEX declarations match.

  For example, VSI 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 VSI Fortran declaration of COMPLEX (KIND=4) matches a Compaq Fortran 77 declaration of COMPLEX*8; COMPLEX (KIND=8) matches COMPLEX*16.

  Your source programs may contain INTEGER, LOGICAL, REAL, or COMPLEX declarations without a kind parameter (or size specifier). In this case, when compiling the VSI Fortran procedures (FORTRAN command) and Compaq Fortran 77 procedures (FORTRAN/OLDF77 command), either omit or specify the equivalent qualifiers for controlling the sizes of these declarations.

  For more information on these qualifiers, see Section 2.3.26 for INTEGER and LOGICAL declarations, Section 2.3.37 for REAL and COMPLEX declarations, and Section 2.3.17 for DOUBLE PRECISION declarations.

- VSI Fortran uses the same argument-passing conventions as Compaq Fortran 77 on OpenVMS Alpha systems (see Section 10.2.4).

- You can return nearly all function return values from a VSI Fortran function to a calling Compaq Fortran 77 routine, with the following exceptions:

  - You cannot return VSI Fortran pointer data from VSI Fortran to a Compaq Fortran 77 calling routine.

  - You cannot return VSI Fortran user-defined data types from a VSI Fortran function to a Compaq Fortran 77 calling routine.

Example 10.6 and Example 10.7 show passing an array from a VSI Fortran program to a Compaq Fortran 77 subroutine that prints its value.

Example 10.6 shows the VSI 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 10.6 is not needed, but does allow data type checking.

Example 10.6. VSI 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 implicit interface subroutine
 twice.
 CALL MEG(XP)
 END
```

Example 10.7 shows the Compaq Fortran 77 program called by the VSI Fortran program (file ARRAY_F77.FOR).

Example 10.7. Compaq Fortran 77 Subroutine Called by a VSI Fortran Program

```
! Get array argument from F90. File: ARRAY_F77.FOR

 SUBROUTINE MEG(A)
 INTEGER A(3)
 PRINT *,A
 END
```

These files (shown in Example 10.6 and Example 10.7) might be compiled, linked, and run as follows:

```
$ FORTRAN ARRAY_TO_F77.F90
$ FORTRAN /OLD_F77 ARRAY_F77.FOR
$ LINK/EXECUTABLE=ARRAY_TO_F77 ARRAY_TO_F77, ARRAY_F77
$ RUN ARRAY_TO_F77
           1           2           3
           1           2           3
```

In Example 10.6, because array A is not defined as a pointer in the interface block, the VSI 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 10.7.

For More Information:

- On how VSI Fortran handles arguments and function return values, see Section 10.2.4.
- On explicit interfaces, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On compatibility between the VSI Fortran and Compaq Fortran 77 languages, see Appendix B.
- On use association, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On other aspects of the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On calling the Fortran 77 compiler with the /OLD_F77 compiler option, see Section 2.3.34.

10.9.2. Using Data Items in Common Blocks

To make global data available across VSI Fortran and Compaq Fortran 77 procedures, use common blocks.

Common blocks are supported by both VSI Fortran and Compaq Fortran 77, 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 VSI 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 VSI 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 FORTRAN command (VSI Fortran compiler).

- Specify the same alignment characteristics with the /ALIGNMENT qualifier when compiling both VSI Fortran procedures (FORTRAN command) and Compaq Fortran 77 procedures (FORTRAN/OLDF77 command).

  When compiling the source files with both the FORTRAN and FORTRAN/OLDF77 commands, consistently use the /ALIGN=COMMONS qualifier. 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, VSI 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 VSI Fortran declaration of COMPLEX (KIND=4) matches a Compaq Fortran 77 declaration of COMPLEX*8; COMPLEX (KIND=8) matches COMPLEX*16.

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 qualifiers that control the sizes of these declarations when compiling the procedures with multiple commands (same rules as Section 10.9.1).

10.9.3. I/O to the Same Unit Number

VSI Fortran and Compaq Fortran 77 share the same run-time system, so you can perform I/O to the same unit number by VSI Fortran and Compaq Fortran 77 procedures. For instance, a VSI Fortran main program can open the file, a Compaq Fortran 77 function can issue READ or WRITE statements to the same unit, and the VSI Fortran main program can close the file.

For More Information:

- On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On passing arguments, function return values, and the contents of registers on OpenVMS systems, see the *VSI OpenVMS Calling Standard*.
- On VSI Fortran intrinsic data types, see Chapter 8.
- On VSI Fortran I/O, see Chapter 6.

10.10. Calling Between VSI Fortran and VSI C

Before creating a mixed-language program that contains procedures written in VSI Fortran and C, you need to know how to:

- Compile and link the program
- Use equivalent data arguments passed between the two languages

10.10.1. Compiling and Linking Files

Use the FORTRAN command to compile VSI Fortran source files and CC to compile C source files. Link the object files using a LINK command.

For example, the following FORTRAN command compiles the VSI Fortran main program EX1.F90 and the called C function UPEN.C:

```
$ FORTRAN EX1.F90
$ CC UPEN.C
```

The following LINK command creates the executable program:

```
$ LINK EX1, UPEN
```

10.10.2. Procedures and External Names

When designing a program that will use VSI Fortran and C, be aware of the following general rules and available VSI Fortran capabilities:

- The OpenVMS Linker only allows one main program. Declare either the VSI Fortran or the C program, but not both, as the main program.

  In VSI Fortran, you can declare a main program:

  - With the PROGRAM and END PROGRAM statements
  - With an END statement

  To create a VSI 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.

- VSI Fortran
- External names in C and are usually converted to uppercase.

  Because both VSI Fortran and C make external names uppercase by default, external names should be uppercase unless requested otherwise.

  Consistently use the CC qualifier /NAMES and the FORTRAN qualifier /NAMES to control the way C and VSI Fortran treat external names (see Section 2.3.32).

- You can consider using the following VSI Fortran facility provided to simplify the VSI 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 10.4).

10.10.3. Invoking a C Function from VSI Fortran

You can use a function reference or a CALL statement to invoke a C function from a VSI Fortran main or subprogram.

If a value will be returned, use a function reference:

C Function Declaration	VSI 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	VSI Fortran Subroutine Invocation

`void calc( argument-list)`
`{`
`...`
`} ;`
	`EXTERNAL CALC`
	`...`
	`CALL CALC( argument-list)`

10.10.4. Invoking a VSI Fortran Function or Subroutine from C

A C main program or function can invoke a VSI Fortran function or subroutine by using a function prototype declaration and invocation.

If a value is returned, use a FUNCTION declaration:

VSI 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:

VSI Fortran Declaration	C Invocation

`SUBROUTINE CALC( argument-list)`
`...`
`END SUBROUTINE CALC`
	`extern void calc( argument-list)`
	
	`calc( argument-list);`
	`...`

10.10.5. Equivalent Data Types for Function Return Values

Both C and VSI Fortran pass most function return data by value, but equivalent data types must be used. The following table lists a sample of equivalent function declarations in VSI Fortran and C. See Table 10.8 for a complete list of data declarations.

C Function Declaration	VSI 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 VSI Fortran functions of type COMPLEX and DOUBLE COMPLEX, unless you pass a struct of two float (or double float) C values (see Section 10.10.9).

The floating-point format used in memory is determined by the /FLOAT qualifier for both the FORTRAN and CC commands. When floating-point data is passed as an argument or is globally available, the same floating-point format must be used in memory by both the C and VSI Fortran parts of your program.

The VSI 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.

For More Information:

- On using the CC command, see the *HP C User's Guide for OpenVMS*.

- On using the FORTRAN command, see Chapter 2.
- On VSI Fortran intrinsic data types, see Chapter 8.
- On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the VSI C language, see the *VSI C Language Reference Manual*.

10.10.6. Argument Association and Equivalent Data Types

VSI Fortran follows the argument-passing rules described in Section 10.2.4. These rules include:

- Passing arguments by reference (address)
- Receiving arguments by reference (address)
- Passing character data using a character descriptor (address and length)

10.10.6.1. VSI Fortran Intrinsic Data Types

VSI Fortran lets you specify the lengths of its intrinsic numeric data types with the following:

- The kind parameter, such as REAL (KIND=4), which is sometimes abbreviated as REAL(4). Intrinsic integer and logical kinds are 1, 2, 4, and 8. Intrinsic real and complex kinds are 4 (single-precision) and 8 (double-precision).
- A default-length name without a kind parameter, such as REAL or INTEGER. Certain FORTRAN command qualifiers can change the default kind, as described in Section 2.3.26 (for INTEGER and LOGICAL declarations), Section 2.3.37 (for REAL and COMPLEX declarations), and Section 2.3.17 (for DOUBLE PRECISION declarations).
- The VSI 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 FORTRAN command qualifier):

```
INTEGER (KIND=4) :: A1
INTEGER (4)      :: A2
INTEGER          :: A3
INTEGER*4        :: A4
```

Character data in VSI Fortran is passed and received by character descriptor. Dummy character arguments can use assumed-length for accepting character data of varying length.

For More Information:

On VSI Fortran intrinsic data types, see Chapter 8.

10.10.6.2. Equivalent VSI Fortran and C Data Types

The calling routine must pass the same number of arguments expected by the called routine. Also, 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 VSI Fortran typically passes argument data by reference.

You must determine the appropriate data types in each language that are compatible. When you call a C routine from a VSI Fortran main program, certain Fortran cDEC$ ATTRIBUTES directives may be useful to change the default passing mechanism (such as cDEC$ ATTRIBUTES C) as discussed in Section 10.4.2.

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 VSI Fortran subprogram, all arguments must be passed by reference because this is what the VSI 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 10.8.

Table 10.8. VSI Fortran and C Data Types

VSI Fortran Data Declaration	C Data Declaration
integer (kind=2) x	short int x;
integer (kind=4) x	int x;
integer (kind=8) x	__int64 x;
logical x	unsigned x;
real x	float x;
double precision x	double x;
real (kind=16) x	long double x;[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;[2]
character(len=5)	char x[5]

[1]VSI C interprets this as either a 128-bit IEEE X_FLOAT or a 64-bit floating-point number, depending on the value specified in the CC /L_DOUBLE-SIZE qualifier. The default is /L_DOUBLE-SIZE=128.

[2]The equivalent C declaration is long double (may not support X_floating).

Be aware of the various sizes supported by VSI Fortran for integer, logical, and real variables (see Chapter 8), and use the size consistent with that used in the C routine.

VSI 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.

The floating-point format used in memory is determined by the /FLOAT qualifier for both the FORTRAN and CC commands. When floating-point data is passed as an argument or is globally available, the same floating-point format must be used in memory both by the C and VSI Fortran parts of your program.

Any character string passed by VSI Fortran is *not* automatically null-terminated. To null-terminate a string from VSI Fortran, use the CHAR intrinsic function (described in the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

10.10.7. Example of Passing Integer Data to C Functions

Example 10.8 shows C code that declares the two functions HLN and MGN. These functions display the arguments received. The function HLN expects the argument by value, whereas MGN expects the argument by reference (address).

Example 10.8. C Functions Called by a VSI 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 10.9 shows the VSI Fortran (main program) code that calls the two C functions HLN and MGN.

Example 10.9. 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
  PRINT *,"101==",I
  I = 99
  CALL MGN(I)             ! pass by reference
  PRINT *,"101==",I
  END
```

The files (shown in Example 10.8 and Example 10.9) might be compiled, linked, and run as follows:

```
$ FORTRAN PASS_INT_TO_CFUNCS.F90
$ CC PASS_INT_TO_C.C
$ LINK/EXECUTABLE=PASS_INT PASS_INT_TO_CFUNCS, PASS_INT_TO_C
$ RUN PASS_INT
99==99
99==          99
99==99
101==         101
99==99
101==         101
```

10.10.8. Example of Passing Complex Data to C Functions

Example 10.10 shows VSI 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 10.10. 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 VSI 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", c1, c2);
}
```

The main routine (shown in Example 10.10) might be compiled and linked to the object file created by the compilation of the VSI Fortran subroutine and then run as follows:

```
$ FORTRAN CV_MAIN.F90
$ FORTRAN CV_SUB.F90
$ LINK/EXECUTABLE=CV.EXE CV_MAIN.OBJ, CV_SUB.OBJ
$ RUN CV
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 CV_SUB.C
$ LINK/EXECUTABLE=CV2.EXE CV_MAIN.OBJ CV_SUB.OBJ
$ RUN CV2
1.000000,0.000000
```

10.10.9. Handling User-Defined Structures

User-defined derived types in VSI 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 qualifier (or pragma) to alter that alignment.

  Derived-type data in VSI Fortran is naturally aligned (the compiler adds needed padding bytes) unless you specify the /ALIGNMENT= RECORDS=PACKED (or equivalent) qualifier (see Section 2.3.3).

- All elements of the structures are in the same order.

  VSI 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 10.10.6.

- The structure must be passed by reference (address).

10.10.10. Handling Scalar Pointer Data

When VSI 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, VSI Fortran dereferences the VSI 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, VSI Fortran passes the VSI Fortran pointer by reference.

When passing scalar numeric data without the pointer attribute, VSI 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 VSI Fortran subprogram, the C program can use pointers to pass numeric data by reference.

Example 10.11 shows a VSI 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), the pointer is passed. Without an explicit interface IFUNC2, the target data is passed.

Example 10.11. Calling C Functions and Passing Pointer Arguments

```
! Pass scalar pointer argument to C. File: SCALAR_POINTER_FUNC.F90

INTERFACE
  FUNCTION IFUNC1(A)
  INTEGER, POINTER :: A
  INTEGER IFUNC1
  END FUNCTION
END INTERFACE

INTEGER, POINTER :: X
INTEGER, TARGET :: Y

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 "y" by reference. C expects "int *"
PRINT *,Y
END
```

Example 10.12 shows the C function declarations that receive the VSI Fortran pointer or target arguments from the example in Example 10.11.

Example 10.12. C Functions Receiving Pointer Arguments

```
/* C functions Fortran 90 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 10.11 and Example 10.12) might be compiled, linked, and run as follows:

```
$ CC  SCALAR_POINTER.C
$ FORTRAN SCALAR_POINTER_FUNC.F90
$ LINK/EXECUTABLE=POINTER  SCALAR_POINTER, SCALAR_POINTER_FUNC
$ RUN POINTER
a=88
         100
```

```
a=99
         101
          77
```

10.10.11. Handling Arrays

There are two major differences between the way the C and VSI Fortran languages handle arrays:

- VSI 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 VSI 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 VSI 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 VSI 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 VSI Fortran array as 0.

VSI Fortran orders arrays in column-major order. The following VSI 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 VSI 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 VSI Fortran array data:

- Consider using a 0 (zero) as the lowest bounds in the VSI Fortran array declaration.

  You may need to specify the VSI Fortran declaration with a lowest 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 VSI 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 VSI 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 10.2.7).

For information about performance when using multidimensional arrays, see Section 5.4.

Example 10.13 shows a C function declaration for function EXPSHAPE, which prints the passed explicit-shape array.

Example 10.13. C Function That Receives an Explicit-Shape Array

```
/* Get explicit-shape arrays from Fortran. File: EXPARRAY_FUNC.C */

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 10.14 shows a VSI Fortran program that calls the C function EXPSHAPE (shown in Example 10.13).

Example 10.14. VSI Fortran Program That Passes an Explicit-Shape Array

```
! Pass an explicit-shape array from Fortran to C. File: EXPARRAY.F90

INTEGER :: X(2,3)
X = RESHAPE( (/(I,I=1,6)/), (/2,3/) )

CALL EXPSHAPE(X)
END
```

The files (shown in Example 10.13 and Example 10.14) might be compiled, linked, and run as follows:

```
$ FORTRAN EXPARRAY.F90
$ CC EXPARRAY_FUNC.C
$ LINK/EXECUTABLE=EXPARRAY EXPARRAY, EXPARRAY_FUNC
$ RUN 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 VSI Fortran, see Section 10.2.5.

10.10.12. Handling Common Blocks of Data

The following notes apply to handling common blocks of data between VSI Fortran and C:

- In VSI Fortran, you declare each common block with the COMMON statement. In C, you can use any global variable defined as a struct.
- Data types must match unless you desire implicit equivalencing. If so, you must adhere to the alignment restrictions for VSI 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 $BLANK.

- You should specify the same alignment characteristics in C and VSI Fortran. To specify the alignment of common block data items, specify the /ALIGN=COMMONS=NATURAL or /ALIGN=COMMONS=STANDARD qualifiers when compiling VSI Fortran procedures using the FORTRAN command or specify data declarations carefully (see Section 5.3).

The following examples show how C and VSI Fortran code can access common blocks of data. The C code declares a global structure, calls the f_calc VSI 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 VSI 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 SUBROUTINE F_CALC
```

The C program then prints the structure member values 356 and 5.9 set by the VSI Fortran function.

Chapter 11. Using OpenVMS Record Management Services

This chapter describes:

- Section 11.1: Overview of OpenVMS Record Management Services
- Section 11.2: RMS Data Structures
- Section 11.3: RMS Services
- Section 11.4: User-Written Open Procedures
- Section 11.5: Example of Block Mode I/O

11.1. Overview of OpenVMS Record Management Services

You can call OpenVMS Record Management Services (RMS) directly from VSI Fortran programs. RMS is used by all utilities and languages for their I/O processing, allowing files to be accessed efficiently, flexibly, with device independence, and taking full advantage of the capabilities of the OpenVMS operating system.

You need to know the basic concepts concerning files on OpenVMS systems and system-service calling conventions before reading this chapter. Basic file concepts are covered in the *Guide to OpenVMS File Applications*, and system-service calling conventions are covered in Chapter 10.

You should also have access to the *VSI OpenVMS Record Management Services Reference Manual*. Although not written specifically for VSI Fortran programmers, it is the definitive reference source for all information on the use of RMS.

This chapter will help you understand the material in the *VSI OpenVMS Record Management Services Reference Manual* in terms of Fortran concepts and usage. You should also be able to more fully understand the material in the *Guide to OpenVMS File Applications*, which covers more areas of RMS in greater detail than this chapter.

The easiest way to call RMS services directly from VSI Fortran is to use a USEROPEN routine, which is a subprogram that you specify in an OPEN statement. The VSI Fortran Run-Time Library (RTL) I/O support routines call the USEROPEN routine in place of the RMS services at the time a file is first opened for I/O.

The VSI Fortran RTL sets up the RMS data structures on your behalf with initial field values that are based on parameters specified in your OPEN statement. This initialization usually eliminates most of the code needed to set up the proper input to RMS Services. If you specify the USEROPEN keyword in the OPEN statement, control transfers to the specified USEROPEN routine that can further change RMS data structures and then call the appropriate RMS services, including SYS$OPEN (or SYS$CREATE) and SYS$CONNECT.

When you use USEROPEN routines, you can take advantage of the power of RMS without most of the declarations and initialization code normally required. Section 11.4 describes how to use USEROPEN routines and gives examples. You should be familiar with the material in Section 11.2 before reading Section 11.4.

11.2. RMS Data Structures

RMS system services have so many options and capabilities that it is impractical to use anything other than several large data structures to provide their arguments. You should become familiar with all of the RMS data structures before using RMS system services.

The RMS data structures are:

- File Access Block (FAB): used to describe files in general.
- Record Access Block (RAB): used to describe the records in files.
- Name Block (NAM): used to give supplementary information about the name of files beyond that provided with the FAB.
- Extended Attributes Blocks (XABs): a family of related blocks that are linked to the FAB or RAB to communicate to RMS any file attributes beyond those expressed in the FAB.

The RMS data structures are used both to pass arguments to RMS services and to return information from RMS services to your program. In particular, an auxiliary structure, such as a NAM or XAB block, is commonly used explicitly to obtain information optionally returned from RMS services.

The *VSI OpenVMS Record Management Services Reference Manual* describes how each of these data structures is used in calls to RMS services. In this section, a brief overview of each block is given, describing its purpose and how it is manipulated in VSI Fortran programs.

In general, there are six steps to using the RMS control blocks in calls to RMS system services:

1. Declare the structure of the blocks and the symbolic parameters used in them by including the appropriate definition library modules from the Fortran default library FORSYSDEF.TLB.
2. Declare the memory allocation for the blocks that you need with a RECORD statement.
3. Declare the system service names by including the library module $SYSSRVNAM from FORSYSDEF.TLB.
4. Initialize the values of fields needed by the service you are calling. The structure definitions provided for these blocks in the FORSYSDEF library modules provide only the field names and offsets needed to reference the RMS data structures. You must assign all of the field values explicitly in your VSI Fortran program.

   Two fields of each control block are mandatory; they must be filled in with the correct values before they are used in any service call. These are the block id (BID, or COD in the case of XABs) and the block length (BLN). These are checked by all RMS services to ensure that their input blocks have proper form.

   These fields must be assigned explicitly in your VSI Fortran programs, unless you are using the control blocks provided by the Fortran RTL I/O routines, which initialize all control block fields. See Table 11.1 for a list of the control field values provided by the Fortran RTL I/O routines.
5. Invoke the system service as a function reference, giving the control blocks as arguments according to the specifications in the RMS reference manual.
6. Check the return status to ensure that the service has completed successfully.

Steps 1-4 are described for each type of control block in Section 11.2.2 to Section 11.2.5. See Section 11.3 for descriptions of steps 5 and 6.

11.2.1. Using FORSYSDEF Library Modules to Manipulate RMS Data Structures

The definition library FORSYSDEF.TLB contains the required Fortran declarations for all of the field offsets and symbolic values of field contents described in the *VSI OpenVMS Record Management Services Reference Manual*. The appropriate INCLUDE statement needed to access these declarations for each structure is described wherever appropriate in the text that follows.

In general, you need to supply one or more RECORD statements to allocate the memory for the structures that you need. See the *VSI OpenVMS Record Management Services Reference Manual* for a description of the naming conventions used in RMS service calls. Only the convention for the PARAMETER declarations is described here.

The FORSYSDEF library modules contain several different kinds of PARAMETER declarations. The declarations are distinguished from each other by the letter following the dollar sign ($) in their symbolic names. Each is useful in manipulating field values, but the intended use of the different kinds of PARAMETER declarations is as follows:

- Declarations that define only symbolic field values are identified by the presence of a “C_” immediately after the block prefix in their names. For example, the RAB$B_RAC field has three symbolic values, one each for sequential, keyed, and RFA access modes. The symbolic names for these values are RABC_SEQ, RABC_KEY, and RAB$C_RFA. You use these symbolic field values in simple assignment statements. For example:

  ```
  INCLUDE '($RABDEF)'
  RECORD /RABDEF/ MYRAB
  . . .
  MYRAB.RAB$B_RAC = RAB$C_SEQ
  . . .
  ```

- Declarations that use mask values instead of explicit values to define bit offsets are identified by the presence of “M_” immediately after the block prefix in their names. For example, the FAB$L_FOP field is an INTEGER*4 field with the individual bits treated as flags. Each flag has a mask value for specifying a particular file processing option. For instance, the MXV bit specifies that RMS should maximize the version number of the file when it is created. The mask value associated with this bit has the name FAB$M_MXV.

  In order to use these parameters, you must use .AND. and .OR. to turn off and on specific bits in the field without changing the other bits. For example, to set the MXV flag in the FOP field, you would use the following program segment:

  ```
  INCLUDE '($FABDEF)'
  RECORD /FABDEF/ MYFAB
  . . .
  MYFAB.FAB$L_FOP = MYFAB.FAB$L_FOP .OR. FAB$M_MXV
  ```

- Two types of declarations that define symbolic field values are used to define flag fields within a larger named field. These are identified by the presence of “S_” or “V_” immediately after the block prefix in their names.

  The “S_” form of the name defines the size of that flag field (usually the value 1, for single bit flag fields), and the “V_” form defines the bit offset from the beginning of the larger field. These forms can be used with the symbolic bit manipulation functions to set or clear the fields without destroying the other flags. To perform the same operation as the previous example, but using the “V_” and “S_” flags, specify the following:

```
INCLUDE '($FABDEF)'
RECORD /FABDEF/ MYFAB
. . .
MYFAB.FAB$L_FOP = IBSET(MYFAB.FAB$L_FOP,FAB$V_MXV)
. . .
```

For most of the FAB, RAB, NAM, and XAB fields that are not supplied with symbolic values, you will need to supply sizes or pointers. For sizes, you can use ordinary numeric constants or other numeric scalar quantities. To set the maximum record number into the FAB$L_MRN field, you could use the following statement:

```
MYFAB.FAB$L_MRN = 5000
```

To supply the required pointers, usually from one block to another, you must use the %LOC built-in function to retrieve addresses. To fill in the FAB$L_NAM field in a FAB block with the address of the NAM block that you want to use, you can use the following program fragment:

```
INCLUDE '($FABDEF)'
INCLUDE '($NAMDEF)'
. . .
RECORD /FABDEF/ MYFAB, /NAMDEF/ MYNAM
. . .
MYFAB.FAB$L_NAM = %LOC(MYNAM)
```

11.2.2. File Access Block (FAB)

The File Access Block (FAB) is used for calling the following services:

SYS$CLOSE	SYS$OPEN
SYS$CREATE	SYS$PARSE
SYS$DISPLAY	SYS$REMOVE
SYS$ENTER	SYS$RENAME
SYS$ERASE	SYS$SEARCH
SYS$EXTEND	

The purpose of the FAB is to describe the file being manipulated by these services. In addition to the fields that describe the file directly, there are pointers in the FAB structure to auxiliary blocks used for more detailed information about the file. These auxiliary blocks are the NAM block and one or more of the XAB blocks.

To declare the structure and parameter values for using FAB blocks, include the $FABDEF library module from FORSYSDEF. For example:

```
INCLUDE '($FABDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the FAB is described at length in the *VSI OpenVMS Record Management Services Reference Manual*.

If you are using a USEROPEN procedure, the actual allocation of the FAB is performed by the Fortran Run-Time Library I/O support routines, and you only need to declare the first argument to your USEROPEN routine to be a record with the FAB structure. For example:

Calling program:

```
. . .
EXTERNAL MYOPEN
```

```
. . .
OPEN (UNIT=8, . . . , USEROPEN=MYOPEN)
. . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
INCLUDE '($FABDEF)'
. . .
RECORD /FABDEF/ FABARG
. . .
```

Usually, you need to declare only one FAB block, but some situations you need to use two. For example, the SYS$RENAME service requires one FAB block to describe the old file name and another to describe the new file name. In any of these cases, you can declare whatever FAB blocks you need with a RECORD statement. For example:

```
INCLUDE '($FABDEF)'
. . .
RECORD /FABDEF/ OLDFAB, NEWFAB
```

If you use any of the above service calls without using a USEROPEN routine, you must initialize the required FAB fields in your program. The FAB fields required for each RMS service are listed in the descriptions of the individual services in the *VSI OpenVMS Record Management Services Reference Manual*. Most services also fill in output values in the FAB or one of its associated blocks. Descriptions of these output fields are also provided with the service descriptions.

In the example programs in the *VSI OpenVMS Record Management Services Reference Manual*, these initial field values are described as they would be used in MACRO programs, where the declaration macros allow initialization arguments. In each case where the MACRO example shows a field being initialized in a macro, you must have a corresponding initialization at run time in your program.

The *VSI OpenVMS Record Management Services Reference Manual* contains an example that shows the use of the ALQ parameter for specifying the initial allocation size of the file in blocks:

```
! Program that uses XABDAT and XABDAT_STORE
    .
    .
    .
MYFAB: $FAB ALQ=500, FOP=CBT, FAC=<PUT>, -
FNM=<DISK$:[PROGRAM]SAMPLE_FILE.DAT>, -
ORG=SEQ, RAT=CR, RFM=VAR, SHR=<NIL>, MRS=52, XAB=MYXDAT
    .
    .
    .
```

As described in the section on the XAB$L_ALQ field (in the same manual), this parameter sets the FAB field FAB$L_ALQ. To perform the same initialization in VSI Fortran, you must supply a value to the FAB$L_ALQ field using a run-time assignment statement. For example:

```
MYFAB.FAB$L_ALQ = 500
```

The FAB$B_BID and FAB$B_BLN fields must be filled in by your program prior to their use in an RMS service call, unless they have already been supplied by the VSI Fortran RTL I/O routines. You should always use the symbolic names for the values of these fields; for example:

```
INCLUDE '($FABDEF)'
. . .
RECORD /FABDEF/ MYFAB
```

```
. . .
MYFAB.FAB$B_BID = FAB$C_BID
MYFAB.FAB$B_BLN = FAB$C_BLN
. . .
STATUS = SYS$OPEN( . . . )
. . .
```

11.2.3. Record Access Block (RAB)

The Record Access Block (RAB) is used for calling the following services:

SYS$CONNECT	SYS$READ
SYS$DELETE	SYS$RELEASE
SYS$DISCONNECT	SYS$REWIND
SYS$FIND	SYS$SPACE
SYS$FLUSH	SYS$TRUNCATE
SYS$FREE	SYS$UPDATE
SYS$GET	SYS$WAIT
SYS$NXTVOL	SYS$WRITE
SYS$PUT	

The purpose of the RAB is to describe the record being manipulated by these services. The RAB contains a pointer to the FAB used to open the file being manipulated, making it unnecessary for these services to have a FAB in their argument lists. Also, a RAB can point to certain types of XABs.

Using the FOR$RAB Intrinsic Function

To declare the structure and parameter values for using RAB blocks, include the $RABDEF library module from FORSYSDEF.For example:

```
INCLUDE '($RABDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the RAB is described at length in the *VSI OpenVMS Record Management Services Reference Manual*.

If you are using a USEROPEN procedure, the actual allocation of the RAB is performed by the VSI Fortran Run-Time Library I/O support routines, and you only need to declare the second argument to your USEROPEN routine to be a record with the RAB structure. For example:

Calling program:

```
. . .
EXTERNAL MYOPEN
. . .
OPEN (UNIT=8, . . . , USEROPEN=MYOPEN)
. . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
. . .
INCLUDE '($RABDEF)'
. . .
RECORD /RABDEF/ RABARG
. . .
```

If you need to access the RAB used by the Fortran I/O system for one of the open files in your program, you can use the FOR$RAB intrinsic function (do not declare it as EXTERNAL). You can use FOR$RAB even if you did not use a USEROPEN routine to open the file. The FOR$RAB intrinsic function takes a single INTEGER*4 variable (or constant) argument, the unit number of the open file for which you want to obtain the RAB address. The function result is the address of the RAB for that unit.

If you use the FOR$RAB function in your program, you should declare it to be INTEGER*4 if you assign the result value to a variable. If you do not, your program will assume that it is a REAL function and will perform an improper conversion to INTEGER.

To use the result of the FOR$RAB call, you must pass it to a subprogram as an actual argument using the %VAL built-in function. This allows the subprogram to access it as an ordinary VSI Fortran record argument. For example, the main program for calling a subroutine to print the RAB fields could be coded as follows:

```
INTEGER (KIND=4) RABADR, FOR$RAB
. . .
OPEN(UNIT=14, FILE='TEST.DAT', STATUS='OLD')
. . .
RABADR = FOR$RAB(14)
. . .
CALL DUMPRAB(%VAL(RABADR))
. . .
```

If you need to access other control blocks in use by the RMS services for that unit, you can obtain their addresses using the link fields they contain. For example:

```
SUBROUTINE DUMPRAB(RAB)
. . .
INTEGER (KIND=4) FABADR
INCLUDE '($RABDEF)'
RECORD /RABDEF/ RAB
. . .
FABADR = RAB.RAB$L_FAB
. . .
CALL DUMPFAB(%VAL(FABADR))
. . .
```

In this example, the routine DUMPRAB obtains the address of the associated FAB by referencing the RAB$L_FAB field of the RAB. Other control blocks associated with the FAB, such as the NAM and XAB blocks, can be accessed using code similar to this example.

Usually, you need to declare only one RAB block. Sometimes, however, you may need to use more than one. For example, the multistream capability of RMS allows you to connect several RABs to a single FAB. This allows you to simultaneously access several records of a file, keeping a separate context for each record. You can declare whatever RAB blocks you need with a RECORD statement. For example:

```
INCLUDE '($RABDEF)'
. . .
RECORD /RABDEF/ RAB1, RABARRAY(10)
```

If you use any of the above service calls without using a USEROPEN routine, you must initialize the required RAB fields in your program. The RAB fields required for each RMS service are listed in the descriptions of individual services in the *VSI OpenVMS Record Management Services Reference Manual*. Most services also fill in output values in the RAB. Descriptions of these output fields are provided with the service descriptions.

In the example programs supplied in the *VSI OpenVMS Record Management Services Reference Manual*, these initial field values are described as they would be used in MACRO programs, where the declaration macros allow initialization arguments. Thus, in each case where the MACRO example shows a field being initialized in a declaration macro, you must have a corresponding initialization at run time in your program.

For example, the *VSI OpenVMS Record Management Services Reference Manual* contains an example that shows the use of the RAC parameter for specifying the record access mode to use:

```
  .
  .
  .

  SRC_FAB:
          $FAB    FAC=<GET>,-           ; File access for GET only
                  FOP=<SQO>,-           ; DAP file transfer mode
                  FNM=<SRC:>            ; Name of input file
  SRC_RAB:
          $RAB    FAB=SRC_FAB,-         ; Address of associated FAB
                  RAC=SEQ,-             ; Sequential record access
                  UBF=BUFFER,-          ; Buffer address
                  USZ=BUFFER_SIZE       ; Buffer size

  .
  .
  .
```

In this example, sequential access mode is used. As described in the section on the RAC field (in the same manual), this parameter sets the RAB$B_RAC field to the value RAB$C_SEQ. This means that to perform the same initialization in Fortran, you must supply RAC field values by a run-time assignment statement. For example:

```
MYRAB.RAB$B_RAC = RAB$C_SEQ
```

The RAB$B_BID and RAB$B_BLN fields must be filled in by your program prior to their use in an RMS service call, unless they have been supplied by the Fortran RTL I/O routines. You should always use the symbolic names for the values of these fields. For example:

```
INCLUDE '($RABDEF)'
. . .
RECORD /RABDEF/ MYRAB
. . .
MYRAB.RAB$B_BID = RAB$C_BID
MYRAB.RAB$B_BLN = RAB$C_BLN
. . .
STATUS = SYS$CONNECT(MYRAB)
. . .
```

11.2.4. Name Block (NAM)

The Name Block (NAM) can be used with the FAB in most FAB-related services in order to supply to or receive from RMS more detailed information about a file name. The NAM block is never given directly as an argument to an RMS service; to supply it you must link to it from the FAB. See Section 11.2.1 for an example of this.

To declare the structure and parameter values for using NAM blocks, include the $NAMDEF library module from FORSYSDEF. For example:

```
INCLUDE '($NAMDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the NAM is described in the *VSI OpenVMS Record Management Services Reference Manual*.

If you are using a USEROPEN procedure, the actual allocation of the NAM is performed by the VSI Fortran Run-Time Library I/O support routines. Because the NAM block is linked to the FAB, it is not explicitly given in the USEROPEN routine argument list.

To access the NAM, you need to call a subprogram, passing the pointer by value and accessing the NAM in the subprogram as a structure. For example:

Calling program:

```
. . .
EXTERNAL MYOPEN
. . .
OPEN (UNIT=8, . . . , USEROPEN=MYOPEN)
. . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
. . .
INCLUDE '($FABDEF)'
. . .
RECORD /FABDEF/ FABARG
. . .
CALL NAMACCESS(%VAL(FABARG.FAB$L_NAM))
. . .
```

NAM accessing routine:

```
SUBROUTINE NAMACCESS(NAMARG)
. . .
INCLUDE '($NAMDEF)'
. . .
RECORD /NAMDEF/ NAMARG
. . .
IF (NAMARG.NAM$B_ESL .GT. 132) GO TO 100
. . .
```

Usually, you only need to declare one NAM block. You can declare whatever NAM blocks you need with a RECORD statement. For example:

```
INCLUDE '($NAMDEF)'
. . .
RECORD /NAMDEF/ NAM1, NAM2
```

Most often, you use the NAM block to pass and receive information about the components of the file specification, such as the device, directory, file name, and file type. For this reason, most of the fields of the NAM block are CHARACTER strings and lengths. When using the NAM block, you should be familiar with the argument passing mechanisms for CHARACTER arguments described in Section 10.8.4.3.

Your program must fill in the NAM$B_BID and NAM$B_BLN fields prior to their use in an RMS service call, unless they have been supplied by the VSI Fortran RTL I/O routines. You should always use the symbolic names for the values of these fields. For example:

```
INCLUDE '($NAMDEF)'
. . .
RECORD /NAMDEF/ MYNAM
. . .
MYNAM.NAM$B_BID = NAM$C_BID
MYNAM.NAM$B_BLN = NAM$C_BLN
. . .
```

11.2.5. Extended Attributes Blocks (XABs)

Extended Attributes Blocks (XABs) are a family of related structures for passing and receiving additional information about files. There are different kinds of XABs:

- Allocation Control (XABALL)
- Date and Time (XABDAT)
- File Header Characteristics (XABFHC)
- Item List (XABITM)
- Journaling (XABJNL)
- Key Definition (XABKEY)
- Protection (XABPRO)
- Recovery Unit (XABRU)
- Revision Date and Time (XABRDT)
- Summary (XABSUM)
- Terminal (XABTRM)

The XABs are described in the *VSI OpenVMS Record Management Services Reference Manual*. XABs are generally smaller and simpler than the FAB, RAB, and NAM blocks because each describes information about a single aspect of the file. You do not have to use all of them; for any given call to an RMS service routine, use only those that are required.

Often the XAB fields override the corresponding fields in the FAB. For example, the allocation XAB describes the file's block allocation in more detail than the FAB$L_ALQ field can. For this reason, XAB$L_ALQ (the allocation field in the XABALL structure) always overrides the FAB$L_ALQ value.

The XABs used for any given RMS service call are connected to the FAB in a linked list. The head of the list is the FAB$L_XAB field in the FAB. This field contains the address of the first XAB to be used. Each successive XAB in the list links to the next using the XAB$L_NXT field. This field contains the address of the next XAB in the list. The order of the XABs in the list does not matter, but each kind of XAB must not appear more than once in the list.

The only kind of XAB that can be connected to a RAB instead of a FAB is the terminal XAB. It is linked to the RAB with the RAB$L_XAB field. This is needed because the terminal control information is dynamic and potentially changes with each record operation performed.

To declare the structure and parameter values for using the different XAB blocks, include the appropriate XAB definition library module from FORSYSDEF. (The names of the XAB definition library modules are listed previously in this section.) Also, because the XABs are a family of related control blocks, you need to include the $XABDEF library module from FORSYSDEF.TLB in order to

declare the fields common to all XABs. For example, to declare the fields used in the Date and Time XAB, use the following declarations:

```
INCLUDE '($XABDATDEF)'
INCLUDE '($XABDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. All of the fields in the XABs are described in the *VSI OpenVMS Record Management Services Reference Manual*.

If you are using a USEROPEN procedure, the actual allocation of the XABs used by the open operation is performed by the VSI Fortran Run-Time Library I/O support routines. Because the XAB blocks are linked to the FAB, the XAB block addresses are not explicitly given in the USEROPEN routine argument list.

To access the XABs, you need to call a subprogram and pass a pointer to it using the %VAL built-in function or a pointer argument. For an example of this method, see Section 11.2.3.

To allocate space for an XAB block in your program, you need to declare it with a RECORD statement. For example:

```
INCLUDE '($XABDATDEF)'
INCLUDE '($XABPRODEF)'
. . .
RECORD /XABDATDEF/ MYXABDAT, /XABPRODEF/ MYXABPRO
. . .
```

For each XAB that you declare in your program, you must supply the correct COD and BLN fields explicitly. These field offsets are common to all XABs and are contained in the $XABDEF library module in FORSYSDEF.TLB. The block id and length are unique for each kind of XAB and the symbolic values for them are contained in the separate XAB declaration library modules in FORSYSDEF.TLB. For example, to properly initialize a Date and Time XAB, you could use the following code segment:

```
INCLUDE '($XABDEF)'
INCLUDE '($XABDATDEF)'
RECORD /XABDEF/ MYXABDAT
. . .
MYXABDAT.XAB$B_COD = XAB$C_DAT
MYXABDAT.XAB$B_BLN = XAB$C_DATLEN
. . .
```

11.3. RMS Services

In general, you need to do the same things when calling an RMS service that you need to do when calling any OpenVMS service: declare the name, pass arguments, and check status values. However, RMS services have some additional conventions and ease-of-use features that you should be aware of.

For More Information:

- On calling OpenVMS system services, see Section 10.8.
- On each RMS service, see the *VSI OpenVMS Record Management Services Reference Manual*.

11.3.1. Declaring RMS System Service Names

As with the other system services, you should use the $SYSSRVNAM library module in FORSYSDEF to declare the names of all of the RMS services. For example:

```
INCLUDE '($SYSSRVNAM)'
```

This library module contains comments describing each OpenVMS system service, including all of the RMS services, and INTEGER*4 and EXTERNAL declarations for each. Including the library module allows you to use the names of the RMS services in your programs without further declaration.

11.3.2. Arguments to RMS Services

Most RMS services require three arguments:

- The first is the control block to be used, generally a RAB or FAB, and is mandatory.
- The second and third arguments are the addresses of routines to be called if the RMS service fails or succeeds, and these are optional.

Some RMS services take other arguments, but these services are rarely needed. You should always refer to the documentation for the specific service that you are calling for detailed information on its arguments.

Most RAB and FAB fields are ignored by most RMS services. The documentation of each service in the *VSI OpenVMS Record Management Services Reference Manual* describes which fields are input for that service and which are output, for each control block used. Services that take a FAB as an argument are called the File Control services. Services that take a RAB as an argument are called the Record Control services. Typically, you need to use both when doing RMS I/O in your program.

In general, fields that are not documented as required for input to a service are ignored and can be left uninitialized. The exceptions are the Block Id (BID or COD) and Block Length (BLN) fields; these must always be initialized. See the preceding sections about the respective blocks for examples of how to initialize these fields.

The output of many RMS services provides the values required for input to other RMS services. For this reason, you usually only need to initialize a few fields in each block to their nondefault values. This is especially true when using RMS blocks declared with the VSI Fortran RTL I/O routines as when using USEROPEN routines or the FOR$RAB function.

For More Information:

On passing arguments to system services, see Section 10.8.4.

11.3.3. Checking Status from RMS Services

You should always invoke RMS services as functions, rather than calling them as subroutines (see Section 10.8.2 for a general discussion of this topic). It is particularly important to check the status of RMS services because they usually do not cause an error when they fail. If the status is not checked immediately, the failure will go undetected until later in the program where it will be difficult to diagnose.

In most cases, you only need to check for success or failure by testing whether the returned status is true or false. Some services have alternate success-status possibilities. You should always check for these in cases where the program depends on the correct operation of the services.

The RMS services have a unique set of status return symbols not used by any of the other OpenVMS system services. You should always use these symbols whenever you check the individual status values returned. To obtain the declarations for these symbols, include the $RMSDEF library module from FORSYSDEF.TLB. For example:

```
INCLUDE '($RMSDEF)'
```

This statement includes in your program the declarations for all of the symbolic RMS return values.

The *VSI OpenVMS Record Management Services Reference Manual* documents the symbolic values, both success and failure, that can be returned from each of the services. Your program should always test each service-result status against these symbolic values and take appropriate action when a failure status is detected. You should always declare status variables as INTEGER*4 type in order to avoid unexpected numeric conversions. The recommended action depends on whether you are using RMS services in a USEROPEN routine.

The VSI Fortran RTL I/O routines that invoke your USEROPEN routine are expecting an RMS status as an output value. For this reason, you need to return the RMS status value as the function value for both failure and success conditions. For example:

```
INTEGER FUNCTION MYOPEN(FAB,RAB,LUN)
. . .
INCLUDE '($SYSSRVNAM)'  ! Declare RMS service names
. . .
MYOPEN = SYS$OPEN(FAB)
IF (.NOT. MYOPEN) RETURN
. . .
RETURN
END
```

In this case, if the SYS$OPEN service fails, it returns an error status into the function result variable MYOPEN. If the test of MYOPEN does not indicate success, the function returns the actual RMS status as its value. Then the RTL I/O routines will signal the appropriate Fortran error normally, as if a USEROPEN routine had not been used.

If the SYS$OPEN call succeeds, the program continues, and the RMS$_NORMAL success status will ultimately be returned to the Fortran RTL. This value will cause the OPEN statement that specifies MYOPEN to complete successfully.

If you are not using a USEROPEN routine, your program must indicate the error status directly, unless it is prepared to deal with it. Often, the easiest way to indicate an error and issue a helpful message is to signal the RMS condition directly with LIB$SIGNAL or LIB$STOP. For example:

```
. . .
INCLUDE '($SYSSRVNAM)'  ! Declare RMS service names
INTEGER (KIND=4) STATUS ! Declare a status variable
. . .
STATUS = SYS$GET(MYRAB)
IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
```

For More Information:

On the use of LIB$SIGNAL and LIB$STOP, see Chapter 14.

11.3.4. Opening a File

To perform input or output operations on a file, your program must first open the file and establish an active RMS I/O stream. To open a file, your program generally needs to call either the SYS$CREATE or SYS$OPEN services, followed by the SYS$CONNECT service. When your program uses an OPEN statement without a USEROPEN routine, the VSI Fortran RTL I/O routines call these RMS services.

You can use these options related to opening a file:

- Use the SYS$OPEN service to open an existing file. SYS$OPEN returns an error status if the file cannot be found.
- Use the SYS$CREATE service to intentionally create a new file.
- Use SYS$CREATE with the CIF bit in the FAB$L_FOP field to open a file that may or may not exist. The SYS$CREATE service will either open the file (if it exists) or create a new one (if it does not exist). You can use the SUP bit to force SYS$CREATE to create a new file even if one already exists.

The value of the FAB$B_FAC field of the FAB indicates to RMS what record operations are to be done on the file being opened. If a record operation that was not indicated by the FAC field (such as a SYS$PUT) is attempted, the record service will not perform the operation and will return a failure status. This file protection feature prevents you from accidentally corrupting a file when you use the wrong RMS service.

The SYS$CONNECT service establishes an active I/O stream, using a RAB, to a file that has been previously opened by your program. RMS identifies all active I/O streams by a unique identifier, called the Internal Stream Identifier (ISI). This value is stored in the RAB$W_ISI field of the RAB for each active stream being processed.

This field must always be zero when calling SYS$CONNECT. The SYS$CONNECT service initializes this field, so that subsequent operations using that RAB can be uniquely identified. Under some circumstances, you can establish more than one simultaneously active I/O stream to the same file. See the *VSI OpenVMS Record Management Services Reference Manual* for more information on this topic.

11.3.5. Closing a File

To close a file, use the SYS$CLOSE service. This terminates all active I/O streams under way on that file and frees all RMS resources being used for processing that file. Use the SYS$DISCONNECT service if you want to end one active I/O stream, but continue processing the file using another stream. This service sets the RAB$W_ISI value to zero so that the RAB can be reused for another stream.

11.3.6. Writing Data

To write data to a file, use the SYS$PUT or SYS$WRITE service. Your program must set the PUT bit in the FAB$B_FAC field when the file is opened; otherwise, the service attempting the write operation will fail.

Use the SYS$PUT service when you want to write data in **record mode** (the default). In record mode, RMS buffers data automatically and performs the actual output operation for a whole group of records at a time. This is the mode used for all VSI Fortran WRITE statements. Because most programs and utilities can read data written in record mode, this mode should be used when the data being written is to be read and processed by a general program or utility.

Use the SYS$WRITE service when you want to bypass the record management capabilities of RMS and write blocks of data directly to the device without additional buffering. This mode is called **block mode** I/O and is generally much faster and uses less CPU resources than record mode. For this reason, it is the preferred mode for writing large amounts of unformatted data to a device.

Block mode should only be used when the program that needs to read the data can also use block mode. If the program that is to read the data cannot use block mode, you must use some other means to guarantee that the data being written can be accessed. For instance, it is not generally possible to read data written with SYS$WRITE using ordinary VSI Fortran READ statements. Before

using SYS$WRITE, read the special restrictions on using block mode in the *VSI OpenVMS Record Management Services Reference Manual*, because SYS$WRITE (block mode) may have different device dependencies than SYS$PUT (record mode).

11.3.7. Reading Data

To read data from a file, use the SYS$GET or SYS$READ service. Your program must set the GET bit in the FAB$B_FAC field when the file is opened; otherwise, the service attempting the read operation will fail.

Use the SYS$GET service when you want to read data in record mode (the default). In this mode, RMS buffers data automatically and performs the actual input operation for a whole group of records at a time. This is the mode used for all VSI Fortran READ statements. This mode should be used whenever the program or utility that wrote the data used record mode, unless your reading program can buffer and deblock the data itself.

Use the SYS$READ service when you want to read data using block mode I/O (see Section 11.3.6. Using SYS$READ is the preferred mode for reading large amounts of unformatted data from a device, but it should only be used when the data was written by a utility or program that wrote the data in block mode.

If the file was written using record mode, RMS control information may be intermixed with the data, making it difficult to process. Before using SYS$READ, read the special restrictions on using block mode in the *VSI OpenVMS Record Management Services Reference Manual*, because SYS$READ (block mode) may have different device dependencies than SYS$GET (record mode).

11.3.8. Other Services

RMS provides many other file and record processing services beyond just opening, closing, reading, and writing. Other file processing services include:

- SYS$PARSE and SYS$SEARCH: process wildcard and incomplete file specifications and search for a sequence of files to be processed.
- SYS$DISPLAY: retrieves file attribute information.
- SYS$ENTER: inserts a file name into a directory file.
- SYS$ERASE: deletes a file and removes the directory entry used to specify it.
- SYS$EXTEND: increases the amount of disk space allocated to the file.
- SYS$REMOVE: removes directory entries for a file.
- SYS$RENAME: removes a directory entry for a file and inserts a new one in another directory.

Other record processing services include:

- SYS$FIND: positions the record stream at the desired record for later reading or writing.
- SYS$DELETE: deletes a record from the file.
- SYS$SPACE: skips over one or more blocks in block I/O mode.
- SYS$TRUNCATE: truncates a file after a given record.
- SYS$UPDATE: updates the value of an existing record.

For More Information:

On the RMS services, see the *VSI OpenVMS Record Management Services Reference Manual*.

11.4. User-Written Open Procedures

The USEROPEN keyword in an OPEN statement provides you with a way to access RMS facilities that are otherwise not available to VSI Fortran programs. It specifies a user-written external procedure (USEROPEN procedure) that controls the opening of a file. The USEROPEN keyword has the form:

```
USEROPEN = procedure-name
```

The procedure-name represents the symbolic name of a user-written open procedure. The procedure must be declared in an EXTERNAL statement, and should be a FUNCTION that returns an INTEGER*4 result.

When an OPEN statement—with or without the USEROPEN keyword—is executed, the Run-Time Library uses the OPEN statement keywords to establish the RMS File Access Block (FAB) and the Record Access Block (RAB), as well as its own internal data structures. If a USEROPEN keyword is included in the OPEN statement, the Run-Time Library then calls your USEROPEN procedure instead of opening the file according to its normal defaults. The procedure can then provide additional parameters to RMS and can obtain results from RMS.

The three arguments passed to a user-written open procedure by the Run-Time Library are:

1. Address of the FAB
2. Address of the RAB
3. Address of a longword integer containing the unit number

Using this information, your USEROPEN procedure can then perform the following operations:

- Modify the FAB and RAB (optional).
- Issue SYS$OPEN and SYS$CONNECT functions or SYS$CREATE and SYS$CONNECT functions when VSI Fortran I/O is to be performed (required).

  Your USEROPEN procedure should invoke the RMS SYS$OPEN routine if the file to be opened already exists (STATUS='OLD') or should call the RMS SYS$CREATE routine for any other file type (STATUS='NEW', 'UNKNOWN', or not specified). The status value specified in the OPEN statement is not represented in either the FAB or RAB.
- Check status indicators returned by RMS services (required).

  Your procedure should return immediately if an RMS service returns a failure status.
- Obtain information returned by RMS in the FAB and RAB by storing FAB and RAB values in program variables (optional).
- Return a success or failure status value to the Run-Time Library (required).

  The RMS services SYS$CREATE, SYS$OPEN, and SYS$CONNECT return status codes. Thus, it is not necessary to set a separate status value as the procedure output if execution of one of these macros is the final step in your procedure.

A USEROPEN routine can set FAB fields before the corresponding file is opened (or created). However, once the file is open (except if the FAB$V_UFO bit is set), the user should not alter any of the FAB fields.

A USEROPEN routine can likewise set RAB fields before the record stream has been connected (SYS$CONNECT). However, once the file is connected to the record stream, the user should not alter any of the RAB fields.

Once a FAB or RAB is used by the Fortran RTL, it should be treated as read-only by the user, because the Fortran RTL may need to set and alter those fields as needed to complete the user Fortran I/O statements. Any user modification of a FAB or RAB after its initial use may not have the intended effect for subsequent Fortran I/O statements.

For More Information:

On the FAB and RAB, see the *VSI OpenVMS Record Management Services Reference Manual*.

11.4.1. Examples of USEROPEN Routines

The following OPEN statement either creates a 1000-block contiguous file or returns an error. (The default VSI Fortran interpretation of the INITIALSIZE keyword is to allocate the file contiguously on a best-effort basis, but not to generate an error if the space is not completely contiguous).

```
  EXTERNAL CREATE_CONTIG
  OPEN (UNIT=10, FILE='DATA', STATUS='NEW', INITIALSIZE=1000, &
    USEROPEN=CREATE_CONTIG)
```

User-written open procedures are often coded in BLISS or MACRO; however, they can also be coded in VSI Fortran using VSI Fortran's record handling capability.

The following function creates a file after setting the RMS FOP bit (FAB$V_CTG) to specify contiguous allocation.

```
!       UOPEN1
!
! Program to demonstrate the use of a simple USEROPEN routine
!

  PROGRAM UOPEN1
  EXTERNAL CREATE_CONTIG

! OPEN the file specifying the USEROPEN routine

  OPEN (UNIT=10, FILE='DATA', STATUS='NEW', INITIALSIZE=1000, &
    USEROPEN=CREATE_CONTIG)

  STOP
  END PROGRAM UOPEN1

! CREATE_CONTIG

! Sample USEROPEN function to force RMS to allocate contiguous
! blocks for the initial creation of a file.

  INTEGER FUNCTION CREATE_CONTIG(FAB,RAB,LUN)
```

```
!       Required declarations

  INCLUDE '($FABDEF)'              ! FAB Structure
  INCLUDE '($RABDEF)'              ! RAB Structure
  INCLUDE '($SYSSRVNAM)'           ! System service name declarations
  RECORD /FABDEF/ FAB, /RABDEF/ RAB

! Clear the "Contiguous-best-try" bit, set the "Contiguous" bit

  FAB.FAB$L_FOP = FAB.FAB$L_FOP .AND. .NOT. FAB$M_CBT
  FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_CTG

!       Perform the create and connect, and return status

  CREATE_CONTIG = SYS$CREATE(FAB)
  IF (.NOT. CREATE_CONTIG) RETURN
  CREATE_CONTIG = SYS$CONNECT(RAB)
  RETURN
  END FUNCTION CREATE_CONTIG
```

11.4.2. RMS Control Structures

Use of the USEROPEN keyword has some restrictions. The Run-Time Library constructs the following RMS control structures before calling the USEROPEN procedure:

FAB	File Access Blck
RAB	Record Access Block
NAM	Name Block
XAB	Extended Attributes Blocks
ESA	Expanded String Area
RSA	Resultant String Area

A USEROPEN procedure should not alter the allocation of these structures, although it can modify the contents of many of the fields. Your procedure can also add additional XAB control blocks by linking them anywhere into the XAB chain. You must exercise caution when changing fields that have been set as a result of VSI Fortran keywords, because the Run-Time Library may not be aware of the changes. For example, do not attempt to change the record size in your USEROPEN procedure; instead, use the VSI Fortran keyword RECL. Always use an OPEN statement keyword if one is available.

Although the FAB, RAB, and NAM blocks remain defined during the time that the unit is opened, the XAB blocks are present only until the file has been successfully opened. The locations of the ESA and RSA strings are changed after the file is opened, so your USEROPEN procedure should not store the addresses of the RMS control structures. Instead, have your program call FOR$RAB to obtain the address of the RAB once the file is opened and then access the other structures through the RAB.

Note

Future releases of the Run-Time Library may alter the use of some RMS fields. Therefore, you may have to alter your USEROPEN procedures accordingly.

Table 11.1 shows which FAB and RAB fields are either initialized before your USEROPEN procedure is called or examined upon return from your USEROPEN procedure. All fields are initialized in response

to OPEN statement keywords or default to zero. Fields labeled with a hyphen (-) are initialized to zero. Fields labeled with an asterisk (*) are returned by RMS.

Table 11.1. RMS Fields Available with USEROPEN

Field Name	Description	VSI Fortran OPEN Keyword and Value
FAB$B_ACMODES	File access modes	Contains FAB$V_CHAN_MODE and FAB$V_LNM_MODE.
FAB$L_ALQ	Allocation quantity	n if INITIALSIZE=n
FAB$B_BKS	Bucket size	(BLOCKSIZE + 511)/512
FAB$W_BLS	Block size	n if BLOCKSIZE=n
FAB$V_CHAN_MODE	Channel access mode protection (2-bit subfield in FAB$B_ACMODES)	0=usermode
FAB$L_CTX	Context	–
FAB$W_DEQ	Default file extension quantity	n if EXTENDSIZE=n
FAB$L_DEV	Device characteristics	*
FAB$L_DNA	Default file specification string address	UNIT= `nnn` Set to FOR `nnn`.DAT or FORREAD.DAT, FORACCEPT.DAT, FORTYPE.DAT, or FORPRINT.DAT or to default file specification string
FAB$B_DNS	Default file specification string size	Set to length of default file specification string
FAB$B_FAC	File access	READONLY Set to 0 if READONLY (RMS default), else set to FAB$M_GET + FAB$M_PUT + FAB$M_UPD + FAB$M_TRN + FAB$M_DEL
FAB$L_FNA	File specification string address	FILE=filename if FILE present, else set to FOR `nnn`, FOR$READ, FOR$ACCEPT, FOR$TYPE, FOR$PRINT, SYS$INPUT, or SYS$OUTPUT
FAB$B_FNS	File specification string size	Set to length of file specification string
FAB$L_FOP	File processing options	
FAB$V_ASY	Asynchronous operation	– (not used)
FAB$V_CBT	Contiguous best try	1 if INITIALSIZE=n
FAB$V_CIF	Create if nonexistent	1 if READONLY not specified and STATUS= UNKNOWN) or STATUS omitted
FAB$V_CTG	Contiguous allocation	–
FAB$V_DFW	Deferred write	1

Field Name	Description	VSI Fortran OPEN Keyword and Value
FAB$V_DLT	Delete on close service	Set at Fortran close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_KFO	Known file open	–
FAB$V_MXV	Maximize version number	–
FAB$V_NAM	Name block inputs	–
FAB$V_NEF	Not positioned at end of file	1 unless ACCESS= APPEND
FAB$V_NFS	Not file structured	–
FAB$V_OFP	Output file parse	–
FAB$V_POS	Current position (after closed file)	–
FAB$V_PPF	Process permanent file	–
FAB$V_RCK	Read check	–
FAB$V_RWC	Rewind on close service	–
FAB$V_RWO	Rewind on open service	–
FAB$V_SCF	Submit command (when closed)	Set at Fortran close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_SPL	Spool to printer	Set at Fortran close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_SQO	Sequential only	1 if a network file and ACCESS= SEQUENTIAL or APPEND, else 0
FAB$V_SUP	Supersede	–
FAB$V_SYNCSTS	Immediate asynchronous completion	* (not used)
FAB$V_TEF	Truncate at end-of-file	–
FAB$V_TMD	Temporary, marked for delete	1 if STATUS= SCRATCH, else 0
FAB$V_TMP	Temporary (file with no directory entry)	–
FAB$V_UFO	User file open or create file only	–
FAB$V_WCK	Write check	–
FAB$B_FSZ	Fixed control area size	–
FAB$W_IFI	Internal file identifier	*
FAB$W_GBC	Global buffer count	–
FAB$B_JOURNAL	Journal flags status	–
FAB$V_LNM_MODE	Logical name translation access mode (subfield in FAB$B_ACMODES)	–

Field Name	Description	VSI Fortran OPEN Keyword and Value
FAB$L_MRN	Maximum record number	n if MAXREC=n
FAB$W_MRS	Maximum record size	n if RECORDTYPE= FIXED or ORGANIZATION= RELATIVE or = INDEXED, else 0
FAB$L_NAM	Name block address	Set to address of name block; both the expanded and resultant string areas are set up, but the related filename string is not
FAB$B_ORG	File organization	FAB$C_IDX if ORGANIZATION= INDEXED FAB$C_REL if ORGANIZATION= RELATIVE FAB$C_SEQ if ORGANIZATION= SEQUENTIAL or omitted
FAB$B_RAT	Record attributes	
FAB$V_FTN	Fortran carriage control	1 if CARRIAGECONTROL= FORTRAN or not specified
FAB$V_CR	Print LF and CR	1 if CARRIAGECONTROL= LIST
FAB$V_BLK	Do not cross block boundaries	– (1 if NOSPANBLOCKS)
FAB$B_RFM	Record format	FAB$C_FIX if RECORDTYPE= FIXED FAB$C_VAR if RECORDTYPE= VARIABLE FAB$C_VAR if RECORDTYPE= SEGMENTED FAB$C_STM if RECORDTYPE= STREAMF AB$C_STMCR if RECORDTYPE= STREAM_CR FAB$C_STMLF if RECORDTYPE= STREAM_LF
FAB$B_RTV	Retrieval window size	–
FAB$L_SDC	Spooling device characteristics	*
FAB$B_SHR	File sharing	
FAB$V_SHRPUT	Allow other PUTs	1 if SHARED
FAB$V_SHRGET	Allow other GETs	1 if SHARED
FAB$V_SHRDEL	Allow other DELETEs	1 if SHARED
FAB$V_SHRUPD	Allow other UPDATEs	1 if SHARED
FAB$V_NIL	Allow no other operations	–
FAB$V_UPI	User-provided interlock	–
FAB$V_MSE	Multistream allowed	–

Field Name	Description	VSI Fortran OPEN Keyword and Value
FAB$L_STS	Completion status code	*
FAB$L_STV	Status value	*
FAB$L_XAB	Extended attribute block address	The XAB chain always has a File Header Characteristics (FHC) extended attribute block in order to get longest record length (XAB$W_LRL). If the KEY=keyword is specified, key index definition blocks will also be present. VSI may add additional XABs in the future. Your USEROPEN procedure may insert XABs anywhere in the chain.
RAB$L_BKT	Bucket code	–
RAB$L_CTX	Context	–
RAB$L_FAB	FAB address	Set to address of FAB
RAB$W_ISI	Internal stream ID	*
RAB$L_KBF	Key buffer address	Set to address of longword containing logical record number if ACCESS= DIRECT
RAB$B_KRF	Key of reference	0
RAB$B_KSZ	Key size	–
RAB$B_MBC	Multiblock count	If BLOCKSIZE=n, use (n + 511)/512
RAB$B_MBF	Multibuffer count	n if BUFFERCOUNT=n
RAB$L_PBF	Prompt buffer address	–
RAB$B_PSZ	Prompt buffer size	–
RAB$B_RAC	Record access mode	
RAB$C_KEY	If ACCESS= DIRECT or KEYED	
RAB$C_SEQ	If ACCESS= SEQUENTIAL or APPEND, or ACCESS omitted	
RAB$C_RFA	–	
RAB$L_RBF	Record buffer address	Set later
RAB$L_RFA	Record file address	Set later
RAB$L_RHB	Record header buffer	–
RAB$L_ROP	Record processing options	
RAB$V_ASY	Asynchronous	–
RAB$V_BIO	Block I/O	–
RAB$V_CCO	Cancel CTRL/O	–
RAB$V_CVT	Convert to uppercase	–
RAB$V_EOF	End-of-file	1 if ACCESS= APPEND
RAB$V_ETO	Extended terminal (XABTRM) operation	–

Field Name	Description	VSI Fortran OPEN Keyword and Value
RAB$V_FDL	Fast delete	–
RAB$V_KGE or RAB$V_EQNXT	Key greater than or equal to	–
RAB$V_KGT or RAB$V_NXT	Key greater than	–
RAB$V_LIM	Limit	–
RAB$V_LOA	Load buckets according to fill size	–
RAB$V_LOC	Locate mode	1
RAB$V_NLK	No lock	–
RAB$V_NXR	Nonexistent record	–
RAB$V_PMT	Prompt	–
RAB$V_PTA	Purge type-ahead	–
RAB$V_RAH	Read-ahead	1
RAB$V_REA	Lock record for read	–
RAB$V_RLK	Lock record for write	–
RAB$V_RNE	Read no echo	–
RAB$V_RNF	Read no filter	–
RAB$V_RRL	Read regardless of lock	–
RAB$V_SYNCSTS	Immediate asynchronous completion	* (not used)
RAB$V_TMO	Timeout	–
RAB$V_TPT	Truncate on PUT	1
RAB$V_UIF	Update if	1 if ACCESS= DIRECT
RAB$V_ULK	Manual unlocking	–
RAB$V_WAT	Wait for locked record	–
RAB$V_WBH	Write-behind	1
RAB$W_RSZ	Record size	Set later
RAB$L_STS	Completion status code	*
RAB$L_STV	Status value	*
RAB$B_TMO	Timeout period	–
RAB$L_UBF	User record area address	Set later
RAB$W_USZ	User record area size	Set later
RAB$L_XAB	Extended attribute block address	The XAB chain allows you to set or obtain additional information about an I/O operation.

RMS does not allow multiple instances of the same type XAB. To be compatible with future releases of the Run-Time Library, your procedure should scan the XAB chain for XABs of the type to be inserted. If one is found, it should be used instead.

11.5. Example of Block Mode I/O

The following example shows a complete application that calls the RMS block I/O services SYS$WRITE and SYS$READ directly from VSI Fortran. A complete program called BIO.F90 writes out an array of REAL*8 values to a file using SYS$WRITE, closes the file, and then reads the data back in using SYS$READ operations with a different I/O transfer size. This program consists of five routines:

BIO	Main control program
BIOCREATE	USEROPEN routine to create the file
BIOREAD	USEROPEN routine to open the file for READ access
OUTPUT	Function that actually outputs the array
INPUT	Function that actually reads the array and checks it

11.5.1. Main Block Mode I/O Program—BIO

The following main program specifies the USEROPEN specifier in its OPEN statements.

```
 !  File: BIO.F90
 !
 !       Program to demonstrate the use of RMS Block I/O operations
 !       from HP Fortran

       PROGRAM BIO

 !  Declare the Useropen routines as external
    EXTERNAL BIOCREATE, BIOREAD
 !  Declare status variable, functions, and unit number

     LOGICAL (KIND=4)  STATUS, OUTPUT, INPUT
     INTEGER (KIND=4) IUN/1/

 !       Open the file                                          ❶
     OPEN(UNIT=IUN, FILE='BIODEMO.DAT', FORM='UNFORMATTED',
&      STATUS='NEW', RECL=128, BLOCKSIZE=512, ORGANIZATION='SEQUENTIAL',
&      IOSTAT=IOS,  ACCESS='SEQUENTIAL', RECORDTYPE='FIXED',
&      USEROPEN=BIOCREATE, INITIALSIZE=100)
    IF (IOS .NE. 0) STOP 'Create failed'                       ❷

 !       Now perform the output
     STATUS = OUTPUT(%VAL(FOR$RAB(IUN)))                       ❸
     IF (.NOT. STATUS) STOP 'Output failed'                    ❷

 !       Close the file for output
     CLOSE (UNIT=IUN)

 !  Confirm output complete
     TYPE *, 'Output complete, file closed'

 !  Now open the file for input                                ❷
     OPEN(UNIT=IUN, FILE='BIODEMO.DAT', FORM='UNFORMATTED',  &
      STATUS='OLD', IOSTAT=IOS, USEROPEN=BIOREAD, DISP='DELETE')
     IF (IOS .NE. 0) STOP 'Open for read failed'               ❷

 !       Now read the file back
     STATUS = INPUT(%VAL(FOR$RAB(IUN)))                        ❸
     IF (.NOT. STATUS) STOP 'Input failed'                     ❷
```

```
!       Success, output that all is well
      STOP 'Correct completion of Block I/O demo'
      END PROGRAM BIO
```

❶ Most of the necessary OPEN options for the file are specified with OPEN statement parameters. This is recommended whenever an OPEN statement qualifier exists to perform the desired function because it allows the VSI Fortran RTL I/O processing routines to issue appropriate error messages when an RMS routine returns an error status.

Note the discrepancy between RECL and BLOCKSIZE in the first OPEN statement. Both keywords specify 512 bytes, but the number given for RECL is 128. This is because the unit implied in the RECL keyword is longwords for unformatted files.

When using Block I/O mode, the blocksize used in the I/O operations is determined by the routine that actually does the operation. The OUTPUT routine actually transfers *two* 512-byte blocks at a time; the INPUT routine actually transfers *four* 512-byte blocks at once (see Section 11.5.2).

In general, the larger the transfers, the more efficiently the I/O is performed. The maximum I/O transfer size allowed by RMS is 65535 bytes.

❷ The error processing in this example routine is very crude; the program simply stops with an indicator of where the problem occurred. In real programs, you should provide more extensive error processing and reporting functions.

❸ The intrinsic function FOR$RAB is used to supply the appropriate RAB address to the OUTPUT and INPUT routines. The %VAL function is used to transform the address returned by the FOR$RAB intrinsic function to the proper argument passing mechanism. This allows the dummy argument RAB in INPUT and OUTPUT to be addressed properly.

11.5.2. Block Mode I/O USEROPEN Functions—BIOCREATE and BIOREAD

The only condition required for block I/O is the setting of the BIO bit in the File Access field of the FAB, using the normal declarations needed to define the symbols properly. If you wish to perform both block and record I/O on the file without closing it, you need to set the BRO bit as well. For more information on mixing block and record mode I/O, see the *VSI OpenVMS Record Management Services Reference Manual*. Note that the only difference between BIOCREATE and BIOREAD is the use of SYS$CREATE and SYS$OPEN services, respectively.

```
! Procedure name: BIOCREATE

! USEROPEN routine to set the Block I/O bit and create the BLOCK I/O file.

  INTEGER FUNCTION BIOCREATE(FAB, RAB, LUN)
  INTEGER LUN

!       Declare the necessary interface names

  INCLUDE '($FABDEF)'
  INCLUDE '($RABDEF)'
  INCLUDE '($SYSSRVNAM)'

!       Declare the FAB and RAB blocks

  RECORD /FABDEF/ FAB, /RABDEF/ RAB

! Set the Block I/O bit in the FAC (GET and PUT bits set by RTL)
```

```
      FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_BIO

! Now do the Create and Connect

  BIOCREATE = SYS$CREATE(FAB)
  IF (.NOT. BIOCREATE) RETURN
  BIOCREATE = SYS$CONNECT(RAB)
  IF (.NOT. BIOCREATE) RETURN

! Nothing more to do at this point, just return

  RETURN
  END FUNCTION BIOCREATE


! Procedure name: BIOREAD

! USEROPEN routine to set the Block I/O bit and open the Block I/O demo
! file for reading

  INTEGER FUNCTION BIOREAD(FAB, RAB, LUN)
  INTEGER LUN

! Declare the necessary interface names

  INCLUDE '($FABDEF)'
  INCLUDE '($RABDEF)'
  INCLUDE '($SYSSRVNAM)'

! Declare the FAB and RAB blocks

      RECORD /FABDEF/ FAB, /RABDEF/ RAB

! Set the Block I/O bit in the FAC (GET and PUT bits set by RTL)

      FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_BIO

! Now do the Open and Connect

  BIOREAD = SYS$OPEN(FAB)
  IF (.NOT. BIOREAD) RETURN
  BIOREAD = SYS$CONNECT(RAB)
  IF (.NOT. BIOREAD) RETURN

!       Nothing more to do at this point, just return

  RETURN
  END FUNCTION BIOREAD
```

11.5.2.1. OUTPUT Routine

The following routine initializes the array A and performs the SYS$WRITE operations. Beyond the normal RTL initialization, only the RSZ and RBF fields in the RAB need to be initialized in order to perform the SYS$WRITE operations. The %LOC function is used to create the address value required in the RBF field.

One of the main reasons that block mode I/O is so efficient is that it avoids copy operations by using the data areas of the program directly for the output buffer. When writing to a disk device, the program must specify a value for RSZ that is a multiple of 512 or else the final block would be only partly filled.

```
! Procedure name: OUTPUT

! Function to output records in block I/O mode

  LOGICAL FUNCTION OUTPUT(RAB)

!       Declare RMS names

  INCLUDE '($RABDEF)'
  INCLUDE '($SYSSRVNAM)'

!       Declare the RAB

  RECORD /RABDEF/ RAB

! Declare the Array to output

  REAL(KIND=8) A(6400)

!       Declare the status variable

  INTEGER(KIND=4) STATUS

!       Initialize the array

  DO I=6400,1,-1
    A(I) = I
  ENDDO

! Now, output the array, two 512-byte (64 elements) blocks at a time

  OUTPUT = .FALSE.
  RAB.RAB$W_RSZ = 1024
  DO I=0,99,2

! For each block, set the buffer address to the proper array element

    RAB.RAB$L_RBF = %LOC(A(I*64+1))
    STATUS = SYS$WRITE(RAB)
    IF (.NOT. STATUS) RETURN
  ENDDO

!       Successful output completion

  OUTPUT = .TRUE.
  RETURN
  END FUNCTION OUTPUT
```

11.5.2.2. INPUT Routine

The following routine reads the array A from the file and verifies its values. The USZ and UBF fields of the RAB are the only fields that need to be initialized. The I/O transfer size is twice as large as the OUTPUT routine. This can be done because the OUTPUT routine writes an integral number of 512-

byte blocks to a disk device. This method cannot be used if the writing routine either specifies an RSZ that is not a multiple of 512 or attempts to write to a magnetic tape device.

```
! Procedure name: INPUT
!
! Function to input records in block I/O mode

  LOGICAL FUNCTION INPUT(RAB)

! Declare RMS names

  INCLUDE '($RABDEF)'
  INCLUDE '($SYSSRVNAM)'

! Declare the RAB

  RECORD /RABDEF/ RAB

! Declare the Array to output

  REAL(KIND=8) A(6400)

!       Declare the status variable

  INTEGER(KIND=4) STATUS

! Now, read the array, four 512-byte (64 elements) blocks at a time

  INPUT = .FALSE.
  RAB.RAB$W_USZ = 2048
  DO I=0,99,4

! For each block, set the buffer address to the proper array element

    RAB.RAB$L_UBF = %LOC(A(I*64+1))
    STATUS = SYS$READ(RAB)
    IF (.NOT. STATUS) RETURN
  ENDDO

!       Successful input completion if data is correct

  DO I=6400,1,-1
     IF (A(I) .NE. I) RETURN
  ENDDO

  INPUT = .TRUE.
  RETURN
  END FUNCTION INPUT
```

Chapter 12. Using Indexed Files

This chapter describes:

- Section 12.1: Overview of Indexed Files
- Section 12.2: Creating an Indexed File
- Section 12.3: Writing Records to an Indexed File
- Section 12.4: Reading Records from an Indexed File
- Section 12.5: Updating Records in an Indexed File
- Section 12.6: Deleting Records from an Indexed File
- Section 12.7: Current Record and Next Record Pointers
- Section 12.8: Exception Conditions When Using Indexed Files

12.1. Overview of Indexed Files

Sequential and direct access have traditionally been the only file access modes available to Fortran programs. To overcome some of the limitations of these access modes, VSI Fortran supports a third access mode, called **keyed access**, which allows you to retrieve records, at random or in sequence, based on key fields that are established when you create a file with indexed organization. (See Section 6.9.2 for details about keyed access mode).

You can access files with indexed organization using sequential access or keyed access, or a combination of both.

- Keyed access retrieves records randomly based on the particular key fields and key values that you specify.
- Sequential access retrieves records in a sequence based on the direction of the key and on the values within the particular key field that you specify.

Once you have read a record by means of an indexed read request, you can then use a sequential read request to retrieve records with ascending key field values, beginning with the key field value in the record retrieved by the initial read request.

Indexed organization is especially suitable for maintaining complex files in which you want to select records based on one of several criteria. For example, a mail-order firm could use an indexed organization file to store its customer list. Key fields could be a unique customer order number, the customer's zip code, and the item ordered. Reading sequentially based on the zip-code key field would enable you to produce a mailing list sorted by zip code. A similar operation based on customer-order-number key field or item-number key field would enable you to list the records in sequences of customer order numbers or item numbers.

12.2. Creating an Indexed File

You can create a file with an indexed organization by using either of these methods:

- Use the Fortran OPEN statement to specify the file options supported by VSI Fortran.

- Use the RMS EDIT/FDL Utility to select features not directly supported by VSI Fortran.

Any indexed file created with EDIT/FDL can be accessed by VSI Fortran I/O statements.

When you create an indexed file, you define certain fields within each record as key fields. The **primary key**, identified as key number zero, must be present as a field in every record. **Alternate keys** are numbered from 1 through 254. An indexed file can have as many as 255 key fields (1 primary key and up to 254 alternate keys) defined. In practice, however, few applications require more than 3 or 4 key fields.

The data types used for key fields must be INTEGER (KIND=1), INTEGER (KIND=2), INTEGER (KIND=4), INTEGER (KIND=8), or CHARACTER.

In designing an indexed file, you must decide the byte positions of the key fields. For example, in creating an indexed file for use by a mail-order firm, you might define a file record to consist of the following fields:

```
STRUCTURE /FILE_REC_STRUCT/
  INTEGER(KIND=4) ORDER_NUMBER      ! Positions 1:4, key 0
  CHARACTER(LEN=20)  NAME           ! Positions 5:24
  CHARACTER(LEN=20)  ADDRESS        ! Positions 25:44
  CHARACTER(LEN=19)  CITY           ! Positions 45:63
  CHARACTER(LEN=2)   STATE          ! Positions 64:65
  CHARACTER(LEN=9)   ZIP_CODE       ! Positions 66:74, key 1
  INTEGER(KIND=2)    ITEM_NUMBER    ! Positions 75:76, key 2
END STRUCTURE
   .
   .
   .
RECORD /FILE_REC_STRUCT/ FILE_REC
```

Instead of using a record structure, you can define a the fields of a record using a derived-type definition with the SEQUENCE statement:

```
TYPE FILE_REC
  SEQUENCE
  INTEGER(KIND=4) ORDER_NUMBER      ! Positions 1:4, key 0
  CHARACTER(LEN=20)  NAME           ! Positions 5:24
  CHARACTER(LEN=20)  ADDRESS        ! Positions 25:44
  CHARACTER(LEN=19)  CITY           ! Positions 45:63
  CHARACTER(LEN=2)   STATE          ! Positions 64:65
  CHARACTER(LEN=9)   ZIP_CODE       ! Positions 66:74, key 1
  INTEGER(KIND=2)    ITEM_NUMBER    ! Positions 75:76, key 2
END TYPE FILE_REC
   .
   .
   .
```

Given this record definition, you can use the following OPEN statement to create an indexed file:

```
OPEN (UNIT=10, FILE='CUSTOMERS.DAT', STATUS='NEW', &
  ORGANIZATION='INDEXED', ACCESS='KEYED', RECORDTYPE='VARIABLE', &
  FORM='UNFORMATTED', RECL=19, &
  KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER), &
  IOSTAT=IOS, ERR=9999)
```

This OPEN statement establishes the attributes of the file, including the definition of a primary key and two alternate keys. The definitions of the integer keys do not explicitly state INTEGER (KIND=4) and

INTEGER (KIND=2). The data type sizes are determined by the number of character positions allotted to the key fields (4- and 2-digit positions in this case respectively).

If you specify the KEY keyword when opening an existing file, the key specification that you give must match that of the file.

VSI Fortran uses RMS default key attributes when creating an indexed file. These defaults are as follows:

- The values in primary key fields cannot be changed when a record is rewritten. Duplicate values in primary key fields is prohibited.
- The values in alternate key fields can be changed. Duplicate values in alternate key fields is permitted.

You can use the EDIT/FDL Utility or a USEROPEN routine to override these defaults and to specify other values not supported by VSI Fortran, such as null key field values, null key names, and key data types other than integer and character.

For More Information:

- On the use of the USEROPEN keyword in OPEN statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On indexed file options, see the *VSI OpenVMS Record Management Services Reference Manual*.
- On the EDIT/FDL Utility, see the *VMS File Definition Language Facility Manual* and the *Guide to OpenVMS File Applications*.

12.3. Writing Records to an Indexed File

You can write records to an indexed file with either formatted or unformatted indexed WRITE statements. Each write operation inserts a new record into the file and updates the key indexes so that the new record can be retrieved in a sequential order based on the values in the respective key fields.

For example, you could add a new record to the file for the mail-order firm (see Section 12.2) with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

12.3.1. Duplicate Values in Key Fields

It is possible to write two or more records with the same value in a single key field. The attributes specified for the file when it was created determine whether this duplication is allowed. By default, VSI Fortran creates files that allow duplicate alternate key field values and prohibit duplicate primary key field values. If duplicate key field values are present in a file, the records with equal values are retrieved on a first-in/first-out basis.

For example, assume that five records are written to an indexed file in this order (for clarity, only key fields are shown):

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375
942	02163	2736

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
903	14853	375
1348	44901	1047
1263	33032	690

If the file is later opened and read sequentially by primary key (ORDER_NUMBER), the order in which the records are retrieved is not affected by the duplicated value (375) in the ITEM_NUMBER key field. In this case, the records would be retrieved in the following order:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
903	14853	375
942	02163	2736
1023	70856	375
1263	33032	690
1348	44901	1047

However, if the read operation is based on the second alternate key (ITEM_NUMBER), the order in which the records are retrieved is affected by the duplicate key field value. In this case, the records would be retrieved in the following order:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375
903	14853	375
1263	33032	690
1348	44901	1047
942	02163	2736

The records containing the same key field value (375) are retrieved in the order in which they were written to the file.

12.3.2. Preventing the Indexing of Alternate Key Fields

When writing to an indexed file that contains variable-length records, you can prevent entries from being added to the key indexes for any alternate key fields. This is done by omitting the names of the alternate key fields from the WRITE statement. The omitted alternate key fields must be at the end of the record; another key field cannot be specified after the omitted key field.

For example, the last record (ORDER_NUMBER 1263) in the mail-order example could be written with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC.ORDER_NUMBER, FILE_REC.NAME, &
   FILE_REC.ADDRESS, FILE_REC.CITY, FILE_REC.STATE, FILE_REC.ZIP_CODE
```

Because the field name FILE_REC.ITEM_NUMBER is omitted from the WRITE statement, an entry for that key field is not created in the index. As a result, an attempt to read the file using the alternate key ITEM_NUMBER would not retrieve the last record and would produce the following listing:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
903	14853	375
1348	44901	1047
942	02163	2736

You can omit only trailing alternate keys from a record; the primary key must always be present.

12.4. Reading Records from an Indexed File

You can read records in an indexed file with either sequential or indexed READ statements (formatted or unformatted) under the keyed mode of access. By specifying ACCESS= KEYED in the OPEN statement, you enable both sequential and keyed access to the indexed file.

Indexed READ statements position the file pointers (see Section 12.7) at a particular record, determined by the key field value, the key-of-reference, and the match criterion. Once you retrieve a particular record by an indexed READ statement, you can then use sequential access READ statements to retrieve records with increasing key field values.

The form of the external record's key field must match the form of the value you specify in the KEY keyword. If the key field contains character data, you should specify the KEY keyword value as a CHARACTER data type. If the key field contains binary data, then the KEY keyword value should be of INTEGER data type.

If you write a record to an indexed file with formatted I/O, the data type is converted from its internal representation to an external representation. As a result, the key value must be specified in the external form when you read the data back with an indexed read. Otherwise, a match will occur when you do not expect it.

The following VSI Fortran program segment prints the order number and zip code of each record where the first five characters of the zip code are greater than or equal to 10000 but less than 50000:

```
! Read first record with ZIP_CODE key greater than or equal to '10000'.

   READ (UNIT=10,KEYGE='10000',KEYID=1,IOSTAT=IOS,ERR=9999) FILE_REC

!  While the zip code previously read is within range, print the
!  order number and zip code, then read the next record.

   DO WHILE (FILE_REC.ZIP_CODE .LT. '50000')
     PRINT *, 'Order number', FILE_REC.ORDER_NUMBER, 'has zip code', &
           FILE_REC.ZIP_CODE
    READ (UNIT=10,IOSTAT=IOS,END=200,ERR=9999) FILE_REC

! END= branch will be taken if there are no more records in the file.

   END DO
200  CONTINUE
```

The error branch on the keyed READ in this example is taken if no record is found with a zip code greater than or equal to 10000; an attempt to access a nonexistent record is an error. If the sequential READ has accessed all records in the file, an end-of-file status occurs, as with other file organizations.

If you want to detect a failure of the keyed READ, you can examine the I/O status variable, IOS, for the appropriate error number (see Table 7.1 for a list of the returned error codes).

12.5. Updating Records in an Indexed File

The REWRITE statement updates existing records in an indexed file. You cannot replace an existing record simply by writing it again; a WRITE statement would attempt to add a new record.

An update operation is accomplished in two steps:

1. You must read the record in order to make it the current record.

2. You execute the REWRITE statement.

For example, to update the record containing ORDER_NUMBER 903 (see prior examples) so that the NAME field becomes 'Theodore Zinck', you might use the following Fortran code segment:

```
READ (UNIT=10,KEY=903,KEYID=0,IOSTAT=IOS,ERR=9999) FILE_REC
FILE_REC.NAME = 'Theodore Zinck'
REWRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

When you rewrite a record, key fields may change. The attributes specified for the file when it was created determine whether this type of change is permitted. The primary key value can never change on a REWRITE operation. If necessary, delete the old record and write a new record.

12.6. Deleting Records from an Indexed File

The DELETE statement allows you to delete records from an indexed file. The DELETE and REWRITE statements are similar; a record must first be locked by a READ statement before it can be operated on.

The following Fortran code segment deletes the second record in the file with ITEM_NUMBER 375 (refer to previous examples):

```
READ (UNIT=10,KEY=375,KEYID=2,IOSTAT=IOS,ERR=9999)
READ (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
IF (FILE_REC.ITEM_NUMBER .EQ. 375) THEN
    DELETE (UNIT=10, IOSTAT=IOS, ERR=9999)
ELSE
    PRINT *, 'There is no second record.'
END IF
```

Deletion removes a record from all defined indexes in the file.

12.7. Current Record and Next Record Pointers

The RMS file system maintains two pointers into an open indexed file:

- The **next record pointer** indicates the record to be retrieved by a sequential read. When you open an indexed file, the next record pointer indicates the record with the lowest primary key field value. Subsequent sequential read operations cause the next record pointer to be the one with the next higher value in the same key field. In case of duplicate key field values, records are retrieved in the order in which they were written.

- The **current record pointer** indicates the record most recently retrieved by a READ operation; it is the record that is locked from access by other programs sharing the file.

The current record is the one operated on by the REWRITE statement and the DELETE statement. The current record is undefined until a read operation is performed on the file. Any file operation other than a read causes the current record pointer to become undefined. Also, an error results if a rewrite or delete operation is performed when the current record pointer is undefined.

12.8. Exception Conditions When Using Indexed Files

You can expect to encounter certain exception conditions when using indexed files. The two most common of these conditions involve valid attempts to read locked records and invalid attempts to create duplicate keys. Provisions for handling both of these situations should be included in a well-written program.

When an indexed file is shared by several users, any read operation may result in a “specified record locked” error. One way to recover from this error condition is to ask if the user would like to reattempt the read. If the user's response is positive, then the program can go back to the READ statement. For example:

```
    INCLUDE '($FORIOSDEF)'
 .
 .
 .

100  READ (UNIT=10,IOSTAT=IOS) DATA

   IF (IOS .EQ. FOR$IOS_SPERECLOC) THEN
     TYPE *, 'That record is locked. Press RETURN'
     TYPE *, 'to try again, or Ctrl/Z to discontinue'
     READ (UNIT=*,FMT=*,END=900)
     GO TO 100
   ELSE IF (IOS .NE. 0) THEN
     CALL ERROR (IOS)
   END IF
```

You should avoid looping back to the READ statement without first providing some type of delay (caused by a request to try again, or to discontinue, as in this example). If your program reads a record but does not intend to modify the record, you should place an UNLOCK statement immediately after the READ statement. This technique reduces the time that a record is locked and permits other programs to access the record.

The second exception condition, creation of duplicate keys, occurs when your program tries to create a record with a key field value that is already in use. When duplicate key field values are not desirable, you might have your program prompt for a new key field value whenever an attempt is made to create a duplicate. For example:

```
   INCLUDE '($FORIOSDEF)'

200  WRITE (UNIT=10,IOSTAT=IOS) KEY_VAL, DATA

   IF (IOS .EQ. FOR$IOS_INCKEYCHG) THEN
      TYPE *, 'This key field value already exists. Please'
      TYPE *, 'enter a different key field value, or press'
      TYPE *, 'Ctrl/Z to discontinue this operation.'
      READ (UNIT=*,FMT=300,END=999) KEY_VAL
      GO TO 200
   ELSE IF (IOS .NE. 0) THEN
      CALL ERROR (IOS)
   END IF
```

Chapter 13. Interprocess Communication

This chapter describes how to exchange and share data between local and remote processes:

- Section 13.1: VSI Fortran Program Section Usage
- Section 13.2: Local Processes: Sharing and Exchanging Data
- Section 13.3: Remote Processes: Sharing and Exchanging Data

Local processes involve a single OpenVMS processor, and remote processes involve separate processors that are interconnected by means of DECnet.

13.1. VSI Fortran Program Section Usage

You may need to change program section attributes to allow shared access to an installed shareable image.

The storage required by a VSI Fortran program unit is allocated in contiguous areas called program sections (PSECTs). The VSI Fortran compiler implicitly declares these PSECTs:

- $CODE$
- $DATA$
- BSS
- $LITERAL$
- $LINK$

Each common block you declare causes allocation of a PSECT with the same name as the common block. (The unnamed common block PSECT is named $BLANK.) Memory allocation and sharing are controlled by the linker according to the attributes of each PSECT; PSECT names and attributes are listed in Table 13.1.

Each procedure in your program is named according to the name specified in the PROGRAM, BLOCK DATA, FUNCTION, or SUBROUTINE statement used in creating the object module. The defaults applied to PROGRAM and BLOCK DATA statements are `source-file-name`$MAIN and `source-file-name`$DATA, respectively.

Table 13.1. PSECT Names and Attributes

PSECT Name	Use	Attributes
$CODE$	Executable code	PIC, CON, REL, LCL, SHR, EXE, NORD, NOWRT, OCTA
$LINK$	Linkage information (procedure descriptors, linkage pairs, and literals)	NOPIC, CON, REL, LCL, NOSHR, NOEXE, RD, NOWRT, OCTA
$DATA$	Initialized user local static variables and compiler temporary variables	NOPIC, CON, REL, LCL, NOSHR, NOEXE, RD, WRT, OCTA

PSECT Name	Use	Attributes
BSS	Uninitialized user local variables and compiler temporary variables	NOPIC, CON, REL, LCL, NOSHR, NOEXE, RD, WRT, OCTA
$LITERAL$	Literals used in FORMAT statements	NOPIC, CON, REL, LCL, NOSHR, NOEXE, RD, WRT, OCTA
$BLANK	Blank common block	NOPIC, OVR, REL, GBL, NOSHR, NOEXE, RD, WRT, OCTA
names	Named common blocks	NOPIC, OVR, REL, GBL, NOSHR, NOEXE, RD, WRT, OCTA

You can use the cDEC$ PSECT (such as !DEC$ PSECT) directive or use a linker options file to change some of the attributes of a common block.

Table 13.2 describes the meanings of VSI Fortran PSECT attributes.

Table 13.2. VSI Fortran PSECT Attributes

Attribute	Meaning
PIC/NOPIC	Position independent or position dependent code
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
SHR/NOSHR	Shareable or nonshareable
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable (reserved by VSI)
WRT/NOWRT	Writable or nonwritable
LONG/QUAD/ OCTA	Longword, quadword, or octaword alignment

When the linker constructs an executable image, it divides the executable image into sections. Each image section contains PSECTs that have the same attributes. By arranging image sections according to PSECT attributes, the linker is able to control memory allocation. The linker allows you to allocate memory to your own specification by means of commands you include in an options file that is input to the linker.

For More Information:

- On the cDEC$ PSECT (!DEC$ PSECT) compiler directive statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On examples of linker options files used for shared installed common data, see Section 13.2.2.
- On the linker options file and special program sections, see the *VSI OpenVMS Linker Utility Manual*.

13.2. Local Processes: Sharing and Exchanging Data

Interprocess communication mechanisms provided for local processes include the following capabilities:

- Program image sharing in shareable image libraries
- Data sharing in installed common areas
- Information passing by means of mailboxes
- Information passing over DECnet network links

These capabilities are discussed in the sections that follow.

VOLATILE declarations are required when you use certain run-time features of the operating system, including values that can be read or written by methods other than direct assignment, or during a routine call.

If a variable can be accessed using rules in addition to those provided by the standard Fortran 90/95 language, declare the variable as VOLATILE. For example, if a variable in COMMON can change value by means of an OpenVMS AST routine or condition handler, you must declare that common block variable or the entire COMMON block as volatile.

Consider the following uses of variables as candidates for a VOLATILE declaration if asynchronous access might occur:

- Variables in common blocks
- Variables in modules
- Addresses not saved by the %LOC built-in function
- Variables with the TARGET attribute
- Variables declared with the SAVE statement in recursive routines
- Dummy arguments

  Alternatively, if the only local accesses occur when the variable is passed as a dummy argument, the command-line option /ASSUME=DUMMY_ALIASES can be used instead of a VOLATILE declaration (see Section 5.8.9).
- Variables in shared global sections

13.2.1. Sharing Images in Shareable Image Libraries

If you have a routine that is invoked by more than one program, you should consider establishing it as a shareable image and installing it on your system.

Establishing a routine as a shareable image provides the following benefits:

- Saves disk space: The executable images to which the shareable image is linked do not actually include the shareable image. Only one copy of the shareable image exists.
- Simplifies maintenance: If you use symbol vectors, you can modify, recompile, and relink a shareable image without having to relink the executable images that reference it.

Installing a shareable image as shared (INSTALL command, /SHARED qualifier) can also save memory.

The steps to creating and installing a shareable image are:

1. Compile the source file containing that routine that you want to establish as a shareable image.
2. Link the shareable image object file that results from step 1, specifying any object files that contain routines referenced by the shareable image object file.

   The OpenVMS Linker provides a variety of options that you should consider before performing the link operation. For detailed information on shareable images and linker options, see the *VSI OpenVMS Linker Utility Manual*.
3. Create a shareable image library using the Library Utility's LIBRARY command. For detailed information on creating shareable image libraries, see the *Guide to Creating OpenVMS Modular Procedures*.
4. Install the shareable image (the results of step 3) on your system as a shared image by using the Install Utility's INSTALL command (with the /SHARED qualifier). For detailed information on how to perform this operation, see the *VMS Install Utility Manual*.

Any programs that access a shareable image must be linked with that image. When performing the link operation, you must specify one of the following items on your LINK command:

- The name of the shareable image library containing the symbol table of the shareable image. Use the /LIBRARY qualifier to identify a library file.
- A linker options file that contains the name of the shareable image file. Use the /SHAREABLE qualifier to identify a shareable image file. (If you specify the /SHAREABLE qualifier on the LINK command line and you do not specify an options file, the linker creates a shareable image of the object file you are linking).

The resulting executable image contains the contents of each object module and a pointer to each shareable image.

13.2.2. Sharing Data in Installed Common Areas

Sharing the same data among two or more processes can be done using installed common areas.

Typically, you use an installed common area for interprocess communication or for two or more processes to access the same data simultaneously.

13.2.2.1. Creating and Installing the Shareable Image Common Area

To communicate between processes using a common area, first install the common area as a shareable image:

1. Create the common area: Write a VSI Fortran program that declares the variables in the common area and defines the common area. This program should not contain executable code. For example:

   ```
   COMMON /WORK_AREA/ WORK_ARRAY(8192)
   END
   ```

   This common area can use the BLOCK DATA statement.

   When compiling the source file that contains the common block declarations, consistently use the /ALIGNMENT and /GRANULARITY qualifiers (see Section 13.2.2.3). For example:

   ```
   $ FORTRAN/ALIGN=COMMONS=NATURAL/GRANULARITY=LONGWORD INC_COMMON.F90
   ```

2. Make the common area a shareable image: Compile the program containing the common area and use the LINK/SHAREABLE command to create a shareable image containing the common area. You need to specify a linker options file (shown here as SYS$INPUT to allow typed input) to specify the PSECT attributes of the COMMON block PSECT and include it in the global symbol table:

```
$ LINK/SHAREABLE INC_COMMON ,SYS$INPUT/OPTION
    SYMBOL_VECTOR=(WORK_AREA=PSECT)
    PSECT_ATTR=WORK_AREA,SHR
    Ctrl/Z
```

   With VSI Fortran on OpenVMS Alpha systems, the default PSECT attribute for a common block is NOSHR (see Section 13.1). To use a shared installed common block, you *must* specify one of the following:

   - The SHR attribute in a cDEC$ PSECT directive in the source file
   - The SHR attribute in the linker options file for the shareable image to be installed and for each executable image that references the installed common block

   If the !DEC$ PSECT (same as cDEC$ PSECT) directive specified the SHR attribute, the LINK command is as follows:

```
$ LINK/SHAREABLE INC_COMMON ,SYS$INPUT/OPTION
             SYMBOL_VECTOR=(WORK_AREA=PSECT)
             Ctrl/Z
```

   The source line containing the !DEC$ PSECT directive is as follows:

```
!DEC$ PSECT /INC_COMMON/ SHR
```

3. Copy the shareable image: Once created, you should copy the shareable image into SYS$SHARE: before it is installed. The file protection of the .EXE file must allow write access for the processes running programs that will access the shareable image (shown for Group access in the following COPY command):

```
$ COPY/LOG DISK$:[INCOME.DEV]INC_COMMON.EXE SYS$SHARE:*.*/
PROTECTION=G:RWE
```

   If you do not copy the installed shareable image to SYS$SHARE, before running executable images that reference the installed shareable common image, you must define a logical name that specifies the location of that image.

4. Install the shareable image: Using an account with CMKRNL privilege, invoke the interactive Install Utility. When the INSTALL> prompt appears, type a line containing the following:

   a. The CREATE (or ADD) command

   b. The complete file specification of the shareable image that contains the common area (file type defaults to EXE)

   c. The qualifiers /WRITABLE and /SHARED

   The Install utility installs your shareable image and reissues the INSTALL> prompt. Type EXIT to exit. For example:

```
$ INSTALL
```

```
INSTALL> CREATE SYS$SHARE:INC_COMMON/WRITABLE/SHARED
INSTALL> EXIT
$
```

A disk containing an installed image cannot be dismounted until you invoke the Install Utility and type DELETE, followed by the complete file specification of the image.

For More Information:

- On default PSECT attributes, see Section 13.1.
- On the !DEC$ PSECT (cDEC$ PSECT) compiler directive statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
- On the Install Utility, see the *VMS Install Utility Manual*.
- On synchronizing access to a common block installed as a shareable image, see Section 13.2.2.3.
- On the linker, see Chapter 3 and the *VSI OpenVMS Linker Utility Manual*.
- On sharing common block data using a global section, see Section F.4 and the *VSI OpenVMS Programming Concepts Manual*.
- On page size differences between OpenVMS VAX and OpenVMS Alpha systems, see *Migrating an Application from OpenVMS VAX to OpenVMS Alpha*.

13.2.2.2. Creating Programs to Access the Shareable Image Common Area

After the common area has been installed as a shareable image, use the following steps to access the data from any executable program:

1. Include the same variable declarations and common area declarations in the accessing program or programs.

   All common block data declarations must be compatible wherever the common block is referenced.

2. Compile the program.

   When compiling the program that contains the common block declarations, consistently use the *same* /ALIGNMENT and /GRANULARITY qualifiers used to compile the common block data declaration program that has been installed as a shareable image (see Section 13.2.2.3).

   For example, assume the two programs named INCOME and REPORT that will access a common block WORK_AREA in the installed shareable image INC_COMMON:

   ```
   $ FORTRAN/ALIGN=COMMONS=NATURAL/GRANULARITY=LONGWORD INCOME.F90
   $ FORTRAN/ALIGN=COMMONS=NATURAL/GRANULARITY=LONGWORD REPORT.F90
   ```

3. Link the accessing program against the installed common area program.

   You must use an options file to specify the common area program as a shareable image.

   The following are LINK commands:

   ```
   $ LINK INCOME, INCOME/OPTION
   ```

```
$ LINK REPORT, INCOME/OPTION
```

The linker options file INCOME.OPT contains the following lines:

```
INC_COMMON/SHAREABLE
PSECT_ATTR=WORK_AREA, SHR
```

If a !DEC$ PSECT (cDEC$ PSECT) directive specified the SHR PSECT attribute, the linker options file INCOME.OPT would contain the following line:

```
INC_COMMON/SHAREABLE
```

The source line containing the !DEC$ PSECT directive would be as follows:

```
!DEC$ PSECT /INC_COMMON/ SHR
```

For each executable image that references the installed shareable image containing the shared common area, you must specify the SHR PSECT attribute by using either of these:

- !DEC$ PSECT (cDEC$ PSECT) directive
- Linker options file

The two programs access the same area of memory through the installed common block (program section name) WORK_AREA in the installed shareable image INC_COMMON.

4. If the installed image is not located in SYS$SHARE, you must define a logical name that specifies the location of that image. The logical name (in this example INC_COMMON) is the name of the installed image.

5. Execute the accessing program.

   In the previous series of examples, the two programs INCOME and REPORT access the same area of memory through the installed common block WORK_AREA in the installed shareable image INC_COMMON.

For More Information:

- On the /ALIGNMENT qualifier, see Section 2.3.3.
- On the /GRANULARITY qualifier, see Section 2.3.23.
- On intrinsic data types, see Chapter 8.
- On the Alpha architecture, see the *Alpha Architecture Reference Manual*.
- On the Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.

13.2.2.3. Synchronizing Access

If more than one process or thread will write to a shared global section containing COMMON block data, the user program may need to synchronize access to COMMON block variables.

Compile all programs referencing the shared common area with the same value for the /ALIGNMENT and /GRANULARITY qualifiers. For example:

```
$ FORTRAN/ALIGN=COMMONS=NATURAL /GRANULARITY=LONGWORD INC_COMMON
```

Using /GRANULARITY=LONGWORD for 4-byte variables or /GRANULARITY= QUADWORD for 8-byte variables ensures that adjacent data is not accidentally affected. To ensure access to 1-byte variables, specify /GRANULARITY=BYTE. Because accessing data items less than four bytes slows run-time performance, you might want to consider synchronizing read and write access to the data on the same node.

Typically, programs accessing shared data use common event flag clusters to synchronize read and write access to the data on the same node. In the simplest case, one event flag in a common event flag cluster might indicate that a program is writing data, and a second event flag in the cluster might indicate that a program is reading data. Before accessing the shared data, a program must examine the common event flag cluster to ensure that accessing the data does not conflict with an operation already in progress.

Other ways of synchronizing access on a single node include the OpenVMS lock manager system services (SYS$ENQ and SYS$DEQ), the hibernate and wake system services (SYS$HIBER and SYS$WAKE), or using Assembler code.

For More Information:

- On the use of event flags, see the *VSI OpenVMS System Services Reference Manual*.
- On sharing common block data using a global section, see Section F.4 and the *VSI OpenVMS Programming Concepts Manual*.
- On page size differences between OpenVMS VAX and OpenVMS Alpha systems, see *Migrating an Application from OpenVMS VAX to OpenVMS Alpha*.
- On the /ALIGNMENT qualifier, see Section 2.3.3.
- On the /GRANULARITY qualifier, see Section 2.3.23.
- On intrinsic data types, see Chapter 8.
- On the Alpha architecture, see the *Alpha Architecture Reference Manual*.
- On the Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.

13.2.3. Creating and Using Mailboxes to Pass Information

It is often useful to exchange data between processes, as when synchronizing execution or sending messages.

A mailbox is a record-oriented pseudo I/O device that allows you to pass data from one process to another. Mailboxes are created by the Create Mailbox system service (SYS$CREMBX). The following sections describe how to create mailboxes and how to send and receive data using mailboxes.

13.2.3.1. Creating a Mailbox

SYS$CREMBX creates the mailbox and returns the number of the I/O channel assigned to the mailbox. You must specify a variable for the I/O channel. You should also specify a logical name to be associated with the mailbox. The logical name identifies the mailbox for other processes and for VSI Fortran I/O statements.

The SYS$CREMBX system service also allows you to specify the message and buffer sizes, the mailbox protection code, and the access mode of the mailbox; however, the default values for these arguments are usually sufficient.

The following segment of code creates a mailbox named MAILBOX. The number of the I/O channel assigned to the mailbox is returned in ICHAN.

```
INCLUDE '($SYSSRVNAM)'
INTEGER (KIND=2) ICHAN
ISTATUS = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
```

Note

Do not use MAIL as the logical name for a mailbox. If you do so, the system will not execute the proper image in response to the OpenVMS command MAIL.

13.2.3.2. Sending and Receiving Data Using Mailboxes

Sending data to and receiving data from a mailbox is like other forms of VSI Fortran I/O. The mailbox is simply treated as a record-oriented I/O device.

Use VSI Fortran formatted sequential READ and WRITE statements to send and receive messages. The data transmission is performed synchronously; a program that writes a message to a mailbox waits until the message is read, and a program that reads a message from a mailbox waits until the message is written before it continues transmission. When the writing program closes the mailbox, an end-of-file condition is returned to the reading program.

Do not attempt to write a record of zero length to a mailbox; the program reading the mailbox interprets this record as an end-of-file. Zero-length records are produced by consecutive slashes in FORMAT statements.

The following sample program creates a mailbox assigned with the logical name MAILBOX. The program then performs an open operation specifying the logical name MAILBOX as the file to be opened. It then reads file names from FNAMES.DAT and writes them to the mailbox until all of the records in the file have been transmitted.

```
      CHARACTER (LEN=64) FILENAME
      INCLUDE '($SYSSRVNAM)'
      INTEGER (KIND=2) ICHAN
      INTEGER (KIND=4) STATUS

      STATUS = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
      IF (.NOT. STATUS) GO TO 99

      OPEN (UNIT=9, FILE='MAILBOX', STATUS='NEW',  &
            CARRIAGECONTROL='LIST', ERR=99)

      OPEN (UNIT=8, FILE='FNAMES.DAT', STATUS='OLD')

 10   READ (8,100,END=98) FILENAME
      WRITE (9,100) FILENAME

100   FORMAT(A)
      GO TO 10

 98   CLOSE (UNIT=8)
```

```
      CLOSE (UNIT=9)
      STOP

 99   WRITE (6,*) 'Mailbox error'
      STOP
      END
```

The following sample program reads messages from a mailbox that was assigned the logical name MAILBOX when it was created. The messages comprise file names, which the program reads. The program then types the files associated with the file names.

```
      CHARACTER(LEN=64)  FILNAM
      CHARACTER(LEN=123)  TEXT

      OPEN (UNIT=1, FILE='MAILBOX', STATUS='OLD')
   1  READ (1,100,END=12) FILNAM
 100  FORMAT (A)
      OPEN (UNIT=2, FILE=FILNAM, STATUS='OLD')
      OPEN (UNIT=3, FILE='SYS$OUTPUT', STATUS='NEW')

   2  READ (2,100,END=10) TEXT
      WRITE (3,100) TEXT
      GO TO 2

  10  CLOSE (UNIT=2)
      CLOSE (UNIT=3)
      GO TO 1
  12  END
```

For More Information:

- On calling system services, see Chapter 10.
- On the arguments supplied to the Create Mailbox system service, see the *VSI OpenVMS System Services Reference Manual*.

13.3. Remote Processes: Sharing and Exchanging Data

If your computer is a node in a DECnet network, you can communicate with other nodes in the network by means of standard VSI Fortran I/O statements. These statements let you exchange data with a program at the remote computer (task-to-task communication) and access files at the remote computer (resource sharing). There is no apparent difference between these intersystem exchanges and the local interprocess and file access exchanges.

Remote file access and task-to-task communications are discussed separately in the sections that follow.

The system manager at the remote system needs to create the necessary network objects and security controls (such as proxy access). Network file specifications might need to use access control strings, depending on how the remote system access has been implemented.

For More Information:

- On OpenVMS system management, see the *VSI OpenVMS System Manager's Manual*.

- On accessing files across networks, see the *VSI OpenVMS User's Manual*.

13.3.1. Remote File Access

To access a file on a remote system, include the remote node name in the file name specification. For example:

```
BOSTON::DBA0:[SMITH]TEST.DAT;2
```

To make a program independent of the physical location of the files it accesses, you can assign a logical name to the network file specification as shown in the following example:

```
$ DEFINE INVFILE MIAMI::DR4:[INV]INVENT.DAT
```

The logical name INVFILE now refers to the remote file and can be used in the program. For example:

```
OPEN (UNIT=10, FILE='INVFILE', STATUS='OLD')
```

To process a file on the local network node, reassign the logical name; you do not need to modify the source program.

13.3.2. Network Task-to-Task Communication

Network task-to-task communication allows a program running on one network node to interact with a program running on another network node. This interaction is accomplished with standard VSI Fortran I/O statements and looks much like an interactive program/user session.

The steps involved in network task-to-task communications are:

1. **Request the network connection.** The originating program initiates task-to-task communication. It opens the remote task file with a special file name syntax: the name of the remote task file is preceded with TASK= and surrounded with quotation marks. For example:

   ```
   BOSTON::"TASK=UPDATE"
   ```

   Unless the remote task file is contained in the default directory for the remote node's DECnet account, you must specify the pertinent account information (a user name and password) as part of the node name:

   ```
   BOSTON"username password"::"TASK=UPDATE"
   ```

   The form of the remote task file varies, depending on the remote computer's operating system. For OpenVMS systems, this task file is a command file with a file type of COM. The network software submits the command file as a batch job on the remote system.

2. **Complete the network connection.** When the remote task starts, it must complete the connection back to the host. On OpenVMS systems, the remote task completes this connection by performing an open operation on the logical name SYS$NET. When opening the remote task file or SYS$NET, specify either FORM= UNFORMATTED or the combination of FORM= FORMATTED and CARRIAGECONTROL= NONE.

3. **Exchange messages.** When the connection is made between the two tasks, each program performs I/O using the established link.

   Task-to-task communication is synchronous. This means that when one task performs a read, it waits until the other task performs a write before it continues processing.

4. **Terminate the network connection.** To prevent losing data, the program that receives the last message should terminate the network connection using the CLOSE statement. When the network connection is terminated, the cooperating image receives an end-of-file error.

The following is a complete example showing how VSI Fortran programs can exchange information over a network. The originating program prompts for an integer value and sends the value to the remote program. The remote program then adds one to the value and returns the value to the originating program. It is assumed that the remote operating system is an OpenVMS system.

The originating program on the local node contains the following source code:

```
      OPEN (UNIT=10, FILE='PARIS::"TASK=REMOTE"', STATUS='OLD', &
            FORM='UNFORMATTED', ACCESS='SEQUENTIAL', IOSTAT=IOS, ERR=999)

!  Prompt for a number
      PRINT 101
101   FORMAT ($,' ENTER A NUMBER: ')
      ACCEPT *,N

!  Perform the network I/O
      WRITE (UNIT=10, IOSTAT=IOS, ERR=900) N
      READ  (UNIT=10, IOSTAT=IOS, ERR=900) N

!  Display the number and process errors

      PRINT 102, N
102   FORMAT (' The new value is ',I11)
      GO TO 999

900   PRINT *, 'Unexpected I/O Error Number ', IOS
999   CLOSE (UNIT=10)
      END PROGRAM
```

The task file REMOTE.COM on the remote node contains the following OpenVMS DCL commands:

```
$ DEFINE SYS$PRINT NL:                ! Inhibit printing of log
$ RUN DB0:[NET]REMOTE.EXE             ! Run remote program
$ PURGE/KEEP=2 REMOTE.LOG             ! Delete old log files
```

The remote program PARIS::DB0:[NET]REMOTE.EXE contains the following source code:

```
    OPEN (UNIT=10, FILE='SYS$NET', FORM='UNFORMATTED', &
          ACCESS='SEQUENTIAL', STATUS='OLD')
    READ (UNIT=10) N
    N = N + 1
    WRITE (UNIT=10) N
    CLOSE (UNIT=10)
    END PROGRAM
```

For More Information:

On using DECnet, refer to the *DECnet for OpenVMS Networking Manual* and *DECnet for OpenVMS Guide to Networking*.

Chapter 14. Condition-Handling Facilities

This chapter describes:

- Section 14.1: Overview of Condition-Handling Facilities
- Section 14.2: Overview of the Condition-Handling Facility
- Section 14.3: Default Condition Handler
- Section 14.4: User-Program Interactions with the CHF
- Section 14.5: Operations Performed in Condition Handlers
- Section 14.6: Coding Requirements of Condition Handlers
- Section 14.7: Returning from a Condition Handler
- Section 14.8: Matching Condition Values to Determine Program Behavior
- Section 14.9: Changing a Signal to a Return Status
- Section 14.10: Changing a Signal to a Stop
- Section 14.11: Checking for Arithmetic Exceptions
- Section 14.12: Checking for Data Alignment Traps
- Section 14.13: Condition Handler Example

14.1. Overview of Condition-Handling Facilities

An **exception condition**, as the term is used in this chapter, is an event, usually an error, that occurs during the execution of a program and is detected by system hardware or software or by logic in a user application program. To resolve exception conditions, you can create a **condition-handler routine**.

This chapter addresses error handling only as it relates to the creation and use of condition-handler routines. Condition-handler routines are specific to the OpenVMS operating system. For a general discussion of error handling, see Chapter 7.

Examples of the types of exception conditions detected by system hardware and software are:

- Hardware exceptions, such as memory access violations.
- Software exceptions, such as output conversion errors, end-of-file conditions, and invalid arguments to mathematical procedures.

When an exception condition is detected by system hardware or software or by your program, that condition is **signaled** (by means of a **signal call**) to the OpenVMS Condition-Handling Facility (CHF). The CHF then invokes one or more condition-handler routines that will attempt to either resolve the condition or terminate the processing in an orderly fashion.

The CHF allows a main program and each subprogram that follows it, regardless of call depth, to establish a condition-handler routine (one per program unit). Each of these condition-handler routines can potentially handle any or all software or hardware events that are treated as exception conditions by the user program or by the system hardware or software. More than one condition handler for a given condition can be established by different program units in the call stack (see the *VSI OpenVMS Programming Concepts Manual*).

The address of the condition handler for a particular program unit is placed in the call frame for that unit in the run-time call stack.

When the program unit returns to its caller, the call frame is removed and the condition handler for that program unit can no longer be accessed by the CHF. Multiple condition handlers can be accessed by the CHF in the processing of a single exception condition signal. A process-wide handler can be established using the SYS$SETEXV system service.

Throughout this chapter, the term **program unit** refers to an executable Fortran main program, subroutine, or function.

For More Information:

- On multiple condition handlers, see Section 14.7.
- On condition handling concepts, see the *VSI OpenVMS Programming Concepts Manual*.

14.2. Overview of the Condition-Handling Facility

The Condition-Handling Facility (CHF) receives control and coordinates processing of all exception conditions that are signaled to it. The signals are issued under the following circumstances:

- When a user program detects an application-dependent exception condition
- When system hardware or a VSI Fortran software component detects a system-defined exception condition

In cases where the default condition handling is insufficient (see Section 14.3), you can develop your own handler routines and use the VSI Fortran intrinsic function LIB$ESTABLISH to identify your handlers to the CHF. Typically, your needs for special condition handling are limited to the following types of operations:

- To respond to condition codes that are signaled instead of being returned, as in the case of integer overflow errors. (Section 14.9 describes the system-defined handler LIB$SIG_TO_RET, which allows you to treat signals as return values).
- To add additional messages to those messages associated with the originally signaled condition code or to log the occurrence of various application-specific or system-specific conditions.

When an exception condition is detected by a system hardware or software component or by a component in the user application program, the component calls the CHF by means of a signal routine (LIB$SIGNAL or LIB$STOP), passing a value to the CHF that identifies the condition. The CHF takes program control away from the routine that is currently executing and begins searching for a condition-handler routine to call. If it finds one, it establishes a call frame on the run-time call stack and then invokes the handler. The handler routine then attempts to deal with the condition.

The sections that follow describe the CHF in detail – how it operates, how user programs can interact with it, and how users can code their own condition-handling routines:

- Section 14.3 describes default condition handlers established by the system.
- Section 14.4 describes how a user program makes a condition handler known to the CHF and how it signals a condition and passes arguments.
- Section 14.5, Section 14.6, and Section 14.7 contain information about writing a condition-handling routine.
- Section 14.8, Section 14.9, and Section 14.10 describe using the RTL routines LIB$MATCH_COND, LIB$SIG_TO_RET, and LIB$SIG_TO_STOP.
- Section 14.11 contains information on checking for arithmetic exceptions.
- Section 14.12 contains information on checking for data alignment traps.
- Section 14.13 contains some examples of the use of condition handlers.

14.3. Default Condition Handler

When the system creates a VSI Fortran user process, it establishes a system-defined condition handler that will be invoked by the CHF under the following circumstances:

- No user-established condition handlers exist in the call stack. (Any user-established condition handlers in the call stack are always invoked before the default handler is invoked).
- All of the user-established condition handlers in the call stack return the condition code SS$_RESIGNAL to the CHF. (The SS$_RESIGNAL condition code causes the CHF to search for another condition handler. See Section 14.7).

When establishing the default handler, the system has two handlers to choose from: the traceback handler and the catchall handler.

- **Traceback Handler**. Displays the message associated with the:
  - Signaled condition code
  - Traceback message
  - Program unit name and line number of the statement that resulted in the exception condition
  - Relative and absolute program counter values

  In addition, the traceback handler displays the names of the program units in the current calling sequence and the line number of the invocation statements. (For exception conditions with a severity level of warning or error, the number of the next statement to be executed is also displayed).

  After displaying the error information, the traceback handler continues program execution or, if the error is severe, terminates program execution. If the program terminates, the condition value becomes the program exit status.
- **Catchall Handler**. Displays the message associated with the condition code and then either continues program execution or, if the error is severe, terminates execution. If the program

terminates, the condition value becomes the program exit status. In user mode, the catchall handler can be a list of handlers.

The /DEBUG and /TRACEBACK qualifiers – on the FORTRAN and LINK command lines, respectively – determine which default handler is enabled. If you take the defaults for these qualifiers, the traceback handler is established as the default handler. To establish the catchall handler as the default, specify /NODEBUG or /DEBUG=NOTRACEBACK on the FORTRAN command line and /NOTRACEBACK on the LINK command line.

Use the FORTRAN command /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier to ensure precise exception reporting.

For More Information:

- On condition values, see Table 7.1.
- On the FORTRAN command /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier, see Section 2.3.46.

14.4. User-Program Interactions with the CHF

User-program interactions with the CHF are strictly optional and application-dependent. In each program unit, you have the option of establishing (and removing) a single condition handler to handle exceptions that may occur in that program unit or in subsequent subprograms (regardless of call depth). Once a program unit returns to its caller, its call frame is removed and any condition handler that the program unit has established becomes inaccessible.

The condition handler established by the user program can be coded to handle an exception condition signaled either by system hardware, a VSI Fortran system software component, or the user program itself. User-program signals are issued by means of the LIB$STOP and LIB$SIGNAL routines described in Section 14.4.2.

Although condition handlers offer a convenient and structured approach to handling exception conditions, they can have a significant impact on run-time performance when a condition handler is actually used. For commonly occurring application-specific conditions within a loop, for example, it may be wise to use other methods of dealing with the conditions. The best use of the facility is in large applications in which occasional exception conditions requiring special handling are anticipated.

The following sections describe how to establish and remove condition handlers and how to signal exception conditions.

14.4.1. Establishing and Removing Condition Handlers

To establish a condition handler, call the LIB$ESTABLISH intrinsic function. (For compatibility with Compaq Fortran 77 for OpenVMS VAX Systems, VSI Fortran provides the LIB$ESTABLISH and LIB$REVERT routines as intrinsic functions).

The form of the call can be as a subroutine or a function reference:

```
CALL LIB$ESTABLISH (new-handler)
old-handler=LIB$ESTABLISH(new-handler)
```

new-handler

Specifies the name of the routine to be set up as a condition handler.

old-handler

Receives the address of the previously established condition handler.

LIB$ESTABLISH moves the address of the condition-handling routine into the appropriate process context and returns the previous address of a previously established condition handler.

The handler itself could be user-written or selected from a list of utility functions provided with VSI Fortran. The following example shows how a call to establish a user-written handler might be coded:

```
EXTERNAL HANDLER
CALL LIB$ESTABLISH(HANDLER)
```

In the preceding example, HANDLER is the name of a Fortran function subprogram that is established as the condition handler for the program unit containing these source statements. A program unit can remove an established condition handler in two ways:

- Issue another LIB$ESTABLISH call specifying a different handler.
- Issue the LIB$REVERT call.

The LIB$REVERT call has no arguments and can be a subroutine or a function reference:

```
CALL LIB$REVERT
old-handler=LIB$REVERT()
```

The use of *old-handler* for the LIB$REVERT call is the same as for the LIB$ESTABLISH call.

This call removes the condition handler established in the current program unit. Like LIB$ESTABLISH, LIB$REVERT is provided as an intrinsic function.

When the program unit returns to its caller, the condition handler associated with that program unit is automatically removed (the program unit's stack frame, which contains the condition handler address, is removed from the stack).

14.4.2. Signaling a Condition

When a prescribed condition requiring special handling by a condition handler is detected by logic in your program, you issue a condition signal in your program in order to invoke the CHF. A condition signal consists of a call to one of the two system-supplied signal routines in the following forms:

```
EXTERNAL LIB$SIGNAL, LIB$STOP
CALL LIB$SIGNAL(condition-value, arg, ..., arg)
CALL LIB$STOP(condition-value, arg, ..., arg)
```

condition-value

An INTEGER (KIND=4) value that identifies a particular exception condition (see Section 14.4.3) and can only be passed using the %VAL argument-passing mechanism.

arg

Optional arguments to be passed to user-established condition handlers and the system default condition handlers. These arguments consist of messages and formatted-ASCII-output arguments (see the V*MS Run-Time Library Routines Volume*).

The CHF uses these parameters to build the signal argument array SIGARGS (see Section 14.6) before passing control to a condition handler.

Whether you issue a call to LIB$SIGNAL or LIB$STOP depends on the following considerations:

- If the current program unit can continue after the signal is made, call LIB$SIGNAL. The condition handler can then determine whether program execution continues. After the signal is issued, control is not returned to the user program until one of the condition handlers in the call stack resolves the exception condition and indicates to the CHF that program execution should continue.
- If the condition does not allow the current program unit to continue, call LIB$STOP. (The only way to override a LIB$STOP signal is to perform an unwind operation. See Section 14.7).

Figure 14.1 lists all of the possible effects of a LIB$SIGNAL or LIB$STOP call.

Figure 14.1. Effects of Calls to LIB$SIGNAL or LIB$STOP

Call to:	Signaled Condition Severity <2:0>	Default Handler Gets Control	Handler Specifies Continue	Handler Specifies UNWIND	No Handler Is Found (stack bad)
LIB$SIGNAL or hardware exception	<4	condition message RET	RET	UNWIND	Call last chance handler EXIT
	=4	condition message EXIT	RET	UNWIND	Call last chance handler EXIT
LIB$STOP	force (=4)	condition message EXIT	"cannot continue" EXIT	UNWIND	Call last chance handler EXIT

ZK-5162-GE

In Figure 14.1, “cannot continue” indicates an error that results in the following message:

```
IMPROPERLY HANDLED CONDITION, ATTEMPT TO CONTINUE FROM STOP
```

To pass the condition value, you must use the %VAL argument-passing mechanism (see Section 10.3.3). Condition values are usually expressed as condition symbols (see Section 10.8.1). Condition symbols have either of the following forms:

```
fac$_symbol (VSI-defined)

fac__symbol (user-defined)
```

`fac`

A facility name prefix.

symbol

Identifies a specific condition. (See Table 7.1 for a list of VSI Fortran condition symbols).

In the following example, a signal call passes a condition symbol used to report a missing required privilege.

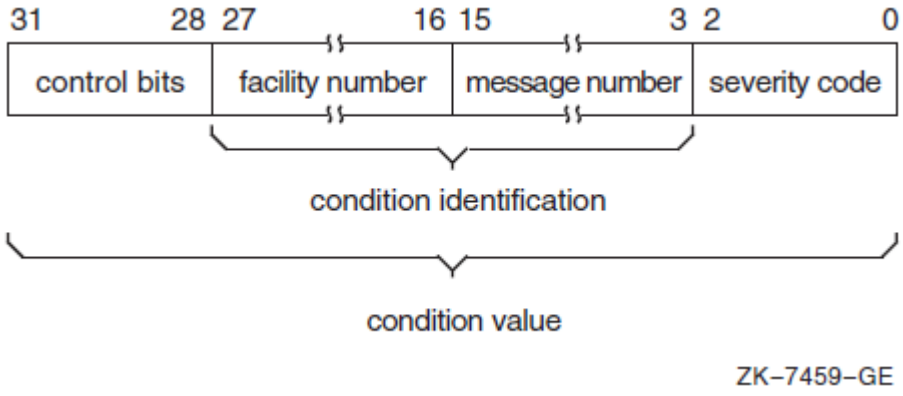
```
CALL LIB$SIGNAL(%VAL(SS$_NOSYSPRV))
```

You can include additional arguments to provide supplementary information about the error. System symbols such as SS$_NOSYSPRV are defined in the library module $SSDEF.

When your program issues a condition signal, the CHF searches for a condition handler by examining the preceding call frames, in order, until it either finds a procedure that handles the signaled condition or reaches the default condition handler. Condition handling procedures should use SS$_RESIGNAL for conditions they are not intended to handle.

14.4.3. Condition Values and Symbols Passed to CHF

The OpenVMS system uses condition values to indicate that a called procedure has either executed successfully or failed, and to report exception conditions. Condition values are INTEGER (KIND=4) values (see the *VSI OpenVMS Programming Concepts Manual* and the *VSI OpenVMS Calling Standard* for details). They consist of fields that indicate which software component generated the value, the reason the value was generated, and the severity of the condition. A condition value has the following fields:



The facility number field identifies the software component that generated the condition value. Bit 27 = 1 indicates a user-supplied facility; bit 27 = 0 indicates a system facility.

The message number field identifies the condition that occurred. Bit 15 = 1 indicates that the message is specific to a single facility; bit 15 = 0 indicates a system-wide message.

Table 14.1 gives the meanings of values in the severity code field.

Table 14.1. Severity Codes for Exception Condition Values

Code (Symbolic Name)	Severity	Response
0 (STS$K_WARNING)	Warning	Execution continues, unpredictable results
1 (STS$K_SUCCESS)	Success	Execution continues, expected results

Code (Symbolic Name)	Severity	Response
2 (STS$K_ERROR)	Error	Execution continues, erroneous results
3 (STS$K_INFORMATION)	Information	Execution continues, informational message displayed
4 (STS$K_SEVERE)	Severe error	Execution terminates, no output
5 - 7	–	Reserved for use by VSI

The symbolic names for the severity codes are defined in the $STSDEF library module in the VSI Fortran Symbolic Definition Library, FORSYSDEF.

A condition handler can alter the severity code of a condition value – either to allow execution to continue or to force an exit, depending on the circumstances.

The condition value is passed in the second element of the array SIGARGS. (See Section 14.6 for detailed information about the contents and use of the array SIGARGS.) In some cases, you may require that a particular condition be identified by an exact match; each bit of the condition value (31:0) must match the specified condition. For example, you may want to process a floating overflow condition only if its severity code is still 4 (that is, only if a previous handler has not changed the severity code).

In many cases, however, you may want to respond to a condition regardless of the value of the severity code. To ignore the severity and control fields of a condition value, use the LIB$MATCH_COND routine (see Section 14.8).

The FORSYSDEF library contains library modules that define condition symbols. When you write a condition handler, you can specify any of the following library modules, as appropriate, with an INCLUDE statement:

- **$CHFDEF –** This library module contains structure definitions for the primary argument list (CHFDEF), the signal array (CHFDEF1), and the mechanism array (CHFDEF2). These symbols have the following form:

  ```
  CHF$_xxxxxxx
  ```

  For example:

  ```
  CHF$_IH_MCH_SAVR0
  ```

- **$FORDEF –** This library module contains definitions for all condition symbols from the VSI Fortran library routines. See Table 7.1 for a list of the VSI Fortran error numbers (IOSTAT values) associated with these symbols. These symbols have the following form:

  ```
  FOR$_error
  ```

  For example:

  ```
  FOR$_INPCONERR
  ```

- **$LIBDEF –** This library module contains definitions for all condition symbols from the OpenVMS general utility library facility. These symbols have the following form:

  ```
  LIB$_condition
  ```

  For example:

  ```
  LIB$_INSVIRMEM
  ```

- **$MTHDEF –** This library module contains definitions for all condition symbols from the mathematical procedures library. These symbols have the following form:

  ```
  MTH$_condition
  ```

  For example:

  ```
  MTH$_SQUROONEG
  ```

- **$SSDEF –** This library module contains definitions for system services status codes, which are frequently used in VSI Fortran condition handlers. These symbols have the following form:

  ```
  SS$_status
  ```

  For example:

  ```
  SS$_BADPARAM
  ```

For More Information:

- On performing an unwind operation, see Section 14.7.
- On a list of VSI Fortran condition symbols, see Table 7.1.
- On the signal array SIGARGS, see Section 14.6.

14.5. Operations Performed in Condition Handlers

A condition handler responds to an exception by analyzing arguments passed to it and by taking appropriate action. Possible actions taken by condition handlers are:

- Condition correction
- Condition reporting
- Execution control

First, the handler must determine whether it can correct the condition identified by the condition code passed by the signal call. If possible, the handler takes the appropriate corrective action and execution continues. If it cannot correct the condition, the handler can resignal the condition; it can request that another condition handler (associated with an earlier program unit in the call stack) attempt to process the exception.

Condition reporting performed by handlers can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execution.
- Signaling the same condition again (resignaling) in order to send the appropriate message to your terminal or log file.
- Changing the severity field of the condition value and resignaling the condition.
- Signaling a different condition, such as producing a message appropriate to a specific application. (The condition handler must establish the application-specific condition handler using LIB$ESTABLISH and then signal the condition using LIB$SIGNAL).

Execution can be affected in a number of ways, such as:

- Continuing from the point of exception. However, if the signal was issued by means of a call to LIB$STOP, the program exits.
- Returning control (unwinding) to the program unit that established the handler. Execution resumes at the point of the call that resulted in the exception. The handler establishes the function value to be returned by the called procedure.
- Returning control (unwinding) to the establisher 's caller (to the program unit that called the program unit that established the handler). The handler establishes the function value to be returned by the program unit that established the handler.

For More Information:

- On returning from condition handlers, see Section 14.7.
- On condition handler examples, see Section 14.13.

14.6. Coding Requirements of Condition Handlers

An VSI Fortran condition handler is an INTEGER (KIND=4) function that has two argument arrays passed to it by the CHF. To meet these requirements, you could define a condition handler as follows:

```
INCLUDE '($CHFDEF)'
INTEGER (KIND=4)  FUNCTION HANDLER(SIGARGS,MECHARGS)
INTEGER (KIND=4)  SIGARGS(*)
RECORD /CHFDEF2/ MECHARGS
```

The CHF creates the signal and mechanism argument arrays SIGARGS and MECHARGS and passes them to the condition handler. Note that the mechanism vector block differs on OpenVMS I64 and OpenVMS Alpha systems (see the *VSI OpenVMS Calling Standard*).

The signal array (SIGARGS) is used by condition handlers to obtain information passed as arguments in the LIB$SIGNAL or LIB$STOP signal call.

Table 14.2 shows the contents of SIGARGS.

Table 14.2. Contents of SIGARGS

Array Element	CHFDEF1 Field Name	Contents
SIGARGS(1)	CHF$IS_SIG_ARGS	Argument count (n)
SIGARGS(2)	CHF$IS_SIG_NAME	Condition code
SIGARGS(3 to n-1) . .	CHF$IS_SIG_ARG1 (first argument)	Zero or more additional arguments, specific to the condition code in SIGARGS(2)
SIGARGS(n)	None	PC (program counter)
SIGARGS(n+1)	None	PS (processor status), lower 32-bits of the 64-bit OpenVMS processor status

- The notation `n` represents the argument count, that is, the number of elements in SIGARGS, not including the first element.
- The first array element SIGARGS(1) or CHF$IS_SIG_ARGS indicates how many additional arguments are being passed in this array. The count does not include this first element.
- SIGARGS(2) or CHF$IS_SIG_NAME indicates the signaled condition (condition value) specified by the call to LIB$SIGNAL or LIB$STOP. If more than one message is associated with the exception condition, the condition value in SIGARGS(2) belongs to the first message.
- SIGARGS(3) or CHF$IS_SIG_ARG1 varies with the type of condition code in SIGARGS(2). This could contain the message description for the message associated with the condition code in SIGARGS(2), including secondary messages from RMS and system services. The format of the message description varies depending on the type of message being signaled. For more information, see the SYS$PUTMSG description in the *VSI OpenVMS System Services Reference Manual*.

  Additional arguments, SIGARGS(`n`-1), can be specified in the call to LIB$SIGNAL or LIB$STOP (see Section 14.4.2).
- The second-to-last element, SIGARGS(`n`), contains the value of the program counter (PC).

  If the condition that caused the signal was a fault (occurring during the instruction's execution), the PC contains the address of that instruction.

  If the condition that caused the signal was a trap (occurring at the end of the instruction), the PC contains the address of the instruction following the call that signaled the condition code.
- The last element, SIGARGS(`n`+1), reflects the value of the processor status (PS) at the time the signal was issued.

A condition handler is usually written in anticipation of a particular condition code or set of condition codes. Because handlers are invoked as a result of any signaled condition code, you should begin your handler routine by comparing the condition code passed to the handler (element 2 of SIGARGS) with the condition codes expected by the handler. If the signaled condition code is not an expected code, you should resignal the condition code by equating the function value of the handler to the global symbol SS$_RESIGNAL (see Section 14.7).

The mechanism array (MECHARGS) is used to obtain information about the procedure activation of the program unit that established the condition handler. MECHARGS is a 90-element array, but only integer registers (R *n*) and floating-point registers (F *n*) are contained beyond element 12 (R0 is in elements 13 and 14 and all registers are 64 bits).

Table 14.3 shows the contents of MECHARGS on OpenVMS Alpha systems.

The contents are essentially the same on I64 and Alpha up to the CHF$IL_MCH_SAVR10_HIGH field name. After that, the I64 registers are saved in the I64 order and the contents of MECHARGS become different. For information about the additional field names for I64, see the *VSI OpenVMS System Services Reference Manual*.

Table 14.3. Contents of MECHARGS on OpenVMS Alpha Systems

INTEGER (KIND=4) Array Element	CHFDEF2 Field Name	Contents
MECHARGS(1)	CHF$IS_MCH_ARGS	Argument count

INTEGER (KIND=4) Array Element	CHFDEF2 Field Name	Contents
MECHARGS(2)	CHF$IS_MCH_FLAGS	Flags
MECHARGS(3), MECHARGS(4)	CHF$PH_MCH_FRAME	Frame address pointer
MECHARGS(5)	CHF$IS_MCH_DEPTH	Call depth
MECHARGS(6)	CHF$IS_MCH_RESVD1	Not used
MECHARGS(7), MECHARGS(8)	CHF$PH_MCH_DADDR	Handler data address
MECHARGS(9), MECHARGS(10)	CHF$PH_MCH_ESF_ADDR	Exception stack frame address
MECHARGS(11), MECHARGS(12)	CHF$PH_MCH_SIG_ADDR	Signal array address
MECHARGS(13), MECHARGS(14)	CHF$IH_MCH_SAVR0 CHF$IL_MCH_SAVR0_LOW CHF$IL_MCH_SAVR0_HIGH	R0 low-order 32 bits high-order 32 bits
MECHARGS(15), MECHARGS(16)	CHF$IH_MCH_SAVR1 CHF$IL_MCH_SAVR1_LOW CHF$IL_MCH_SAVR10_HIGH	R1 low-order 32 bits high-order 32 bits
MECHARGS(17) to MECHARGS(42)	CHF$IH_MCH_SAVR *nn*	R16-R28
MECHARGS(43), MECHARGS(44)	CHF$FH_MCH_SAVF0(2)	F0
MECHARGS(45), MECHARGS(46)	CHF$FH_MCH_SAVF1(2)	F1
MECHARGS(47) to MECHARGS(88)	CHF$IH_MCH_SAVF *nn*(2)	F10-F30
MECHARGS(89), MECHARGS(90)	CHF$PH_MCH_SIG64_ADDR	Address of 64-bit form of signal arrray

- CHF$IS_MCH_ARGS or MECHARGS(1) contains the argument count of this array in quadwords, not including this first quadword (the value 43).
- CHF$IS_MCH_FLAGS or MECHARGS(2) contains various flags to communicate additional information. For instance, if bit zero (0) is set, this indicates that the process has already performed a floating-point operation and that the floating-point registers are valid (CHF$S_FPRREGS_VALID=1 and CHF$V_FPRREGS_VALID=0).
- CHF$PH_MCH_FRAME or MECHARGS(3) and MECHARGS(4) contains the address of the call frame on the stack for the program unit that established the handler.
- CHF$IS_MCH_DEPTH or MECHARGS(5) contains the number of calls made between the program unit that established the handler and the program unit that signaled the condition code.
- MECHARGS(6) is not used.

- CHF$PH_MCH_DADDR or MECHARGS(7) and MECHARGS(8) contain the address of the handler data quadword.
- CHF$PH_MCH_ESF_ADDR or MECHARGS(9) and MECHARGS(10) contain the address of the handler stack frame.
- CHF$PH_MCH_SIG_ADDR or MECHARGS(11) and MECHARGS(12) contains the address of the signal array.
- CHF$IH_MCH_SAVR *nn* to CHF$IH_MCH_SAVF *nn* (2) or MECHARGS(13) to MECHARGS(88) contain certain integer and floating-point registers.
- MECHARGS(89), MECHARGS(90): for information about 64-bit signal arrays, see the *VSI OpenVMS Calling Standard*.

Inside a condition handler, you can use any other variables that you need. If they are shared with other program units (for example, in common blocks), make sure that they are declared volatile. This will ensure that compiler optimizations do not invalidate the handler actions. See Section 5.7.3 and the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/] for more information on the VOLATILE statement.

For More Information:

- On condition values, see Section 14.4.3.
- On the Itanium architecture, see the *Intel Itanium Architecture Software Developer's Manual*.
- On register use, see the *Alpha Architecture Reference Manual* and the *VSI OpenVMS Calling Standard*.
- On condition handler examples, see Section 14.13.

14.7. Returning from a Condition Handler

One way that condition handlers control subsequent execution is by specifying a function return value (symbols defined in library module $SSDEF). Function return values and their effects are defined in Table 14.4.

Table 14.4. Condition-Handler Function Return Values

Symbolic Values	Effects
SS$_CONTINUE	If you assign SS$_CONTINUE to the function value of the condition handler, the handler returns control to the program unit at the statement that signaled the condition (fault) or the statement following the one that signaled the condition (trap).
SS$_RESIGNAL	If you assign SS$_RESIGNAL to the function value of the condition handler, or do not specify a function value (function value of zero), the CHF will search for another condition handler in the call stack. If you modify SIGARGS or MECHARGS before resignaling, the modified arrays are passed to the next handler.

A condition handler can also request a call stack unwind by calling SYS$UNWIND before returning. Unwinding the call stack:

- Removes call frames, starting with the call frame for the program unit in which the condition occurred
- Returns control to an earlier program unit in the current calling sequence

In this case, any function return values established by condition handlers are ignored by the CHF.

You can unwind the call stack whether the condition was detected by hardware or signaled by means of LIB$SIGNAL or LIB$STOP. Unwinding is the only way to continue execution after a call to LIB$STOP.

A stack unwind is typically made to one of two places:

- To the establisher of the condition handler that issues the SYS$UNWIND. You pass the call depth (first half of the third element of the MECHARGS array) as the first argument in the call to SYS$UNWIND. Do not specify a second argument. For example:

  ```
  INCLUDE '($CHFDEF)'
  RECORD /CHFDEF2/ MECHARGS
  CALL SYS$UNWIND(MECHARGS.CHF$IS_MCH_DEPTH,)
  ```

  Control returns to the establisher and execution resumes at the point of the call that resulted in the exception.

- To the establisher's caller. Do not specify either of the arguments in the call to SYS$UNWIND. For example:

  ```
  CALL SYS$UNWIND(,)
  ```

  Control returns to the program unit that called the establisher of the condition handler that issues the call to SYS$UNWIND.

The actual stack unwind is not performed immediately after the condition handler issues the call to SYS$UNWIND. It occurs when a condition handler returns control to the CHF.

During the actual unwinding of the call stack, SYS$UNWIND examines each frame in the call stack to determine whether a condition handler was declared. If a handler was declared, SYS$UNWIND calls the handler with the condition value SS_$UNWIND (indicating that the stack is being unwound) in the condition name argument of the signal array. When a condition handler is called with this condition value, that handler can perform any procedure-specific clean-up operations that may be required. After the condition handler returns, the call frame is removed from the stack.

The system service SYS$GOTO_UNWIND performs a similar function.

For More Information:

On an example that uses SYS$UNWIND, see Section 14.13.

14.8. Matching Condition Values to Determine Program Behavior

In many condition-handling situations, you may want to respond to an exception condition regardless of the value of the severity code passed in the condition value. To ignore the severity and control fields of a condition value, use the LIB$MATCH_COND routine as a function in the following form:

```
index = LIB$MATCH_COND(SIGARGS(2),con-1,...con-n)
```

index

An integer variable that is assigned a value for use in a subsequent computed GOTO statement.

con

A condition value.

The LIB$MATCH_COND function compares bits 27:3 of the value in SIGARGS(2) with bits 27:3 of each specified condition value. If it finds a match, the function assigns the index value according to the position of the matching condition value in the list.

If the match is with the third condition value following SIGARGS(2), then `index` = 3. If no match is found, `index` = 0. The value of the index can then be used to transfer control, as in the following example:

```
      INTEGER (KIND=4) FUNCTION HANDL(SIGARGS,MECHARGS)
      INCLUDE '($CHFDEF)'
      INCLUDE '($FORDEF)'
      INCLUDE '($SSDEF)'
      INTEGER (KIND=4) SIGARGS(*)
      RECORD /CHFDEF2/ MECHARGS

      INDEX=LIB$MATCH_COND(SIGARGS(2), FOR$_FILNOTFOU, &
                  FOR$_NO_SUCDEV, FOR$_FILNAMSPE, FOR$_OPEFAI)

      GO TO (100,200,300,400), INDEX
      HANDL=SS$_RESIGNAL
      RETURN
       ...
100   ! Handle FOR$_FILNOTFOU
       ...
200   ! Handle FOR$_NO_SUCDEV
       ...
300   ! Handle FOR$_FILNAMSPE
       ...
400   ! Handle FOR$_OPEFAI
       .
       .
       .
```

If no match is found between the condition value in SIGARGS(2) and any of the values in the list, then INDEX = 0 and control transfers to the next executable statement after the computed GOTO. A match with any of the values in the list transfers control to the corresponding statement in the GOTO list.

If SIGARGS(2) matches the condition symbol FOR$_OPEFAI, control transfers to statement 400.

For More Information:

- On a list of VSI Fortran condition symbols, see Table 7.1.
- On an example that uses LIB$MATCH_COND, see Section 14.13.

14.9. Changing a Signal to a Return Status

When it is preferable to detect errors by signaling, but the calling procedure expects a returned status, LIB$SIG_TO_RET may be used by the procedure that signals. LIB$SIG_TO_RET is a condition handler that converts any signaled condition to a return status. The status is returned to the caller of the procedure that established LIB$SIG_TO_RET.

The arguments for LIB$SIG_TO_RET are:

```
LIB$SIG_TO_RET (sig-args,mch-args)
```

sig-args

Contains the address of the signal argument array (see Section 14.6).

mch-args

Contains the address of the mechanism argument array (see Section 14.6).

You can establish LIB$SIG_TO_RET as a condition handler by specifying it in a call to LIB$ESTABLISH. You can also establish it by calling it from a user-written condition handler. If LIB$SIG_TO_RET is called from a condition handler, the signaled condition is returned as a function value to the caller of the establisher of that handler when the handler returns to the CHF. When a signaled exception condition occurs, LIB$SIG_TO_RET procedure does the following:

- Places the signaled condition value in the image of R0 that is saved as part of the mechanism argument vector.
- Calls the unwind system service (SYS$UNWIND) with the default arguments. After returning from LIB$SIG_TO_RET (when it is established as a condition handler) or after returning from the condition handler that called LIB$SIG_TO_RET (when LIB$SIG_TO_RET is called from within a condition handler), the stack is unwound to the caller of the procedure that established the handler.

Your calling procedure is then able to test R0 and R1 as if the called procedure had returned a status. Then the calling procedure can specify an error recovery action.

14.10. Changing a Signal to a Stop

The routine LIB$SIG_TO_STOP causes a signal to appear as though it had been signaled by a call to LIB$STOP.

LIB$SIG_TO_STOP can be established as a condition handler or called from within a user-written condition handler.

The argument that you passed to LIB$STOP is a 4-byte condition value (see Section 14.4.3). The argument must be passed using the %VAL argument-passing mechanism.

When a signal is generated by LIB$STOP, the severity code is forced to severe (STS$K_SEVERE) and control cannot be returned to the procedure that signaled the condition.

14.11. Checking for Arithmetic Exceptions

On OpenVMS I64 systems, arithmetic exceptions are indicated by specific condition codes. Examples are SS$_FLTDIV or SS$_INTOVR.

On OpenVMS Alpha systems, arithmetic exceptions are indicated by the condition code SS$_HPARITH. The signal array for arithmetic exceptions (condition code SS$_HPARITH) is

unique to OpenVMS Alpha systems and contains seven longwords (seven INTEGER (KIND=4) array elements):

- SIGARRAY(1) contains the argument count.
- SIGARRAY(2) contains the condition code SS$_HPARITH.
- SIGARRAY(3) contains the integer register write mask.
- SIGARRAY(4) contains the floating-point register write mask.
- SIGARRAY(5) contains the exception summary.
- SIGARRAY(6) contains the exception PC (program counter).
- SIGARRAY(7) contains the exception PS (processor status).

In the integer register write mask and the floating-point register write mask, where each bit represents a register, any bits set indicate the registers that were targets of the exception. The exception summary indicates the type of exception or exceptions in the first seven bits of that longword, as follows:

Bit	**Meaning**
0	Software completion.
1	Invalid floating arithmetic, conversion, or comparison.
2	Invalid attempt to perform a floating-point divide with a divisor of zero. (Integer divide-by-zero is not reported.)
3	Floating-point exponent overflow (arithmetic or conversion).
4	Floating-point exponent underflow (arithmetic or conversion).
5	Floating-point inexact result (arithmetic or conversion).
6	Integer arithmetic overflow or precision overflow during conversion from floating-point to integer.

To allow precise reporting of exceptions, specify the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier on the FORTRAN command line.

If you omit /SYNCHRONOUS_EXCEPTIONS, instructions beyond the instruction causing the exception may have been executed by the time the exception is reported. This causes the PC to be located at a subsequent instruction (inexact exception reporting). Specifying /SYNCHRONOUS_EXCEPTIONS drains the instruction pipeline after appropriate arithmetic instructions, but this slows performance.

For More Information:

- On the /CHECK qualifier, see Section 2.3.11.
- On the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier, see Section 2.3.46.

14.12. Checking for Data Alignment Traps

Although VSI Fortran naturally aligns local variables and provides the /ALIGNMENT qualifier to control alignment of fields in a record structures, derived-type structures, or a common block, certain conditions can result in unaligned data (see Section 5.3).

Unaligned data can result in a data alignment trap, which is an exception indicated by the condition code SS$_ALIGN. The OpenVMS operating system fixes up data alignment traps and does not report them to the program unless requested.

To obtain data alignment trap information, your program can call the SYS$START_ALIGN_FAULT_REPORT service.

You can also run the program within the OpenVMS debugger environment to detect the location of any unaligned data by using the SET BREAK/UNALIGNED debugger command (see Section 4.7).

Use the FORTRAN command /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier to ensure precise exception reporting.

Depending on the type of optimizations that might be applied, use the FORTRAN command /NOOPTIMIZE qualifier to ensure exception reporting.

For More Information:

- On alignment, see Section 5.3.
- On using the OpenVMS Debugger to detect unaligned data, see Section 4.7.
- On the SYS$START_ALIGN_FAULT_REPORT service, see the *VSI OpenVMS System Services Reference Manual*.
- On the /CHECK qualifier, see Section 2.3.11.
- On the /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier, see Section 2.3.46.

14.13. Condition Handler Example

The example in this section demonstrates the use of condition handlers in typical Fortran procedures.

The following example creates a function HANDLER that tests for integer overflow. The program needs to be compiled using the /CHECK=OVERFLOW, /SYNCHRONOUS_EXCEPTIONS (Alpha only), and (if fixed source form) /EXTEND_SOURCE qualifiers.

```
!   This program types a maximum value for a 4-byte integer and
!   then causes an exception with integer overflow, transferring
!   control to a condition handler named HANDLER.
!
!   Compile with: /CHECK=OVERFLOW
!                 /SYNCHRONOUS_EXCEPTIONS
!                 (and if fixed format, /EXTEND_SOURCE)
!
!   File: INT_OVR.F90
!
    INTEGER (KIND=4) INT4

    EXTERNAL HANDLER                                          ❶

    CALL LIB$ESTABLISH (HANDLER)                              ❷
    int4=2147483645
    WRITE (6,*) ' Beginning DO LOOP, adding 1 to ', int4
    DO I=1,10
       INT4=INT4+1                                            ❸
```

```
       WRITE (6,*) ' INT4 NUMBER IS  ', int4
    END DO
    WRITE (6,*) ' The end ...'
    END PROGRAM


!   The function HANDLER that handles the condition SS$_HPARITH

    INTEGER (KIND=4) FUNCTION HANDLER (SIGARGS, MECHARGS)    ❹

    INTEGER (KIND=4) SIGARGS(*),MECHARGS(*)                  ❺
    INCLUDE '($CHFDEF)'
    INCLUDE '($SSDEF)'
    RECORD /CHFDEF1/ SIGARGS
    RECORD /CHFDEF2/ MECHARGS
    INTEGER INDEX
    INTEGER LIB$MATCH_COND

    INDEX = LIB$MATCH_COND(SIGARGS(2), SS$_HPARITH)          ❻
    IF (INDEX .EQ. 0 ) THEN
        HANDLER = SS$_RESIGNAL
    ELSE IF (INDEX .GT. 0) THEN
        WRITE (6,*) '--> Arithmetic exception detected. Now in HANDLER'
        CALL LIB$STOP(%VAL(SIGARGS(2)),)                     ❼
    END IF

    RETURN
    END FUNCTION HANDLER
```

❶ The routine HANDLER is declared as EXTERNAL.
❷ The main program calls LIB$ESTABLISH to establish the condition handler named HANDLER.
❸ Within the DO loop, the value of variable INT4 is incremented. When the overflow occurs, control transfers to the condition handler.
❹ The condition handler (function HANDLER) is declared as an integer function accepting the signal array and mechanism array arguments.
❺ The library modules $SSDEF and $CHFDEF from FORSYSDEF.TLB are included and the SIGARGS and MECHARGS variables are declared as records. In this case, the structure definitions for the MECHARGS array are not needed, so the $CHFDEF library module is not included.
❻ The condition handler is only intended to handle the condition code SS$_HPARITH. For other conditions, SS$_RESIGNAL is returned, allowing the CHF to look for another condition handler.
❼ If the exception is SS$_HPARITH, the condition handler makes a call to LIB$STOP to:

- Display the %SYSTEM-F-HPARITH message.
- Display the traceback information. Note that the PC address is the address of the HANDLER routine. To obtain the approximate address where the exception occurred, the CALL LIB$ESTABLISH line can be commented out in the routine causing the exception and recompiled, relinked, and rerun.
- Stop program execution. This is not a continuable error.

The program is compiled, linked, and executed:

```
$ FORTRAN/CHECK=OVERFLOW/SYNCHRONOUS_EXCEPTIONS INT_OVR    ❶
$ LINK INT_OVR
$ RUN  INT_OVR
```

```
 Beginning DO LOOP, adding 1 to   2147483645
 INT4 NUMBER IS    2147483646
 INT4 NUMBER IS    2147483647                                    ❸
 -> Arithmetic exception detected. Now in HANDLER
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000000,
Fmask=0002023C, summary=1B, PC=00000001, PS=00000001
-SYSTEM-F-FLTINV, floating invalid operation, PC=00000001, PS=00000001
-SYSTEM-F-FLTOVF, arithmetic trap, floating overflow at PC=00000001,
PS=00000001
-SYSTEM-F-FLTUND, arithmetic trap, floating underflow at PC=00000001,
PS=00000001
TRACE-F-TRACEBACK, symbolic stack dump follows                 ❼
Image Name   Module Name    Routine Name  Line Number  rel PC      abs PC
 INT_OVR      INT_OVR$MAIN   HANDLER              1718 0000023C
  0002023C  DEC$FORRTL                                        0 000729F4
  000A49F4
----- above condition handler called with exception 00000504:
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000002,
Fmask=00000000, summary=40, PC=000200CC, PS=0000001B
-SYSTEM-F-INTOVF, arithmetic trap,integer overflow at PC=000200CC,
PS=0000001B
----- end of exception message
                                                          0 84C122BC
84C122BC
INT_OVR      INT_OVR$MAIN    INT_OVR$MAIN           19 000000CC
000200CC
                                                          0 84D140D0
84D140D0
```

❶ The program must be compiled using /CHECK=OVERFLOW.
Using /SYNCHRONOUS_EXCEPTIONS (Alpha only) allows more precise exception reporting. If the program is compiled as fixed-form file, specify the /EXTEND_SOURCE qualifier.

Chapter 15. Using the VSI Extended Math Library (VXML) (Alpha Only)

This chapter describes:

- Section 15.1: What Is VXML?
- Section 15.2: VXML Routine Groups
- Section 15.3: Using VXML from Fortran
- Section 15.4: VXML Program Example

Note

This entire chapter pertains only to VSI Fortran on OpenVMS Alpha systems only.

15.1. What Is VXML?

The VSI Extended Math Library (VXML) provides a comprehensive set of mathematical library routines callable from Fortran and other languages. VXML contains a set of over 1500 high-performance mathematical subprograms designed for use in many different types of scientific and engineering applications. It significantly improves the run-time performance of certain VSI Fortran programs.

VXML is included with VSI Fortran for OpenVMS Alpha Systems and can be installed using the instructions in the *VSI Fortran Installation Guide for OpenVMS Alpha Systems*.

The VXML reference guide is available in both online and hardcopy formats. You can obtain this documentation by accessing the following files:

- SYS$HELP:CXMLREF-VMS.PDF (online) – view with the Adobe Acrobat Reader
- SYS$HELP:CXMLREF-VMS.PS (hardcopy) – print to a PostScript printer

Example programs are also provided with VXML. These programs are located in the following directory:

SYS$COMMON:[SYSHLP.EXAMPLES.VXML]

15.2. VXML Routine Groups

VXML provides a comprehensive set of highly efficient mathematical subroutines designed for use in many different types of scientific and engineering applications. VXML includes the following functional groups of subroutines:

- Basic linear algebra
- Linear system and Eigenproblem solvers
- Sparse linear system solvers

- Signal processing
- Utility subprograms

The routines are described in Table 15.1.

Table 15.1. VXML 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 symmetrically shaped sparse matrices stored using the compressed row storage scheme. The iterative solver package supports 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 VXML routines are available in parallel form as well as serial form on VSI OpenVMS Alpha systems.

15.3. Using VXML from Fortran

To use VXML, you need to make the VXML routines and their interfaces available to your program and specify the appropriate libraries when linking.

The VXML routines can be called explicitly by your program. There are separate VXML libraries for the IEEE and the VAX floating-point formats. You must compile your program for one of these float formats and then link to the matching VXML library (either IEEE or VAX), depending upon how you compiled the program.

Either the IEEE or VAX VXML library can be established as the systemwide default by the system startup procedure. Individual users can select between the VAX and IEEE version by executing the SYS$LIBRARY:VXML$SET_LIB command procedure. For example, the following command alters the default VXML link library for the current user to the VAX format library:

```
$ @SYS$LIBRARY:VXML$SET_LIB VAX
```

For more details, see the section about VXML post-installation startup options in the *VSI Fortran Installation Guide for OpenVMS Alpha Systems*.

If needed, you can instead specify the appropriate VXML library or libraries on the LINK command line (use the /LIBRARY qualifier after each library file name). You must compile your program and then link to the appropriate VXML library (either IEEE or VAX), depending upon how you compiled the program. The following examples show the corresponding VXML commands for compiling and linking for cases where the VXML default library is not used:

```
$ FORTRAN /FLOAT=IEEE_FLOAT MYPROG.F90
$ LINK MYPROG.OBJ, SYS$LIBRARY:VXML$IMAGELIB_TS/LIBRARY
```

The link command uses the name of the VXML library for IEEE floating-point data, `cxml $imagelib_ts`. To use VAX floating-point data, specify the VXML library name as `cxml $imagelib_gs`.

If you are using an older version of VXML, use dxml$ *xxxxx* instead of cxml$ *xxxxx* as the library name. For more information on using VXML and specifying the correct object libraries on the LINK command, see the *Compaq Extended Mathematical Library Reference Manual*.

15.4. VXML Program Example

The free-form Fortran 90 example program below invokes the function SAXPY from the BLAS portion of the VXML libraries. The SAXPY function computes `a*x+y`.

```
   PROGRAM example
!
!  This free-form example demonstrates how to call
!  VXML 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
```

Appendix A. Differences Between VSI Fortran on OpenVMS I64 and OpenVMS Alpha Systems

This appendix describes:

- Section A.1: VSI Fortran Commands on OpenVMS I64 That Are Not Available on OpenVMS Alpha
- Section A.2: VSI Fortran Commands on OpenVMS Alpha That Are Not Available on OpenVMS I64
- Section A.3: Differences in Default Values
- Section A.4: Support for VAX-Format Floating-Point
- Section A.5: Changes in Exception Numbers and Places
- Section A.6: Changes in Exception-Mode Selection

A.1. VSI Fortran Commands on OpenVMS I64 That Are Not Available on OpenVMS Alpha

- /CHECK=ARG_INFO
- /CHECK=FP_MODE

A.2. VSI Fortran Commands on OpenVMS Alpha That Are Not Available on OpenVMS I64

- /ARCHITECTURE
- /MATH_LIBRARY
- /OLD_F77
- /OPTIMIZE=TUNE
- /SYNCHRONOUS_EXCEPTIONS

A.3. Differences in Default Values

The differences are:

- Because the Itanium architecture supports IEEE directly, the default floating-point format on OpenVMS I64 systems is /FLOAT=IEEE_FLOAT. On OpenVMS Alpha systems, the default is /FLOAT=G_FLOAT. See Section 2.3.22.

- The default floating-point exception handling mode on OpenVMS I64 systems is /IEEE_MODE=DENORM_RESULTS). On OpenVMS Alpha systems, the default is /IEEE_MODE=FAST. See Section 2.3.24.

A.4. Support for VAX-Format Floating-Point

Because there is no direct hardware support for VAX-format floating-point on OpenVMS I64 systems, the VAX-format floating-point formats (F, D, and G) are supported indirectly by a three-step process:

1. Conversion from VAX-format variable to IEEE floating-point temporary (using the denormalized number range)

2. Computation in IEEE floating-point

3. Conversion back to VAX-format floating-point and storage into the target variable

There are a number of implications for this approach:

- Exceptions might move, appear, or disappear, because the calculation is done in a different format with a different range of representable values.

- Because there are very small numbers representable in VAX format that can only be represented in IEEE format using the IEEE denorm range, there is a potential loss of accuracy if an intermediate result of the calculation is one of those numbers.

  At worst, the two bits with the least significance will be set to zero. Note that further calculations might result in magnifying the magnitude of this loss of accuracy.

  Note that this small loss of accuracy does not raise signal FOR$_SIGLOSMAT (or FOR$IOS_SIGLOSMAT).

- Expressions that are used to drive control flow but are not stored back into a variable will not be converted back into VAX format. This can cause exceptions to disappear.

- There can be a significant performance cost for the use of VAX-format floating-point.

Note that floating-point query built-ins (such as TINY and HUGE) will return values appropriate to the floating-point format that you select, despite the fact that all formats are supported by IEEE.

A.5. Changes in Exception Numbers and Places

There will be changes in the number of exceptions raised and in the code location at which they are raised.

This is particularly true for VAX-format floating-point calculations, because many exceptions will only be raised at the point where a result is converted from IEEE format to VAX format. Some valid IEEE-format numbers will be too large or too small to convert and will thus raise underflow or overflow. IEEE exceptional values (such as Infinity and NaN) produced during the evaluation of an expression will not generate exceptions until the final conversion to VAX format is done.

If a VAX-format floating-point calculation has intermediate results (such as the `X * Y` in the expression `(X * Y)/ Z`), and the calculation of that intermediate result raised an exception on OpenVMS Alpha

systems, it is not guaranteed that an exception will be raised on OpenVMS I64 systems. An exception will only be raised if the IEEE calculation produces an exception.

A.5.1. Ranges of Representable Values

In general, the range of VAX-format floating-point numbers is the same as the range for IEEE-format. However, the smallest F- or G-format value is one quarter the size of the smallest normal IEEE number, while the largest F- or G-format number is about half that of the largest IEEE number. There are therefore nonexceptional IEEE values that would raise overflows in F- or G-format. There are also nonexceptional F- or G-format values that would raise underflow in IEEE-format in those modes in which denormalized numbers are not supported.

A.5.2. Underflow in VAX Format with /CHECK=UNDERFLOW

OpenVMS Alpha and VAX Fortran applications do not report underflows for VAX-format floating-point operations unless you specifically enable underflow traps by compiling with the /CHECK=UNDERFLOW qualifier (see Section 2.3.11).

The same is true on OpenVMS I64 systems, but with an important caveat: Since all I64 floating-point operations are implemented by means of IEEE-format operations, enabling underflow traps with /CHECK=UNDERFLOW causes exceptions to be raised when values underflow the IEEE-format representation, not the VAX-format one.

This can result in an increased number of underflow exceptions seen with /CHECK=UNDERFLOW when compared with equivalent Alpha or VAX programs, as the computed values may be in the valid VAX-format range, but in the denormalized IEEE-format range.

If your application requires it, a user-written exception handler could catch the IEEE-format underflow exception, inspect the actual value, and determine whether it represented a VAX-format underflow or not.

See Section 8.4 for exact ranges of VAX-format and IEEE-format floating point.

A.6. Changes in Exception-Mode Selection

On OpenVMS Alpha systems, the exception-handling mode and the rounding mode can be chosen on a per-routine basis. This lets you set a desired exception mode and rounding mode using compiler qualifiers. Thus, a single application can have different modes during the execution of different routines.

This is not as easy to do on OpenVMS I64 systems. While the modes can be changed during the execution of a program, there is a significant performance penalty for doing so.

As a result, the VSI Fortran compiler and the OpenVMS linker implement a “whole-program” rule for exception handling and rounding modes. The rule says that the whole program is expected to run in the same mode, and that all compilations will have been done using the same mode qualifiers. To assist in enforcing this rule, the compiler, linker and loader work together:

- The compiler tags each compiled object file with a code specifying the modes selected by the user (directly or using the default) for that compilation.
- The linker tags the generated executable file with a code specifying the mode selected by the user.

- The loader initializes the floating-point status register of the process based on the linker code.

A.6.1. How to Change Exception-Handling or Rounding Mode

If you are using an OpenVMS I64 system and want to change the exception-handling or rounding mode during the execution of a program, use a call to either of the following:

- Fortran routines DFOR$GET_FPE and DFOR$SET_FPE
- System routines SYS$IEEE_SET_FP_CONTROL, SYS$IEEE_SET_PRECISION_MODE, and SYS$IEEE_SET_ROUNDING_MODE

Note

VSI does not encourage users to change the exception-handling or rounding mode within a program. This practice is particularly discouraged for an application using VAX-format floating-point.

A.6.1.1. Calling DFOR$GET_FPE and DFOR$SET_FPE

If you call DFOR$GET_FPE and DFOR$SET_FPE, you need to construct a mask using the literals in SYS$LIBRARY:FORDEF.FOR, module FOR_FPE_FLAGS.

The calling format is:

```
INTEGER*4  OLD_FLAGS, NEW_FLAGS
INTEGER*4  DFOR$GET_FPE, DFOR$GET_FPE

EXTERNAL   DFOR$GET_FPE, DFOR$GET_FPE

! Set the new mask, return old mask.
OLD_FLAGS = DFOR$GET_FPE( NEW_FLAGS )

! Return the current FPE mask.
OLD_FLAGS = DFOR$GET_FPE ()
```

An example (which does no actual computations) follows. For a more complete example, see Example A.1.

```
subroutine via_fortran
include 'sys$library:fordef.for'
include '($IEEEDEF)'
integer new_flags, old_flags
old_flags = dfor$get_fpe()
new_flags = FOR_K_FPE_CNT_UNDERFLOW + FPE_M_TRAP_UND
call dfor$set_fpe( new_flags )
!
!  Code here uses new flags
!
call dfor$set_fpe( old_flags )
return
end subroutine
```

A.6.1.2. Calling SYS$IEEE_SET_FP_CONTROL, SYS$IEEE_SET_PRECISION_MODE, and SYS$IEEE_SET_ROUNDING_MODE

If you call SYS$IEEE_SET_FP_CONTROL, SYS$IEEE_SET_PRECISION_MODE, and SYS$IEEE_SET_ROUNDING_MODE, you need to construct a set of masks using the literals in SYS$LIBRARY:FORSYSDEF.TLB, defined in module IEEEDEF.H (in STARLET). For information about the signature of these routines, see the *VSI OpenVMS System Services Reference Manual*.

An example (which does no actual computations) follows. For a more complete example, see Example A.1.

```
subroutine via_system( rr )
real rr
include '($IEEEDEF)'
integer*8 clear_flags, set_flags, old_flags, new_flags
clear_flags = IEEE$M_MAP_DNZ + IEEE$M_MAP_UMZ
set_flags   = IEEE$M_TRAP_ENABLE_UNF + IEEE$M_TRAP_ENABLE_DNOE
call sys$ieee_set_fp_control(%ref(clear_flags),
%ref(set_flags),%ref(old_flags))
!
!  Code here uses new flags
!
clear_flags = set_flags
call sys$ieee_set_fp_control(%ref(clear_flags),%ref(old_flags),
%ref(new_flags))
return
```

A.6.1.3. Additional Rules That You Should Follow

If you decide to change the exception-handling or rounding mode, be careful to observe the following rules to maintain the “whole-program” rule. Failure to do so might cause unexpected errors in other parts of your program:

- The preexisting mode must be restored at the end of the execution of the section in which the new mode is used. This includes both normal endings, such as leaving a code block, and exits by means of exception handlers.
- It is a good idea to establish a handler to restore the old mode on unwinds, because called code can cause exceptions to be raised (including exceptions not related to floating point).
- The code should be compiled with the same mode qualifiers as the other, “normal” parts of the application, not with the mode that will be set by the call to the special function.
- Be aware that VAX-format expressions are actually calculated in IEEE format, and any change to the modes will impact a calculation in IEEE format, not a calculation in VAX format.
- Consider adding the VOLATILE attribute to the declaration of all the variables used in the calculation in the different mode. This will prevent optimizations that might move all or part of the calculation out of the region in which the different mode is in effect.

A.6.1.4. Whole-Program Mode and Library Calls

System libraries that need to use an alternate mode (for example, the math library) accomplish this by using an architectural mechanism not available to user code: the `.sf1` flags of the floating-point status register (user code uses the `.sf0` flags).

Therefore, a user's choice of exception-handling or rounding mode will not have an impact on any system library used by the program.

A.6.2. Example of Changing Floating-Point Exception-Handling Mode

Example A.1 shows both methods of changing the floating-point exception-handling mode. However, for a real program, you should pick just one of the two methods.

Example A.1. Changing Floating-Point Exception Mode

```
!  SET_FPE.F90: Change floating-point exception handling mode,
!               and check that it worked.
!
!  Compile and link like this:
!
!  $ f90 set_fpe
!  $ lin set_fpe,SYS$LIBRARY:VMS$VOLATILE_PRIVATE_INTERFACES.OLB/lib
!  $ run set_fpe
!
!  The library is needed to bring in the code for LIB$I64_INS_DECODE,
!  which we call for its side-effect of incrementing the PC in the
!  correct manner for I64.
!

    !  This is a place to save the old FPE flags.
    !
    module saved_flags
    integer saved_old_flags
    end module

    !  Turn on underflow detection for one routine
    !  Using the FORTRAN library function FOR_SET_FPE.
    !
    subroutine via_fortran( rr )
    real rr
    include 'sys$library:fordef.for'
    include '($IEEEDEF)'
    integer new_flags, old_flags
    old_flags   = dfor$get_fpe()
    new_flags   = FPE_M_TRAP_UND
    call dfor$set_fpe( new_flags )
    !
    !  Code here uses new flags
    !
    rr = tiny(rr)
    type *,' Expect a catch #1'
    rr = rr / huge(rr)
    !
    call dfor$set_fpe( old_flags )
    return
    end subroutine

    !  Alternatively, do the same using the system routine.
    !
    subroutine via_system( rr )
    real rr
```

```
    include '($IEEEDEF)'
    integer*8 clear_flags, set_flags, old_flags, new_flags
    clear_flags = IEEE$M_MAP_DNZ + IEEE$M_MAP_UMZ
    set_flags   = IEEE$M_TRAP_ENABLE_UNF + IEEE$M_TRAP_ENABLE_DNOE
    call sys$ieee_set_fp_control(%ref(clear_flags), %ref(set_flags),
 %ref(old_flags))
    !
    !  Code here uses new flags
    !
    rr = tiny(rr)
    type *,' Expect a catch #2'
    rr = rr / huge(rr)
    !
    clear_flags = set_flags
    call sys$ieee_set_fp_control(%ref(clear_flags),%ref(old_flags),
%ref(new_flags))
    return
    end subroutine

    !  Main program
    !
    program tester
    use saved_flags
    real, volatile :: r
    !
    !  Establish an exception handler.
    !
    external handler
    call lib$establish( handler )
    !
    !  Save the initial setting of the exception mode flags.
    !
    saved_old_flags = dfor$get_fpe()
    !
    !  This expression underflows, but because this program has
    !  been compiled with /IEEE=DENORM (by default) underflows
    !  do not raise exceptions.
    !
    write (6,100)
100 format(1x,' No catch expected')
    r = tiny(r);
    r = r / huge(r)

    !
    !  Call a routine to turn on underflow and try that expression
    !  again.  After the call, verify that underflow detection has
    !  been turned off.
    !
    call via_fortran( r )
    write (6,100)
    r = tiny(r)
    r = r / huge(r)
    !
    !  Ditto for the other approach
    !
    call via_system( r )
    write (6,100)
    r = tiny(r)
```

```
r = r / huge(r)
end program


!  A handler is needed to catch any exceptions.
!
integer (kind=4) function handler( sigargs, mechargs )
use saved_flags
include '($CHFDEF)'
include '($SSDEF)'
integer sigargs(100)
record /CHFDEF2/ mechargs
integer  lib$match_cond
integer  LIB$I64_INS_DECODE
!
integer index
integer status
integer no_loop      / 20 /
logical int_over
logical int_div
logical float_over
logical float_div
logical float_under
logical float_inval
logical float_denorm
logical HP_arith
logical do_PC_advance
integer   pc_index
integer*8 pc_value
!
!  Don't loop forever between handler and exception
!  (in case something goes wrong).
!
no_loop = no_loop - 1
if( no_loop .le. 0 ) then
    handler = ss$_resignal
    return
endif
!
!  We'll need the PC value of the instruction if
!  this turns out to have been a fault rather than
!  a trap.
!
pc_index = sigargs(1)
pc_value = sigargs(pc_index)
!
!  Figure out what kind of exception we have, and
!  whether it is a fault and we need to advance the
!  PC before continuing.
!
do_PC_advance = .false.
int_over      = .false.
int_div       = .false.
float_over    = .false.
float_div     = .false.
float_under   = .false.
float_inval   = .false.
float_denorm  = .false.
```

```
HP_arith      = .false.
!
index = lib$match_cond(sigargs(2), SS$_INTOVF)
if( index .eq. 0 ) then
    int_over = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_INTDIV)
if( index .eq. 0 ) then
    int_div = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTOVF)
if( index .eq. 0 ) then
    float_over = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTDIV)
if( index .eq. 0 ) then
    float_div = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTUND)
if( index .eq. 0 ) then
    float_under = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTOVF_F)
if( index .eq. 0 ) then
    float_over = .true.
    do_PC_advance = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTDIV_F)
if( index .eq. 0 ) then
    float_div = .true.
    do_PC_advance = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTUND_F)
if( index .eq. 0 ) then
    float_under = .true.
    do_PC_advance = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTINV)
if( index .eq. 0 ) then
    float_inval = .true.
    do_PC_advance = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_INTOVF_F)
if( index .eq. 0 ) then
    int_over = .true.
    do_PC_advance = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_FLTDENORMAL)
```

```
if( index .eq. 0 ) then
    float_denorm = .true.
endif
!
index = lib$match_cond(sigargs(2), SS$_HPARITH)
if( index .eq. 0 ) then
    HP_arith = .true.
endif
!
!  Tell the user what kind of exception this is.
!
handler = ss$_continue
if( float_over) then
    write(6,*) ' - Caught Floating overflow'

else if ( int_over ) then
    write(6,*) ' - Caught Integer overflow'

else if ( int_div ) then
    write(6,*) ' - Caught Integer divide by zero'

else if ( float_div ) then
    write(6,*) ' - Caught Floating divide by zero'

else if ( float_under ) then
    write(6,*) ' - Caught Floating underflow'

else if ( float_inval ) then
    write(6,*) ' - Caught Floating invalid'

else if ( HP_arith ) then
    write(6,*) ' - Caught HP arith error'

else
    write(6,*) '  - Caught something else: resignal '
    !
    !  Here we have to restore the initial floating-point
    !  exception processing mode in case the exception
    !  happened during one of the times we'd changed it.
    !
    call dfor$set_fpe( saved_old_flags )
    !
    handler = ss$_resignal
endif
!
!  If this was a fault, and we want to continue, then
!  the PC has to be advanced over the faulting instruction.
!
if( do_PC_advance .and. (handler .eq. ss$_continue)) then
    status = lib$i64_ins_decode (pc_value)
    sigargs(pc_index) = pc_value
endif
!
return
end function handler
```

Appendix B. Compatibility: Compaq Fortran 77 and VSI Fortran

This appendix describes:

- Section B.1: VSI Fortran and Compaq Fortran 77 Compatibility on Various Platforms
- Section B.2: Major Language Features for Compatibility with Compaq Fortran 77 for OpenVMS Systems
- Section B.3: Language Features and Interpretation Differences Between Compaq Fortran 77 and VSI Fortran on OpenVMS Systems
- Section B.4: Improved VSI Fortran Compiler Diagnostic Detection
- Section B.5: Compiler Command-Line Differences
- Section B.6: Interoperability with Translated Shared Images
- Section B.7: Porting Compaq Fortran 77 for OpenVMS VAX Systems Data
- Section B.8: VAX H_float Representation

B.1. VSI Fortran and Compaq Fortran 77 Compatibility on Various Platforms

Table B.1 summarizes the compatibility of VSI Fortran (HF, which supports the Fortran 95, 90, 77, and 66 standards) and Compaq Fortran 77 (CF77) on multiple platforms (architecture/operating system pairs).

VSI Fortran is available on OpenVMS I64 systems, OpenVMS Alpha Systems, UNIX Alpha systems, Linux Alpha systems, and as VSI Visual Fortran on Windows systems.

Table B.1. Summary of Language Compatibility

Language Feature	HF UNIX Alpha	HF Linux Alpha	CVF Windows x86	HF OpenVMS I64 and Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Linking against static and shared libraries	•	•	•	•	•	•
Create code for shared libraries	•	•	•	•	•	•
Recursive code support	•	•	•	•	•	•
AUTOMATIC and STATIC statements	•	•	•	•	•	•
STRUCTURE and RECORD declarations	•	•	•	•	•	•
INTEGER*1, *2, *4	•	•	•	•	•	•
LOGICAL*1, *2, *4	•	•	•	•	•	•

Language Feature	HF UNIX Alpha	HF Linux Alpha	CVF Windows x86	HF OpenVMS I64 and Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
INTEGER*8 and LOGICAL*8	•	•		•	•	
REAL*4, *8	•	•	•	•	•	•
REAL*16[1]	•	•		•	•	•
COMPLEX*8, *16	•	•	•	•	•	•
COMPLEX*32 [2]	•	•		•	•	
POINTER (CRAY-style)	•	•	•	•	•	•
INCLUDE statements	•	•	•	•	•	•
IMPLICIT NONE statements	•	•	•	•	•	•
Data initialization in type declarations	•	•	•	•	•	•
Automatic arrays	•	•	•	•	•	
VOLATILE statements	•	•	•	•	•	•
NAMELIST-directed I/O	•	•	•	•	•	•
31-character names including $ and _	•	•	•	•	•	•
Source listing with machine code	•	•	•	•	•	•
Debug statements in source	•	•	•	•	•	•
Bit constants to initialize data and use in arithmetic	•	•	•	•	•	•
DO WHILE and END DO statements	•	•	•	•	•	•
Built-in functions %LOC, %REF, %VAL	•	•	•	•	•	•
SELECT CASE construct	•	•	•	•	•	
EXIT and CYCLE statements	•	•	•	•	•	
Variable FORMAT expressions (VFEs)	•	•	•	•	•	•
! marks end-of-line comment	•	•	•	•	•	•
Optional run-time bounds checking for arrays and substrings	•	•	•	•	•	•

Language Feature	HF UNIX Alpha	HF Linux Alpha	CVF Windows x86	HF OpenVMS I64 and Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Binary (unformatted) I/O in IEEE big endian, IEEE little endian, VAX, IBM, and CRAY floating-point formats	•	•	•	•	•	•
Fortran 90/95 standards checking	•	•	•	•		
FORTRAN-77 standards checking					•	•
IEEE exception handling	•	•	•	•	•	
VAX floating data type in memory				•	•	•
IEEE floating data type in memory	•	•	•	•	•	
CDD/Repository DICTIONARY support				•	•	•
KEYED access and INDEXED files				•	•	•
Parallel decomposition	•					•
OpenMP parallel directives	•					
Conditional compilation using IF … DEF constructs	•	•	•	•		
Vector code support						•
Direct inlining of Basic Linear Algebra Subroutines (BLAS)[3]		[3]	[3]	[3]	[3]	•
DATE_AND_TIME returns 4-digit year	•	•	•	•	•	•
FORALL statement and construct	•	•	•	•		
Automatic deallocation of ALLOCATABLE arrays	•	•	•	•		
Dim argument to MAXLOC and MINLOC	•	•	•	•		
PURE user-defined subprograms	•	•	•	•		
ELEMENTAL user-defined subprograms	•	•	•	•		

Language Feature	HF UNIX Alpha	HF Linux Alpha	CVF Windows x86	HF OpenVMS I64 and Alpha	CF77 OpenVMS Alpha	CF77 OpenVMS VAX
Pointer initialization (initial value)	•	•	•	•		
The NULL intrinsic to nullify a pointer	•	•	•	•		
Derived-type structure initialization	•	•	•	•		
CPU_TIME intrinsic subroutine	•	•	•	•		
Kind argument to CEILING and FLOOR intrinsics	•	•	•	•		
Nested WHERE constructs, masked ELSEWHERE statement, and named WHERE constructs	•	•	•	•		
Comments allowed in namelist input	•	•	•	•		
Generic identifier in END INTERFACE statements	•	•	•	•		
Minimal FORMAT edit descriptor field width	•	•	•	•		
Detection of obsolescent and/or deleted features[4]	•	•	•	•		

[1]For REAL*16 data, OpenVMS VAX systems use H_float format, and I64 and Alpha systems use IEEE style X_float format.

[2]For COMPLEX*32 data, I64 and Alpha systems use IEEE style X_float format for both REAL*16 parts.

[3]BLAS and other routines are available with the VSI Extended Mathematical Library (VXML). See Chapter 15.

[4] VSI Fortran flags these deleted and obsolescent features but fully supports them.

B.2. Major Language Features for Compatibility with Compaq Fortran 77 for OpenVMS Systems

To simplify porting Compaq Fortran 77 applications from OpenVMS VAX systems to VSI Fortran on OpenVMS I64 or OpenVMS Alpha systems, the following features (extensions) are provided with VSI Fortran:

- VSI Fortran provides LIB$ESTABLISH and LIB$REVERT as intrinsic functions for compatibility with Compaq Fortran 77 for OpenVMS VAX Systems condition handling; see Chapter 14.

- VSI Fortran provides FOR$RAB as an intrinsic function and it should not be declared as EXTERNAL; see Section 11.2.3

- FORSYSDEF parameter definitions for use with OpenVMS system services (see Appendix E).
- The /VMS qualifier (the default) makes the run-time environment behave more like Compaq Fortran 77 for OpenVMS VAX Systems (see Section 2.3.50).
- Compaq Fortran 77 extensions not part of the Fortran 90 standard that are supported as extensions by VSI Fortran for OpenVMS Alpha Systems include the following:
  - Record structures (STRUCTURE and RECORD statements)
  - Indexed sequential files, relative files, and keyed access
  - USEROPEN routines for RMS control block access
  - 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 (available on Alpha systems) and 16-byte REAL variables. REAL (KIND=16) data is provided in Alpha X_float format (not VAX H_float format)
  - Size specifiers for data declaration statements, such as INTEGER*4, in addition to the KIND type parameter
  - 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
  - Certain intrinsic functions
  - The tab source form (resembles fixed-source form)
  - I/O formatting descriptors

- Additional language features, including the DEFINE FILE, ENCODE, DECODE, and VIRTUAL statements

For More Information:

On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

B.3. Language Features and Interpretation Differences Between Compaq Fortran 77 and VSI Fortran on OpenVMS Systems

This section lists Compaq Fortran 77 extensions to the FORTRAN-77 standard that are interpretation differences or are not included in VSI Fortran for OpenVMS I64 or OpenVMS Alpha systems. Where appropriate, this list indicates equivalent VSI Fortran language features.

VSI Fortran conforms to the Fortran 90 and Fortran 95 standards. The Fortran 90 standard is a superset of the FORTRAN-77 standard. The Fortran 95 standard deletes some FORTRAN-77 features from the Fortran 90 standard. VSI Fortran fully supports all of these deleted features (see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).

VSI Fortran provides many but not all of the FORTRAN-77 extensions provided by Compaq Fortran 77.

B.3.1. Compaq Fortran 77 for OpenVMS Language Features Not Implemented

The following FORTRAN-77 extensions provided by Compaq Fortran 77 on OpenVMS systems (both Alpha and VAX hardware) are not provided by VSI Fortran for OpenVMS I64 or OpenVMS Alpha systems:

- Octal notation for integer constants is not part of the VSI Fortran language. Compaq Fortran 77 for OpenVMS Alpha Systems supports this feature only when the /VMS qualifier is in effect (default). For example:

```
I = "0014          ! Assigns 12 to I, not supported by VSI Fortran
```

- The VSI Fortran language prohibits dummy arguments with nonconstant bounds from being a namelist item. For example:

```
SUBROUTINE FOO(A,N)
  DIMENSION A(N),B(10)
  NAMELIST /N1/ A         ! Incorrect
  NAMELIST /N2/ B         ! Correct
END SUBROUTINE
```

- VSI 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
```

B.3.2. Compaq Fortran 77 for OpenVMS VAX Systems Language Features Not Implemented

Certain language features are available in Compaq Fortran 77 for OpenVMS VAX systems, but are not supported in VSI Fortran for OpenVMS I64 or OpenVMS Alpha systems. These features include features supported by the VAX architecture, VAX hardware support, and older language extensions:

- Automatic decomposition features of FORTRAN /PARALLEL= (AUTOMATIC). For information on a performance preprocessor that allows parallel execution, see Section 5.1.1.
- Manual (directed) decomposition features of FORTRAN /PARALLEL= (MANUAL) using the CPAR$ directives, such as CPAR$ DO_PARALLEL. For information on a performance preprocessor that allows parallel execution, see Section 5.1.1.
- The following I/O and error subroutines for PDP-11 compatibility:

ASSIGN	ERRTST	RAD50
CLOSE	FDBSET	R50ASC
ERRSET	IRAD50	USEREX

  When porting existing programs, calls to ASSIGN, CLOSE, and FBDSET should be replaced with the appropriate OPEN statement. (You might consider converting DEFINE FILE statements at the same time, even though VSI Fortran does support the DEFINE FILE statement).

  In place of ERRSET and ERRTST, OpenVMS condition handling might be used.

- Radix-50 constants in the form `nR xxx`

  For existing programs being ported, radix 50 constants and the IRAD50, RAD50 and R50ASC routines should be replaced by data encoded in ASCII using CHARACTER declared data.

- Numeric local variables are usually (but not always) initialized to a zero value, depending on the level of optimization used. To guarantee that a value will be initialized to zero under *all* circumstances, use an explicit assignment or DATA statement.
- Character constant actual arguments must be associated with character dummy arguments, not numeric dummy arguments, if source program units are compiled separately. (Compaq Fortran 77 for OpenVMS VAX Systems passed 'A' by reference if the dummy argument was numeric).

  To allow character constant actual arguments to be associated with numeric dummy arguments, specify the /BY_REF_CALL qualifier on the FORTRAN command line (see Section 2.3.9).

The following language features are available in Compaq Fortran 77 for OpenVMS VAX systems, but are not supported in VSI Fortran because of architectural differences between OpenVMS I64 and OpenVMS Alpha systems and OpenVMS VAX systems:

- Certain FORSYSDEF symbol definition library modules might be specific to the VAX or Itanium or Alpha architecture. For information on FORSYSDEF text library modules, see Appendix E.
- Precise exception control

  Compaq Fortran 77 for OpenVMS VAX Systems provides precise reporting of run-time exceptions. For performance reasons on OpenVMS I64 and OpenVMS Alpha systems, the default

FORTRAN command behavior is that exceptions are usually reported after the instruction causing the exception. You can request precise exception reporting using the FORTRAN command /SYNCHRONOUS_EXCEPTIONS (Alpha only) qualifier (see Section 2.3.46). For information on error and condition handling, see Chapter 7 and Chapter 14.

- The REAL*16 H_float data type supported on OpenVMS VAX systems

  The REAL (KIND=16) floating-point format on OpenVMS I64 and OpenVMS Alpha systems is X_float (see Chapter 8). For information on the VAX H_float data type, see Section B.8.

- VAX support for D_float, F_float, and G_float

  The OpenVMS Alpha instruction set does not support D_float computations, and the OpenVMS I64 instruction set does not support D_float, F_float or G_float computations. As a result, any data stored in those formats is converted to a native format for arithmetic computations and then converted back to its original format. On Alpha systems, the native format used for D_float is G_float. On I64 systems, S_float is used for F_float data, and T_float is used for D_float and G_float data.

  This means that for programs that perform many floating-point computations, using D_float data on Alpha systems is slower than using G_float or T_float data. Similarly, using D_float, F_float, or G_float data on I64 systems is slower than using S_float or T_float data. Additionally, due to the conversions involved, the results might differ from native VAX D_float, F_float, and G_float computations and results.

  Use the /FLOAT qualifier to specify the floating-point format (see Section 2.3.22).

  To create a VSI Fortran application program to convert D_float data to G_float or T_float format, use the file conversion methods described in Chapter 9.

- Vectorization capabilities

  Vectorization, including /VECTOR and its related qualifiers, and the CDEC$ INIT_DEP_FWD directive are not supported. The Alpha processor provides instruction pipelining and other features that resemble vectorization capabilities.

B.3.3. Compaq Fortran 77 for OpenVMS Language Interpretation Differences

The following FORTRAN-77 extensions provided by Compaq Fortran 77 on OpenVMS systems (both Alpha and VAX hardware) are interpreted differently by VSI Fortran.

- The VSI 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, whereas VSI Fortran evaluates the constant in single precision (by default).

  You can request that a single-precision constant assigned to a double-precision variable be evaluated in double precision, specify the FORTRAN command /ASSUME=FP_CONSTANT qualifier. The Fortran 90 standard requires that the constant be evaluated in single precision, but this can make calculated results differ between Compaq Fortran 77 and VSI Fortran.

In the example below, Compaq Fortran 77 assigns identical values to D1 and D2, whereas VSI 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 VSI 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, VSI Fortran uses the syntax for namelist external records specified by the Fortran 90 standard, rather than the Compaq Fortran 77 syntax (an extension to the FORTRAN-77 and Fortran 90 standards).

  Consider the following program:

```
INTEGER I  NAMELIST /N/ I  I = 5  PRINT N  END
```

  When this program is run after being compiled by the FORTRAN command, the following output appears:

```
$ FORTRAN TEST.F
$ LINK TEST
$ RUN TEST
&N I     =      5
/
```

  When this program is run after being compiled by the FORTRAN command with the /OLDF77 qualifier, the following output appears:

```
$ FORTRAN /OLDF77 TEST.F
$ LINK TEST
$ RUN TEST
$N I     =      5
$END
```

  VSI Fortran accepts Fortran 90 namelist syntax and Compaq Fortran 77 namelist syntax for reading records.

- VSI Fortran does not support C-style escape sequences in standard character literals. Use the C string literal syntax extension or the CHAR intrinsic instead. For example:

```
CHARACTER*2 CRLF
CRLF = '\r\n'         ! Incorrect
CRLF = '\r\n'C        ! Correct
CRLF = CHAR(13)//CHAR(10) ! Standard-conforming alternative
```

- VSI Fortran inserts a leading blank when doing list-directed I/O to an internal file. Compaq Fortran 77 does this only when the /VMS qualifier is in effect (default) on OpenVMS Alpha Systems. For example:

```
CHARACTER*10 C
WRITE(C,*) 'FOO'    ! C = ' FOO'
```

- Compaq Fortran 77 and VSI Fortran produce different output for 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)      ! HP Fortran prints 0.000
```

- VSI Fortran does not allow certain intrinsic procedures (such as SQRT) in constant expressions for array bounds. For example:

```
REAL A(SQRT(31.5))
END
```

- Compaq Fortran 77 returns UNKNOWN while VSI Fortran returns UNDEFINED when the ACCESS, BLANK, and FORM characteristics cannot be determined. For example:

```
INQUIRE(20,ACCESS=acc,BLANK=blk,FORM=form)
```

- VSI Fortran does not allow extraneous parentheses in I/O lists. For example:

```
write(*,*) ((i,i=1,1),(j,j=1,2))
```

- VSI Fortran does not allow control characters within quoted strings, unless you use the C-string extension. For example:

```
character*5 c c = 'ab
nef'    !  not allowed c = 'ab
nef'C   !  allowed end
```

- VSI 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.

  VSI 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 VSI Fortran:

```
REAL*8 R
R = 'abc' // 'def'
WRITE (6,*) R
END
```

  VSI Fortran does allow PARAMETER constants:

```
PARAMETER abcdef = 'abc' // 'def'
REAL*8 R
R = abcdef
WRITE (6,*) 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, VSI 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 VSI Fortran program.

For More Information:

- On argument passing between VSI Fortran and Compaq Fortran 77, see Section 10.9.
- On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

B.3.4. Compaq Fortran 77 for OpenVMS VAX Systems Interpretation Differences

The following language features are interpreted differently in Compaq Fortran 77 for OpenVMS VAX Systems and VSI Fortran for OpenVMS I64 or OpenVMS Alpha systems:

- Random number generator (RAN)

  The RAN function generates a different pattern of numbers in VSI Fortran than in Compaq Fortran 77 for OpenVMS VAX Systems for the same random seed. (The RAN and RANDU functions are provided for Compaq Fortran 77 for OpenVMS VAX Systems compatibility. See the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].)

- INQUIRE(RECL) for unformatted files

  INQUIRE(RECL) for unformatted files with the default RECL unit (longwords) gives different answers for Compaq Fortran 77 for OpenVMS VAX Systems and VSI Fortran if the

existing file has a record length that is not a multiple of 4 bytes. To prevent this difference, use /ASSUME=BYTERECL and specify the proper RECL in bytes in the OPEN statement.

- Hollerith constants in formatted I/O statements

  Compaq Fortran 77 for OpenVMS VAX Systems and VSI 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 value obtained by the READ statement is the output of the WRITE statement (FMT2 is ignored). However, in VSI Fortran, the output of the WRITE statement is "abcdefghij". (The value read by the READ statement has no effect on the value written by the WRITE statement).

B.4. Improved VSI Fortran Compiler Diagnostic Detection

The following language features are detected differently by VSI Fortran than Compaq Fortran 77:

- The VSI 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 VSI 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 VSI 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 VSI 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 VSI 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 VSI 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 VSI 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 VSI 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 VSI 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 VSI Fortran compiler detects the use of a format of the form Ew.dE0 at compile time. For example:

```
1   format(e16.8e0)     ! HP Fortran detects error at compile time
    write(*,1) 5.0      ! Compaq Fortran 77 compiles but an output
                        ! conversion error occurs at run time
```

- VSI Fortran detects passing of a statement function to a routine. For example:

```
foo(x) = x * 2
call bar(foo)
```

```
end
```

- The VSI Fortran compiler enforces the constraint that a branch to a statement shared by one more 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 VSI 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 VSI 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 VSI 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 from integer overflow
I8 = 2147483647_8 + 1_8 ! Correct
```

- The VSI 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 VSI Fortran compiler enforces the constraint that certain intrinsic procedures defined by the Fortran 95 standard cannot be passed as actual arguments. For example, Compaq Fortran 77 allows most intrinsic procedures to be passed as actual arguments, but the VSI Fortran compiler only allows those defined by the Fortran 95 standard (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 90 standard.

- VSI Fortran reports character truncation with an error-level message, not as a warning.

  The following program produces an error message during compilation with VSI 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 VSI Fortran intrinsic procedures with the wrong number of arguments or an incorrect argument type, VSI Fortran reports this with an error-level message, not with a warning. Possible causes include:

  - A VSI 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 VSI 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
```

- VSI 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 VSI 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)
```

- VSI 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).

- VSI Fortran does not detect an extra comma in an I/O statement when the /STANDARD qualifier is specified, whereas Compaq Fortran 77 with the same qualifier identifies an extra comma as an extension. For example:

```
WRITE(*,*) , P(J)
```

- VSI 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
```

- VSI 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)
```

- VSI 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 )
```

For More Information:

- On passing arguments and returning function values between VSI Fortran and Compaq Fortran 77, see Section 10.9.

- On VSI Fortran procedure calling and argument passing, see Section 10.2.

- On the VSI Fortran language, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

B.5. Compiler Command-Line Differences

This section summarizes the differences between VSI Fortran and Compaq Fortran 77 for OpenVMS Systems command lines.

The following commands initiate compilation on OpenVMS systems:

Platform	Language	Command
OpenVMS I64 systems	VSI Fortran	FORTRAN
OpenVMS Alpha systems	VSI Fortran	FORTRAN
OpenVMS Alpha systems	Compaq Fortran 77	FORTRAN/OLD_F77
OpenVMS VAX systems	Compaq Fortran 77	FORTRAN

Most qualifiers are the same between VSI Fortran for OpenVMS Alpha systems and Compaq Fortran 77 for OpenVMS Alpha systems.

B.5.1. Qualifiers Not Available on OpenVMS VAX Systems

Table B.2 lists VSI Fortran compiler qualifiers that have no equivalent Compaq Fortran 77 Version 6.4 for OpenVMS VAX Systems qualifiers.

Table B.2. VSI Fortran Qualifiers Without Equivalents in Compaq Fortran 77 for OpenVMS VAX Systems

Qualifier	Description
/ALIGNMENT= [NO]SEQUENCE	Specifies that components of derived types with the SEQUENCE attribute will obey whatever alignment rules are currently in use (see Section 2.3.3).
/ANNOTATION	Controls whether an annotated listing showing optimizations is included with the listing file (see Section 2.3.5).
/ARCHITECTURE= *keyword*	Specifies the type of Alpha architecture code instructions generated for a particular program unit being compiled (see Section 2.3.6).
/ASSUME=ALT_PARAM[1]	Allows the alternate syntax for PARAMETER statements. The alternate 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 (see Section 2.3.7).
/ASSUME=FP_CONSTANT[1]	Controls whether constants are evaluated in single or double precision (see Section 2.3.7). Compaq Fortran 77 always evaluates single-precision constants in double precision.
/ASSUME=MINUS0[1]	Controls whether the compiler uses Fortran 95 standard semantics for the IEEE floating-point value of -0.0 (minus zero) in the SIGN intrinsic, if the processor is capable of distinguishing the difference between -0.0 and +0.0 (see Section 2.3.7).
/ASSUME=PROTECT_ CONSTANTS	Specifies whether constant actual arguments can be changed (see Section 2.3.7).
/BY_REF_CALL	Allows character constant actual arguments to be associated with numeric dummy arguments (allowed by Compaq Fortran 77 for OpenVMS VAX Systems; see Section 2.3.9).

Qualifier	Description
/CHECK=ARG_INFO (I64 only)	Controls whether run-time checking of the actual argument list occurs (see Section 2.3.11).
/CHECK=ARG_TEMP_ CREATED	Issues a run-time warning message and continues execution if a temporary is created for an array actual argument (see Section 2.3.11).
/CHECK=FP_EXCEPTIONS	Controls whether messages about IEEE floating-point exceptional values are reported at run time (see Section 2.3.11).
/CHECK=FP_MODE (I64 only)	Controls whether run-time checking of the current state of the processor's floating-point status register (FPSR) occurs (see Section 2.3.11).
/CHECK=POWER	Controls whether the compiler evaluates and returns a result for certain arithmetic expressions containing floating- point numbers and exponentiation (see Section 2.3.11).
/DOUBLE_SIZE	Makes DOUBLE PRECISION declarations REAL (KIND=16) instead of REAL (KIND=8) (see Section 2.3.17).
/FAST	Sets several qualifiers that improve run-time performance (see Section 2.3.21).
/FLOAT	Controls the format used for floating-point data (REAL or COMPLEX) in memory, including the selection of either VAX F_float or IEEE S_float for KIND=4 data and VAX G_float, VAX D_float, or IEEE T_float for KIND=8 data (see Section 2.3.22). Compaq Fortran 77 for OpenVMS VAX Systems provides the /[NO]G_FLOAT qualifier.
/GRANULARITY	Controls the granularity of data access for shared data (see Section 2.3.23).
/IEEE_MODE	Controls how floating-point exceptions are handled for IEEE data (see Section 2.3.24).
/INTEGER_SIZE	Controls the size of INTEGER and LOGICAL declarations (see Section 2.3.26).
/MODULE	Controls where module files (.F90$MOD) are placed. If you omit this qualifier or specify /NOMODULE, the .F90$MOD files are placed in your current default directory (see Section 2.3.31).
/NAMES	Controls whether external names are converted to uppercase, lowercase, or as is (see Section 2.3.32).
/OPTIMIZE	Most keywords are not available on Compaq Fortran 77 for OpenVMS VAX Systems, including INLINE, LOOPS, PIPELINE, TUNE, and UNROLL (see Section 2.3.35).
/REAL_SIZE	Controls the size of REAL and COMPLEX declarations (see Section 2.3.37).
/REENTRANCY	Specifies whether code generated for the main program and Fortran procedures it calls will rely on threaded or asynchronous reentrancy (see Section 2.3.39).
/ROUNDING_MODE	Controls how floating-point calculations are rounded for IEEE data (see Section 2.3.40).
/SEPARATE_COMPILATION	Controls whether the VSI Fortran compiler:

Qualifier	Description
	• Places individual compilation units as separate modules in the object file like Compaq Fortran 77 for OpenVMS VAX Systems (/SEPARATE_COMPILATION) • Groups compilation units as a single module in the object file (/NOSEPARATE_COMPILATION, the default), which allows more interprocedure optimizations. For more information, see Section 2.3.41.
/SEVERITY[1]	Changes compiler diagnostic warning messages to have a severity of error instead of warning (see Section 2.3.42).
/SOURCE_FORM	Controls whether the compiler expects free or fixed form source (see Section 2.3.44).
/STANDARD	Flags extensions to the Fortran 90 or Fortran 95 standards (see Section 2.3.45).
/SYNTAX_ONLY	Requests that only syntax checking occurs and no object file is created (see Section 2.3.47).
/WARNINGS	Certain keywords are not available on Compaq Fortran 77 for OpenVMS VAX Systems (see Section 2.3.51).
/VMS	Requests that VSI Fortran use certain Compaq Fortran 77 for OpenVMS VAX Systems conventions (see Section 2.3.50).

[1]This qualifier applies only to VSI Fortran and does not apply to Compaq Fortran 77 on any platform.

B.5.2. Qualifiers Specific to Compaq Fortran 77 for OpenVMS VAX Systems

This section summarizes Compaq Fortran 77 for OpenVMS VAX Systems compiler options that have no equivalent VSI Fortran options.

Table B.3 lists compilation options that are specific to Compaq Fortran 77 for OpenVMS VAX Systems Version 6.4.

Table B.3. Compaq Fortran 77 for OpenVMS VAX Systems Options Not in VSI Fortran

Compaq Fortran 77 for OpenVMS VAX Systems Qualifier	Description
/BLAS=(INLINE, MAPPED)	Specifies whether Compaq Fortran 77 for OpenVMS VAX Systems recognizes and inlines or maps the Basic Linear Algebra Subroutines (BLAS).
/CHECK=ASSERTIONS	Enables or disables assertion checking.
/DESIGN=[NO]COMMENTS /DESIGN= [NO]PLACEHOLDERS	Analyzes program for design information.
/DIRECTIVES=DEPENDENCE	Specifies whether specified compiler directives are used at compilation.

Compaq Fortran 77 for OpenVMS VAX Systems Qualifier	Description
/PARALLEL=(MANUAL or AUTOMATIC)	Supports parallel processing.
/SHOW=(DATA_ DEPENDENCIES, LOOPS)	Control whether the listing file includes: • Diagnostics about loops that are ineligible for dependence analysis and data dependencies that inhibit vectorization or autodecomposition (DATA_DEPENDENCIES) • Reports about loop structures after compilation (LOOPS)
/VECTOR	Requests vector processing.
/WARNINGS=INLINE	Controls whether the compiler prints informational diagnostic messages when it is unable to generate inline code for a reference to an intrinsic routine. Other /WARNINGS keywords are only available with Compaq Fortran 77 for OpenVMS VAX Systems, including TRUNCATED_SOURCE.

All CPAR$ directives and certain CDEC$ directives associated with directed (manual) decomposition and their associated qualifiers or keywords are also specific to Compaq Fortran 77 for OpenVMS VAX Systems.

For More Information:

- On the FORTRAN command and qualifiers, see Chapter 2.
- On the Compaq Fortran 77 compilation commands and options, see the appropriate Compaq Fortran 77 user manual.

B.6. Interoperability with Translated Shared Images

VSI Fortran provides the ability to interoperate with translated shared images. That is, when creating a native VSI Fortran image, you can add certain qualifiers to the FORTRAN and LINK command lines to allow the resulting image to interoperate with translated shared images at image activation (run time).

To allow the use of translated shared images:

- On the FORTRAN command line, specify the /TIE qualifier.
- On the LINK command line, specify the /NONATIVE_ONLY qualifier (default).

The created executable image contains code that allows the resulting executable image to interoperate with shared (installed) images, including allowing the Compaq Fortran 77 for OpenVMS VAX Systems RTL (FORRTL_TV) to work with the VSI Fortran RTL (DEC$FORRTL).

For More Information:

On porting, see *Migrating an Application from OpenVMS VAX to OpenVMS Alpha*.

B.7. Porting Compaq Fortran 77 for OpenVMS VAX Systems Data

Record types are identical for Compaq Fortran 77 on OpenVMS VAX Systems and VSI Fortran on OpenVMS I64 or OpenVMS Alpha systems.

If you need to convert unformatted floating-point data, keep in mind that Compaq Fortran 77 for OpenVMS VAX Systems programs (VAX hardware) stores:

- REAL*4 or COMPLEX*8 data in F_float format
- REAL*8 or COMPLEX*16 data in either D_float or G_float format
- REAL*16 data in H_float format.

VSI Fortran programs (running on OpenVMS I64 or OpenVMS Alpha systems) store REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 data in one of the formats shown in Table B.4 and REAL*16 data in X_float format.

Table B.4. Floating-Point Data Formats on OpenVMS VAX and OpenVMS I64 and Alpha Systems

Data Declaration	OpenVMS VAX Formats	OpenVMS I64 and Alpha Formats
REAL*4 and COMPLEX*8	VAX F_float	VAX F_float or IEEE S_float
REAL*8 and COMPLEX*16	VAX D_float or VAX G_float	VAX D_float[1], VAX G_float, or IEEE T_float
REAL*16	VAX H_float	X_float format. You can convert VAX H_float REAL*16 data to Alpha X_float format.

[1]The D_float format on I64 and Alpha systems has less precision during computations than on VAX systems.

The floating-point data types supported by VSI Fortran on OpenVMS systems are described in Chapter 8.

Example B.1 shows the use of the CVT$CONVERT_FLOAT RTL routine to convert VAX S_float data to VAX F_float format. This allows the converted value to be used as an argument to the LIB$WAIT routine.

The parameter definitions used in the CVT$CONVERT_FLOAT argument list (such as CVT$K_IEEE_S) are included from the $CVTDEF library module in FORSYSDEF. The S_float value read as an argument is contained in the variable F_IN; the F_float value is returned by CVT$CONVERT_FLOAT to variable F_OUT.

Example B.1. Using the CVT$CONVERT_FLOAT Routine

```
! This program converts IEEE S_float data to VAX F_float format
!
! Compile with:   $  F90/FLOAT=IEEE_FLOAT
!
    PROGRAM CONVERT
```

```
    INCLUDE '($CVTDEF)'
    REAL(KIND=4)  F_IN
    REAL(KIND=4)  F_OUT
    INTEGER ISTAT

    F_IN = 20.0

    PRINT *,' Sample S_float input value is ', F_IN

    ISTAT=CVT$CONVERT_FLOAT(F_IN, %VAL(CVT$K_IEEE_S), F_OUT, &
              %VAL(CVT$K_VAX_F), %VAL(CVT$M_ROUND_TO_NEAREST))

    PRINT *,'Return status ISTAT',ISTAT

!   IF (.NOT. ISTAT) CALL LIB$SIGNAL(%VAL(ISTAT))

    PRINT *, ' Waiting for specified time '

    CALL LIB$WAIT (F_OUT)
    STOP
    END PROGRAM CONVERT
```

B.8. VAX H_float Representation

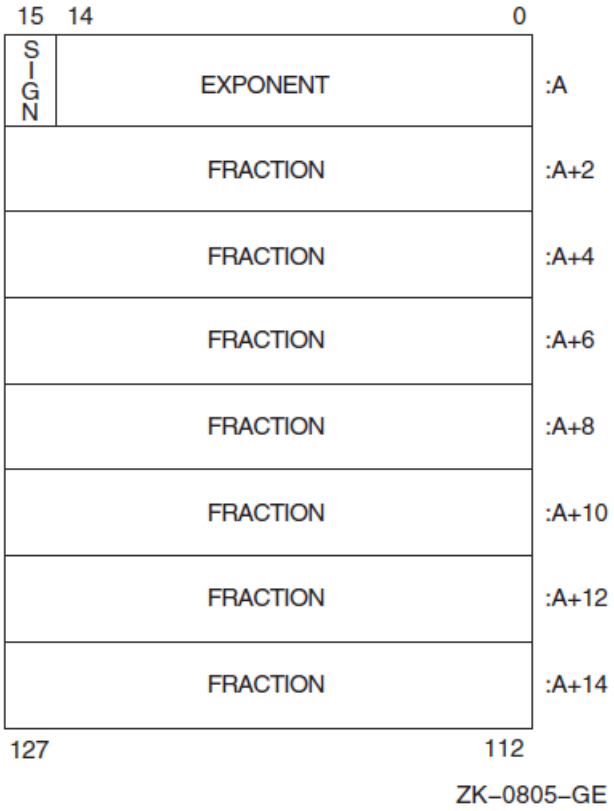
This section describes the REAL*16 VAX H_float data formats used on OpenVMS VAX systems. On OpenVMS I64 and OpenVMS Alpha systems, REAL*16 (extended precision) data is always stored in IEEE X_float format.

With VAX floating-point data types, the binary radix point is to the left of the most-significant bit.

The REAL*16 H_float format is available only on OpenVMS VAX systems; REAL*16 on OpenVMS I64 and OpenVMS Alpha systems use X_float format (see Section 8.4.4).

As shown in Figure B.1, 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.

Figure B.1. VAX H_float REAL*16 Representation (VAX Systems)



The form of a REAL*16 (H_float) 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.

Appendix C. Diagnostic Messages

This appendix describes:

- Section C.1: Overview of Diagnostic Messages
- Section C.2: Diagnostic Messages from the VSI Fortran Compiler
- Section C.3: Messages from the VSI Fortran Run-Time System

C.1. Overview of Diagnostic Messages

Diagnostic messages related to a VSI Fortran program can come from the compiler, the linker, or the VSI Fortran run-time system:

- The VSI Fortran compiler detects syntax errors in the source program, such as unmatched parentheses, invalid characters, misspelled specifiers, and missing or invalid parameters. VSI Fortran compiler messages are described in Section C.2.
- The linker detects errors in object file format and source program errors such as undefined symbols. Linker messages are described in the *OpenVMS System Messages and Recovery Procedures Reference Manual*. Or you can use the DCL command HELP/MESSAGE.
- The VSI Fortran run-time system reports errors that occur during program execution. VSI Fortran run-time messages are listed and described in Section C.3 and DCL HELP (enter HELP FORTRAN).

These messages are displayed on your terminal or in your log file. The format of the messages is:

```
%facility-severity-mnemonic, message_text
```

The contents of the fields of information in diagnostic messages are as follows:

%	The percent sign identifies the line as a message.
`facility`	A 2-, 3-, or 4-letter facility code that identifies the origin of the message, whether it came from the compiler (F90), the linker (LINK), or the run-time system (FOR, SS, or MTH).
`severity`	A single character that determines message severity. The four levels of error message severity are: Fatal (F), Error (E), Warning (W), and Informational (I). The definition of each severity level depends on the facility issuing the message.
`mnemonic`	A 6- to 9-character name that uniquely identifies that message.
`message_text`	Explains the event that caused the message to be generated.

C.2. Diagnostic Messages from the VSI Fortran Compiler

A diagnostic message issued by the compiler describes the detected error and, in some cases, contains an indication of the action taken by the compiler in response to the error.

Besides reporting errors detected in source program syntax, the compiler issues messages indicating errors that involve the compiler itself, such as I/O errors.

C.2.1. Source Program Diagnostic Messages

The severity level of source program diagnostic messages, in order of greatest to least severity, are as follows:

Severity Code	Description
F	Fatal; must be corrected before the program can be compiled. No object file is produced if an F-severity error is detected during compilation.
E	Error; should be corrected. No object file is produced if an F-severity error is detected during compilation.
W	Warning; should be investigated by checking the statements to which W-severity diagnostic messages apply. Warnings are issued for statements that use acceptable, but nonstandard, syntax and for statements corrected by the compiler. An object file is produced, but the program results may be incorrect. Note that W-severity messages are produced unless the /NOWARNINGS qualifier is specified in the FORTRAN command.
I	Information; not an error message and does not call for corrective action. However, the I-severity message informs you that either a correct VSI Fortran statement may have unexpected results or you have used a VSI Fortran extension to the Fortran 90 or Fortran 95 standard.

Typing mistakes are a likely cause of syntax errors; they can cause the compiler to generate misleading diagnostic messages. Beware especially of the following:

- Missing comma or parenthesis in a complicated expression or FORMAT statement.
- Misspelled variable names or (depending on the compilation qualifiers used) case-mismatched variable names. The compiler may not detect this error, so execution can be affected.
- Inadvertent line continuation mark. This can cause a diagnostic message for the preceding line.
- When compiling using fixed-form source files, if the statement line extends past column 72, this can cause the statement to terminate early (unless you specified the /EXTEND_SOURCE qualifier).
- Confusion between the digit 0 and the uppercase letter O. This can result in variable names that appear identical to you but not to the compiler.
- Nonprinting ASCII characters encountered in VSI Fortran programs are generally interpreted as a space character and a warning message appears. For more information on valid nonprinting space characters, see Section 2.3.28.

Because a diagnostic message indicates only the immediate cause, you should always check the entire source statement carefully.

The following examples show how source program diagnostic messages are displayed in interactive mode on your screen.

```
  40    FORMAT (I3,)
...................^
%F90-E-ERROR, An extra comma appears in the format list.
at line number 13 in file DISK$:[USER]SAMP_MESS.FOR;4

        GO TO 66
```

```
..............^
%F90-E-ERROR, This label is undefined.   [66]
at line number 18 in file DISK$:[USER]SAMP_MESS.FOR;4
```

Example C.1 shows how these messages appear in listings.

Example C.1. Sample Diagnostic Messages (Listing Format)

```
MORTGAGE                    30-MAR-1995 14:19:21   HP Fortran Xn.n-xxx
 Page  1
                            30-MAR-1995 14:18:48   DISK$:
[USER]SAMP_MESS.F90;1

      1 !     Program to calculate monthly mortgage payments
      2
      3       PROGRAM MORTGAGE
      4
      5       TYPE 10
      6   10  FORMAT (' ENTER AMOUNT OF MORTGAGE ')
      7       ACCEPT 20, IPV
      8   20  FORMAT (I6)
      9
     10       TYPE 30
     11   30  FORMAT (' ENTER LENGTH OF MORTGAGE IN MONTHS ')
     12       ACCEPT 40, IMON
     13   40  FORMAT (I3,)
        ................1
%F90-E-ERROR, An extra comma appears in the format list.
at line number 13 in file DISK$:[USER]SAMP_MESS.F90;1
     14       TYPE 50
     15   50  FORMAT (' ENTER ANNUAL INTEREST RATE ')
     16       ACCEPT 60, YINT
     17   60  FORMAT (F6.4)
     18       GO TO 66
        ............1
%F90-E-ERROR, This label is undefined.   [66]
at line number 18 in file DISK$:[USER]SAMP_MESS.F90;1
     19   65  YI = YINT/12     ! Get monthly rate
     20       IMON = -IMON
     21       FIPV = IPV * YI
     22       YI = YI + 1
     23       FIMON = YI**IMON
     24       FIMON = 1 - FIMON
     25       FMNTHLY = FIPV/FIMON
     26
     27       TYPE 70, FMNTHLY
     28   70  FORMAT (' MONTHLY PAYMENT EQUALS ',F7.3 )
     29       STOP
     30       END PROGRAM MORTGAGE
```

C.2.2. Compiler-Fatal Diagnostic Messages

Conditions can be encountered of such severity that compilation must be terminated at once. These conditions are caused by hardware errors, software errors, and errors that require changing the FORTRAN command. Printed messages have the form:

```
%F90-F-MNEMONIC, error_text
```

The first line of the message contains the appropriate file specification or keyword involved in the error. The operating system supplies more specific information about the error whenever possible. For example, a file read error might produce the following error message:

```
%F90-F-ERROR, error reading _DBA0:[SMITH]MAIN.FOR;3
-RMS-W-RTB, 512 byte record too big for user's buffer
-F90-F-ABORT, abort
```

The secondary operating system (in this case the RMS facility) message provides helpful information about the actual cause of the error, as described in the *OpenVMS System Messages and Recovery Procedures Reference Manual* or the equivalent OpenVMS HELP/MESSAGE facility.

C.3. Messages from the VSI Fortran Run-Time System

Errors that occur during execution of your VSI Fortran program are reported by diagnostic messages from the VSI Fortran Run-Time Library (RTL). These messages may result from hardware conditions, file system errors, errors detected by RMS, errors that occur during transfer of data between the program and an internal record, computations that cause overflow or underflow, incorrect calls to the VSI Fortran RTL, problems in array descriptions, and conditions detected by the operating system.

As described in Section 7.2, the severity of run-time diagnostic messages can be fatal (F), error (E), warning (W), and informational (I).

The following example shows how run-time messages appear:

```
%FOR-F-ADJARRDIM, adjustable array dimension error
```

Table C.1 is an alphabetical list of run-time diagnostic messages, without the message prefixes FOR, SS, and MTH. (Refer to Table 7.1 for a list of the messages in error-number sequence.) For each message, Table C.1 gives a mnemonic, the message number, the severity of the message, the message text, and an explanation of the message.

You can also view a description of specified messages using the following command:

```
$ HELP FORTRAN RUN_TIME
```

This displays the mnemonics of all VSI Fortran run-time diagnostic messages. You can abbreviate the HELP command words and specify a specific error mnemonic, such as:

```
$ HELP FORTRAN RUN FILNOTFOU
```

Table C.1. Run-Time Error Messages and Explanations

Mnemonic	Number	Severity Code, Message Text, and Explanation
ADJARRDIM	93	F, adjustable array dimension error Upon entry to a subprogram, one of the following errors was detected during the evaluation of dimensioning information: • An upper-dimension bound was less than a lower-dimension bound.

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• The dimensions implied an array that was larger than addressable memory.
ARRSIZEOVF	179	F, Cannot allocate array — overflow on array size calculation An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory.
ASSERTERR	145	F, assertion error The VSI Fortran RTL encountered an assertion error. Please report the problem to VSI.
ATTACCNON	36	F, attempt to access non-existent record One of the following conditions occurred: • A direct access READ, FIND, or DELETE statement attempted to access a nonexistent record from a relative organization file. • A direct access READ or FIND statement attempted to access beyond the end of a sequential organization file. • A keyed access READ statement attempted to access a nonexistent record from an indexed organization file.
BACERR	23	F, BACKSPACE error During execution of a BACKSPACE statement, One of the following conditions occurred: • The file was not a sequential organization file. • The file was not opened for sequential access. (A unit opened for append access must not be backspaced until a REWIND statement is executed for that unit.) • RMS (Record Management Services) detected an error condition during execution of a BACKSPACE statement.
BUG_CHECK	8	F, internal consistency check failure Internal severe error. Please check that the program is correct. Recompile if an error existed in the program. If this error persists, report the problem to VSI.
CLOERR	28	F, CLOSE error An error condition was detected by RMS during execution of a CLOSE statement.
DELERR	55	F, DELETE error During execution of a DELETE statement, one of the following conditions occurred:

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• On a direct access DELETE, the file that did not have relative organization. • On a current record DELETE, the file did not have relative or indexed organization, or the file was opened for direct access. • RMS detected an error condition during execution of a DELETE statement.
DIRIO_KEY	258	F, direct-access I/O to unit open for keyed access The OPEN statement for this unit number specified keyed access and the I/O statement specifies direct access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access. (For more information on statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).
DIV	178	F, divide by zero A floating point or integer divide by zero exception occurred.
DUPFILSPE	21	F, duplicate file specifications 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 by an OPEN statement.
ENDDURREA	24	F, end-of-file during read One of the following conditions occurred: • An RMS 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.
ENDFILERR	33	F, ENDFILE error 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.

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• RMS detected an error during execution of an ENDFILE statement.
ENDRECDUR	268	F, end of record during read An end-of-record condition was encountered during execution of a nonadvancing I/O READ statement that did not specify the EOR branch specifier.
ERRDURREA	39	F, error during read RMS detected an error condition during execution of a READ statement.
ERRDURWRI	38	F, error during write RMS detected an error condition during execution of a WRITE statement.
FILNAMSPE	43	F, file name specification error A file-name specification given to an OPEN or INQUIRE statement was not acceptable to RMS.
FILNOTFOU	29	F, file not found A file with the specified name could not be found during an open operation.
FINERR	57	F, FIND error RMS detected an error condition during execution of a FIND statement.
FLOCONFAI	95	E, floating point conversion failed The attempted unformatted read or write of nonnative floating-point data failed. One of the following conditions occurred for a nonnative floating-point value: • When using VAX data types in memory (/FLOAT=G_FLOAT or /FLOAT=D_FLOAT), the value exceeded the allowable maximum value for the equivalent native format and was set equal to invalid. • When using VAX data types in memory (/FLOAT=G_FLOAT or /FLOAT=D_FLOAT), the value was infinity (plus or minus), Not-a-Number (NaN), or otherwise invalid and was set to invalid. • When using IEEE data types in memory (/FLOAT=IEEE_FLOAT), the value exceeded the allowable maximum value for the equivalent native format and was set equal to infinity (plus or minus).

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• When using IEEE data types in memory (/FLOAT=IEEE_FLOAT), the value was infinity (plus or minus) and was set to infinity (plus or minus). • When using IEEE data types in memory (/FLOAT=IEEE_FLOAT), the value was invalid and was set to Not-a-Number (NaN). Make sure the correct file was specified. Make sure the record layout matches the format VSI Fortran is expecting. Check that the correct nonnative floating-point data format was specified, as described in Chapter 9.
FLODIV0EXC	299	I, `nn` divide-by-zero traps The total number of floating-point divide-by-zero traps encountered during program execution was `nn`. This summary message appears at program completion.
FLOINCEXC	297	I, `nn` floating invalid traps The total number of floating-point invalid data traps encountered during program execution was `nn`. This summary message appears at program completion.
FLOINEEXC	296	I, `nn` floating inexact traps The total number of floating-point inexact data traps encountered during program execution was `nn`. This summary message appears at program completion.
FLOOVEMAT	88	F, floating overflow in math library A floating overflow condition was detected during execution of a math library procedure.
FLOOVREXC	298	I, `nn` floating overflow traps The total number of floating-point overflow traps encountered during program execution was `nn`. This summary message appears at program completion.
FLOUNDEXC	300	I, `nn` floating underflow traps The total number of floating-point underflow traps encountered during program execution was `nn`. This summary message appears at program completion.
FLOUNDMAT	89	E, floating underflow in math library A floating underflow condition was detected during execution of a math library procedure. The result returned was zero.
FLTDIV	73	E, zero divide During a floating-point or decimal arithmetic operation, an attempt was made to divide by 0.0.
FLTINE	140	E, floating inexact

Mnemonic	Number	Severity Code, Message Text, and Explanation
		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.
FLTINV	65	E, floating invalid 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 qualifier can suppress this message (see Section 2.3.11). For information on allowing exceptional IEEE values, see Section 2.3.24.
FLTOVF	72	E, arithmetic trap, floating overflow During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type. The result returned was the reserved operand, –0 for VAX-format floating-point and NaN for IEEE-format floating-point.
FLTUND	74	E, floating underflow During an arithmetic operation, a floating-point value became less than the smallest finite value for that data type and was replaced with a value of zero. When using the /FLOAT=IEEE_FLOAT, depending on the values of the /IEEE_MODE qualifier, the underflowed result was either set to zero or allowed to gradually underflow. See Chapter 8 for ranges of the various data types.
FMTIO_UNF	257	F, formatted I/O to unit open for unformatted transfers 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 specified in the OPEN statement and the file should contain formatted data, specify FORM='FORMATTED' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.
FMYSYN	58	I, format syntax error at or near `xxx` 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 *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].
FORVARMIS	61	F or I, format/variable-type mismatch 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). If /

Mnemonic	Number	Severity Code, Message Text, and Explanation
		CHECK=NOFORMAT is in effect (see Section 2.3.11), the severity is I (program continues). If execution continues, the following actions occurred: • If I, L, O, Z, B, conversion as if INTEGER (KIND=4). • If D, E, or F, conversion as if REAL (KIND=4). To suppress this error message, see Section 2.3.11.
INCFILORG	51	F, inconsistent file organization One of the following conditions occurred: • The file organization specified in an OPEN statement did not match the organization of the existing file. • The file organization of the existing file was inconsistent with the specified access mode; that is, direct access was specified with an indexed organization file, or keyed access was specified with a sequential or relative organization file.
INCKEYCHG	50	F, inconsistent key change or duplicate key A WRITE or REWRITE statement accessing an indexed organization file caused a key field to change or be duplicated. This condition was not allowed by the attributes of the file, as established when the file was created.
INCOPECLO	46	F, inconsistent OPEN/CLOSE parameters Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow: • ACTION= 'READ' or READONLY with STATUS= 'NEW' or STATUS= 'SCRATCH' • ACCESS= 'APPEND' with READONLY, ACTION= 'READ', STATUS= 'NEW', or STATUS= 'SCRATCH' • DISPOSE='DELETE' with READONLY or ACTION='READ' • ACCESS= 'APPEND' with STATUS= 'REPLACE' • ACCESS= 'DIRECT' or 'KEYED' with POSITION= 'APPEND', 'ASIS', or 'REWIND'
INCRECLEN	37	F, inconsistent record length One of the following occurred: • An attempt was made to create a new relative, indexed, or direct access file without specifying a record length.

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• An existing file was opened in which the record length did not match the record size given in an OPEN or DEFINE FILE statement. • An attempt was made to write to a relative, indexed, or direct access file that was not correctly created or opened.
INCRECTYP	44	F, inconsistent record type The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.
INFFORLOO	60	F, infinite format loop The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
INPCONERR	64	F, input conversion error 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.
INPRECTOO	22	F, input record too long 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 of the appropriate size.
INPSTAREQ	67	F, input statement requires too much data 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'.
INSVIRMEM	41	F, insufficient virtual memory The VSI Fortran Run-Time Library attempted to exceed its virtual page limit while dynamically allocating space. Inform your system manager that process quotas and/or system parameters need to be increased. For information about virtual memory use, see Section 1.2 and the *VSI Fortran Installation Guide for OpenVMS I64 Systems* or *VSI Fortran Installation Guide for OpenVMS Alpha Systems*.
INTDIV	71	F, integer zero divide 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 one (1).
INTOVF	70	F, integer overflow During an arithmetic operation, an integer value exceeded its range. The result of the operation was the correct low-order part. Consider specifying a larger integer data size (or use the /INTEGER_SIZE

Mnemonic	Number	Severity Code, Message Text, and Explanation
		qualifier for INTEGER declarations without a kind or specifier.) See Chapter 8 for ranges of the various integer data types.
INVARGFOR	48	F, invalid argument to Fortran Run-Time Library The VSI Fortran compiler passed an invalid coded argument to the Run-Time Library. This can occur if the compiler is newer than the VSI Fortran RTL in use.
INVARGMAT	81	F, invalid argument to math library One of the mathematical procedures detected an invalid argument value.
INVDEALLOC	153	F, allocatable array is not allocated A Fortran 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.
INVDEALLOC2	173	F, A pointer passed to DEALLOCATE points to an array that cannot be deallocated 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 previously allocated with an ALLOCATE statement can be validly passed to DEALLOCATE.
INVKEYSPE	49	F, invalid key specification A key specification in an OPEN statement or in a keyed access READ statement was invalid. For example, the key length may have been zero or greater than 255 bytes, or the key length may not conform to the key specification of the existing file.
INVLOGUNI	32	F, invalid logical unit number A logical unit number less than zero or greater than 2,147,483,647 was used in an I/O statement.
INVMATKEY	94	F, invalid key match specifier for key direction A keyed READ used an invalid key match specifier for the direction of that key. Use KEYGE and KEYGT only on ascending keys. Use KEYLE and KEYLT only on descending keys. Use KEYNXT and KEYNXTNE to avoid enforcement of key direction and match specifier.
INVREALLOC	151	F, allocatable array is already allocated A Fortran allocatable array must not already be allocated when you attempt to allocate it. You must deallocate the array before it can again be allocated.
INVREFVAR	19	F, invalid reference to variable in NAMELIST input

Mnemonic	Number	Severity Code, Message Text, and Explanation
		The variable in error is shown as “varname” in the message text. One of the following conditions occurred: • The variable was not a member of the namelist group. • An attempt was made to subscript the 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 using an unsubscripted array variable.
KEYIO_DIR	260	F, keyed-access I/O to unit open for direct access The OPEN for this unit number specified direct access and the I/O statement specifies keyed access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access. (For more information on statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).
KEYVALERR	45	F, keyword value error in OPEN statement An improper value was specified for an OPEN or CLOSE statement keyword specifier requiring a value.
LISIO_SYN	59	F, list-directed I/O syntax error 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.
LOGZERNEG	83	F, logarithm of zero or negative value An attempt was made to take the logarithm of zero or a negative number. The result returned was –0 for VAX-format floating-point and NaN for IEEE-format floating-point.
MIXFILACC	31	F, mixed file access modes One of the following conditions occurred: • An attempt was made to use both formatted and unformatted operations on the same unit.

Mnemonic	Number	Severity Code, Message Text, and Explanation
		• An attempt was made to use an invalid combination of access modes on a unit, such as direct and sequential. The only valid combination is sequential and keyed access on a unit opened with ACCESS= KEYED. • An attempt was made to execute a Fortran I/O statement on a logical unit that was opened by a program coded in a language other than Fortran.
NOTFORSPE	1	F, not a Fortran-specific error An error occurred in the user program or in the VSI Fortran RTL Fortran RTL that was not a Fortran-specific error.
NO_CURREC	53	F, no current record A REWRITE or current record DELETE operation was attempted when no current record was defined. 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.
NO_SUCDEV	42	F, no such device A file specification included an invalid or unknown device name when an OPEN operation was attempted.
NULPTRERR	146	F, null pointer error Attempted to use a pointer that does not contain an address. Modify the source program, recompile, and relink.
OPEDEFREQ	26	F, OPEN or DEFINE FILE required for keyed or direct access One of the following conditions occurred: • A direct access READ, WRITE, FIND, or DELETE statement specified a file that was not opened with a DEFINE FILE statement or with an OPEN statement specifying ACCESS= DIRECT. • A keyed access READ statement specified a file that was not opened with an OPEN statement specifying ACCESS= KEYED.
OPEFAI	30	F, open failure An error was detected by RMS 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 process I/O file that had been closed

Mnemonic	Number	Severity Code, Message Text, and Explanation
OPEREQSEQ	265	F, operation requires sequential file organization and access Attempted BACKSPACE operation for a unit that is not connected to a sequential file opened for sequential access. Make sure the BACKSPACE statement specified the right unit number and the OPEN statement specified the correct file and access.
OPERREQDIS	264	F, operation requires file to be on disk or tape Attempted BACKSPACE operation for a unit that is not connected to a disk or tape device. Make sure the BACKSPACE statement specified the right unit number and the OPEN statement specified the correct device.
OUTCONERR	63	E or I, output conversion error 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. If /CHECK=NOOUTPUT_CONVERSION is in effect (see Section 2.3.11), the severity is I (program continues). When /CHECK=NOOUTPUT_CONVERSION is in effect or if no ERR address is defined for the I/O statement encountering this error, the program continues and the entire overflowed field is filled with asterisks to indicate the error in the output record.
OUTSTAOVE	66	F, output statement overflows record An output statement attempted to transfer more data than would fit in the maximum record size.
RANGEERR	150	F, range error An integer value appears in a context where the value of the integer is outside the permissible range.
RECIO_OPE	40	F, recursive I/O operation While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted. One of the following conditions may have occurred: • A function subprogram that performs I/O to the same logical unit was referenced in an expression in an I/O list or variable format expression. • An I/O statement was executed at AST level for the same logical unit. • An exception handler (or a procedure it called) executed an I/O statement in response to a signal from an I/O statement for the same logical unit.
RECNUMOUT	25	F, record number outside range A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was created.

Mnemonic	Number	Severity Code, Message Text, and Explanation
RESACQFAI	152	F, unresolved contention for VSI Fortran RTL global resource. Failed to acquire a VSI 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 the main program) and your asynchronous handler attempted to acquire the same global resource.
REWERR	20	F, REWIND error One of the following conditions occurred: • The file was not a sequential organization file. • The file was not opened for sequential or append access. • RMS detected an error condition during execution of a REWIND statement.
REWRITERR	54	F, REWRITE error RMS detected an error condition during execution of a REWRITE statement.
ROPRAND	144	F, reserved operand The VSI Fortran encountered a reserved operand. Please report the problem to VSI.
SEGRECFOR	35	F, segmented record format error 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.
SEQIO_DIR	259	F, sequential-access I/O to unit open for direct access The OPEN for this unit number specified direct access and the I/O statement specifies sequential access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access. (For more information on statements, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/]).
SHORTDATEARG	175	F, DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8 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

Mnemonic	Number	Severity Code, Message Text, and Explanation
		argument to be at least 8 characters in length. Verify that the TIME and ZONE arguments also meet their minimum lengths.
SHORTTIMEARG	176	F, TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10 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.
SHORTZONEARG	177	F, ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5 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 TIME and DATE arguments also meet their minimum lengths.
SIGLOSMAT	87	F, significance lost in math library The magnitude of an argument or the magnitude of the ratio of the arguments to a math library function was so large that all significance in the result was lost. The result returned was the reserved operand, -0.
SPERECLOC	52	F, specified record locked A read operation or direct access write, find, or delete operation was attempted on a record that was locked by another user.
SQUROONEG	84	F, square root of negative value An argument required the evaluation of the square root of a negative value. The result returned was the reserved operand, –0 for VAX-format floating-point and NaN for IEEE-format floating-point.
STKOVF	147	F, stack overflow The VSI Fortran RTL encountered a stack overflow while executing your program.
STRLENERR	148	F, string length error During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range. Try recompiling with /CHECK=BOUNDS or examine source code.
SUBRNG	77	F, subscript out of range An array reference was detected outside the declared array bounds.
SUBSTRERR	149	F, substring error

Mnemonic	Number	Severity Code, Message Text, and Explanation
		An array subscript is outside the dimensioned boundaries of an array. Try recompiling with /CHECK=BOUNDS or examine source code.
SYNERRFOR	62	F, syntax error in format A syntax error was encountered while the VSI Fortran RTL was processing a format stored in an array or character variable.
SYNERRNAM	17	F, syntax error in NAMELIST input “text” The syntax of input to a namelist-directed READ statement was incorrect. (The part of the record in which the error was detected is shown as “text” in the message text.)
TOOMANREC	27	F, too many records in I/O statement One of the following conditions occurred: • An attempt was made to read or write more than one record with an ENCODE or DECODE statement. • An attempt was made to write more records than existed.
TOOMANVAL	18	F, too many values for NAMELIST variable “varname” An attempt was made to assign too many values to a variable during a namelist-directed READ statement. (The name of the variable is shown as “varname” in the message text.)
UNDEXP	82	F, undefined exponentiation An exponentiation that is mathematically undefined was attempted, for example, 0. * *0. The result returned for floating-point operations was –0 for VAX-format floating-point and NaN for IEEE-format floating-point. For integer operations, zero is returned.
UNFIO_FMT	256	F, unformatted I/O to unit open for formatted transfers Attempted unformatted I/O to a unit where the OPEN statement indicated the file was formatted (FORM specifier). Check that the correct unit (file) was specified. If the FORM specifier was not specified in the OPEN statement and the file should contain unformatted data, specify FORM= 'UNFORMATTED' in the OPEN statement. Otherwise, if appropriate, use formatted I/O (such as list-directed or namelist I/O).
UNIALROPE	34	F, unit already open A DEFINE FILE statement specified a logical unit that was already opened.
UNLERR	56	F, UNLOCK error RMS detected an error condition during execution of an UNLOCK statement.
VFEVALERR	68	F, variable format expression value error

Mnemonic	Number	Severity Code, Message Text, and Explanation
		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 one was assumed, except for a P edit descriptor, for which a value of zero was assumed.
WRIREAFIL	47	F, write to READONLY file A write operation was attempted to a file that was declared ACTION= 'READ' or READONLY in the OPEN statement that is currently in effect.
WRONUMARG	80	F, wrong number of arguments An improper number of arguments was used to call a math library procedure.

Appendix D. VSI Fortran Logical Names

This appendix describes:

- Section D.1: Commands for Assigning and Deassigning Logical Names
- Section D.2: Compile-Time Logical Names
- Section D.3: Run-Time Logical Names

D.1. Commands for Assigning and Deassigning Logical Names

VSI Fortran recognizes certain logical names at compile time and run time.

A logical name specified must not contain brackets, semicolons, or periods. The system treats any name containing these punctuation marks as a file specification, not as a logical name.

To view the previously set logical names, use the SHOW LOGICAL command (see HELP SHOW LOGICAL):

```
$ SHOW LOGICAL PROJ_DIR
%SHOW-S-NOTRAN, no translation for logical name PROJ_DIR
```

Use a DEFINE command to set (define) a logical name:

```
$ DEFINE logical-name equivalence-value
```

For example, to associate the process logical name PROJ_DIR with the directory DISK2:[PROJ], type a DEFINE command:

```
$ DEFINE PROJ_DIR  DISK2:[PROJ]
$ SHOW LOGICAL PROJ_DIR
   "PROJ_DIR" = "DISK2:[PROJ]" (LNM$PROCESS_TABLE)
```

To use logical name tables other than the process table, specify qualifiers to the DEFINE (or ASSIGN) command. A system manager can define logical names system-wide, perhaps in a system startup procedure.

To remove the association of a logical name and its value, use the DEASSIGN command:

```
$ DEASSIGN logical-name
```

D.2. Compile-Time Logical Names

Table D.1 describes the logical names that VSI Fortran recognizes at compile time.

Table D.1. VSI Fortran Compile-Time Logical Names

Logical Name	Description
FORT$LIBRARY	Specifies a default text library to be searched for included library modules (INCLUDE statements that specify library modules) that are not explicitly specified.

Logical Name	Description
	FORT$LIBRARY is recognized by both Compaq Fortran 77 and VSI Fortran. For more information, see Section 2.2.4.
FORT$INCLUDE	Specifies an additional directory to be searched for included source files (INCLUDE statements that specify files). For more information, see Section 2.3.25.

D.3. Run-Time Logical Names

Table D.2 describes the logical names VSI Fortran recognizes at run time.

Table D.2. VSI Fortran Run-Time Logical Names

Logical Name	Description
FOR*nnn*	Allows the user to specify the directory and file name at run time for a logical unit (`nnn`) for which the OPEN statement does not specify a file name (leading zeros are required). If the appropriate logical name is not set and the OPEN statement does not specify a file name for that logical unit, a default file name of FOR*nnn*.DAT is used. For more information, see Section 6.6.1.
FOR$READ	Specifies the name of a file to receive input from a READ statement instead of SYS$INPUT. For more information, see Section 6.6.1.
FOR$ACCEPT	Specifies the name of a file instead of SYS$INPUT to receive input from an ACCEPT statement. For more information, see Section 6.6.1.
FOR$PRINT	Specifies the name of a file instead of SYS$OUTPUT to receive output from a PRINT statement. For more information, see Section 6.6.1.
FOR$TYPE	Specifies the name of a file instead of SYS$OUTPUT to receive output from a TYPE statement. For more information, see Section 6.6.1.
FOR$CONVERT*nnn*	For an unformatted file, specifies the nonnative numeric format of the data at run time for a logical unit (`nnn`). Otherwise, the nonnative numeric format of the unformatted data must be specified at compile time by using the FORTRAN command /CONVERT qualifier. For more information, see Chapter 9.
FOR$CONVERT.*ext* and FOR$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 FORTRAN command /CONVERT qualifier. For more information, see Chapter 9.

Appendix E. Contents of the VSI Fortran System Library FORSYSDEF

Table E.1 lists the library modules contained in the VSI Fortran system library FORSYSDEF.TLB, a text library.

The library modules consist of definitions, in Fortran source code, of related groups of system symbols that can be used in calling OpenVMS system services. FORSYSDEF also contains library modules that define the condition symbols and the entry points for Run-Time Library procedures.

Each new version of the operating system may add new FORSYSDEF library modules to the text library FORSYSDEF. To determine whether a particular library module is installed on your system, use the LIBRARY/LIST command.

For example, the following command sequence searches for the library module named $SMGDEF:

```
$ LIBRARY/LIST/ONLY=$SMGDEF SYS$LIBRARY:FORSYSDEF.TLB
```

For More Information:

- On including files and text library modules and INCLUDE statement forms, see Section 2.2.4.
- On including FORSYSDEF library modules, see Section 10.8.1.
- On condition values and symbols, see Section 14.4.3.
- On example programs that use system services, see Appendix F.

Table E.1. Contents of System Library FORSYSDEF

Module Name	Description
$ACCDEF	Accounting manager request type codes
$ACEDEF	Access control list entry structure definitions
$ACLDEF	Access control list interface definitions
$ACRDEF	Accounting record definitions
$AFRDEF	Alignment fault reporting
$AGNDEF	$ASSIGN flags bit definitions
$ALPHADEF	Alpha system hardware model identification
$ARGDEF	Argument descriptor for object language procedure records
$ARMDEF	Access rights mask longword definitions
$ATRDEF	File attribute list description—used to read and write file attributes
$BRKDEF	Breakthru ($BRKTHRU) system service input definitions
$CHFDEF	Condition handling argument list offsets
$CHKPNTDEF	Create checkpointable processes flag definitions
$CHPDEF	Check protection ($CHKPRO) system service definitions
$CLIDEF	Command language interface definitions—define the offset values for structures used to communicate information to CLI
$CLIMSGDEF	Command language interface message definitions

Module Name	Description
$CLISERVDEF	CLI service request code definitions
$CLIVERBDEF	CLI generic verb codes definitions
$CLSDEF	Security classification mask block—contains security and integrity level categories for nondiscretionary access controls
$CLUEVTDEF	Cluster Event Notification Services definitions
$CMBDEF	$CREMBX flags bit definitions
$CQUALDEF	Common qualifier package definitions
$CRDEF	Card reader status bits
$CREDEF	Create options table definitions for library facility
$CRFDEF	CRF$_INSRTREF argument list
$CRFMSG	Return status codes for cross reference program
$CVTDEF	RTL floating-point conversion routines
$CVTMSG	RTL floating-point conversion routines messages
$DCDEF	Device adapter, class, and type definitions
$DDTMDEF	DECdtm transaction manager services structure definitions
$DEVDEF	I/O device characteristics
$DIBDEF	Device information block definitions
$DMTDEF	Flag bits for the Dismount ($DISMOU) system service
$DSCDEF	Descriptor type definitions
$DSTDEF	Debug symbol table definitions
$DTIDEF	Distributed Transaction Information flag definitions
$DTKDEF	Definitions for RTL DECtalk management facility
$DTKMSG	Return status codes for RTL DECtalk management facility
$DVIDEF	Device and volume information data identifier definitions
$DVSDEF	Device scan data identifier definitions
$EEOMDEF	End of module record
$EGPSDEF	Global section descriptor entry - program section definition
$EGSDEF	Global symbol definition record
$EGSTDEF	Universal symbol definition (used by Linker)
$EGSYDEF	Global symbol definition entry
$EIDCDEF	Random entity ident consistency check
$EMHDEF	Module header record
$ENVDEF	Environment definitions in object file
$EOBJRECDEF	Object file record formats
$EOMDEF	End of module record in object/image files
$EOMWDEF	End of module record in object/image with word of psect value
$EPMDEF	Global symbol definition record in object file—entry point definitions
$EPMMDEF	Global section string descriptor entry point definition, version mask symbols

Module Name	Description
$EPMVDEF	Global section string descriptor entry point definition, vectored symbols
$EPMWDEF	Global symbol definition record in object file—entry point definitions with word of psect value
$ERADEF	Erase type code definitions
$ESDFDEF	Universal symbol definition (used by linker)
$ESDFMDEF	Symbol definition for version mask symbols
$ESDFVDEF	Symbol definition for vectored symbols
$ESGPSDEF	Global section descriptor entry - program section definition in shareable image
$ESRFDEF	Symbol reference
$ETIRDEF	Text, information, and relocation record
$EVAX_INSTRDEF	Instruction class definition
$EVX_OPCODES	Alpha instruction format definitions
$FABDEF	RMS File access block definitions
$FALDEF	Messages for the FAL (File Access Listener) facility
$FDLDEF	File Definition Language call interface control flags
$FIBDEF	File identification block definitions
$FIDDEF	File ID structure
$FLTDEF	Flag bits for $SETFLT system service
$FMLDEF	Formal argument definitions appended to procedure definitions in global symbol definition record in object file
$FOR_FP_CLASS	Return values for VSI Fortran FP_CLASS intrinsic procedure
$FORDEF	Condition symbols for VSI Fortran Run-Time Library
$FORIOSDEF	VSI Fortran IOSTAT error numbers
$FSCNDEF	Descriptor codes for SYS$FILESCAN
$GPSDEF	Global symbol definition record in object file—psect definitions
$GSDEF	Global symbol definition (GSD) record in object file
$GSYDEF	Global symbol definition record in object file—symbol definitions
$HLPDEF	Data structures for help processing
$HWDEF	Architecture identification
$IACDEF	Image activation control flags
$IDCDEF	Object file IDENT consistency check structures
$IEEEDEF	IEEE floating-point control register definitions
$IMPDEF	$PERSONA_xxx service definitions
$INITDEF	Values for INIT$_DENSITY item code (media format)
$IODEF	I/O function codes
$JBCMSGDEF	Job controller message definition
$JPIDEF	Job/process information request type codes
$KGBDEF	Key grant block definitions—formats of records in rights database file

Module Name	Description
$LADEF	Laboratory peripheral accelerator device types
$LATDEF	LAT facility definitions
$LATMSGDEF	Error messages for LAT facility
$LBRCTLTBL	Librarian control table definitions
$LBRDEF	Library type definitions
$LCKDEF	Lock manager definitions
$LEPMDEF	Global symbol definition record in object file—module local entry point definitions
$LHIDEF	Library header information array offsets
$LIBCLIDEF	Definitions for LIB$ CLI callback procedures
$LIBDCFDEF	Definitions for LIB$DECODE_FAULT procedure
$LIBDEF	Condition symbols for the general utility library
$LIBDTDEF	Interface definitions for LIB$DT (date/time) package
$LIBFISDEF	LIB$FIND_IMAGE_SYMBOL flags
$LIBICB	LIB$GET_INVO_CONTEXT definitions
$LIBVMDEF	Interface definitions for LIB$VM (virtual memory) package
$LICENSEDEF	License Management Facility definitions
$LKIDEF	Get lock information data identifier definitions
$LMFDEF	License Management Facility definitions
$LNKDEF	Linker option record definition in object file
$LNMDEF	Logical name flag definitions
$LPDEF	Line printer characteristic codes
$LPRODEF	Global symbol definition record in object file—module local procedure definition in object file
$LSDFDEF	Module local symbol definition in object file
$LSRFDEF	Module local symbol reference in object file
$LSYDEF	Module local symbol definition
$MAILDEF	Callable mail definitions
$MHDDEF	Main header record definitions
$MHDEF	Module header record definition in object file
$MMEDEF	Media management extensions definitions
$MMEMSGDEF	Media management extensions messages
$MMIDEF	Media management interface definitions
$MNTDEF	Flag bits and function codes for the MOUNT system service
$MSGDEF	Symbolic names to identify mailbox message senders
$MSGHLPDEF	Message help values
$MT2DEF	Extended magnetic tape characteristic bit definition
$MTADEF	Magnetic tape accessibility routine codes

Module Name	Description
$MTDEF	Magnetic tape characteristic codes
$MTHDEF	Condition symbols from the mathematical procedures library
$NAMDEF	RMS name block field definitions
$NCSDEF	Interface definitions for National Character set package
$NETDEF	DECnet-VAX identification definitions
$NSADEF	Security auditing packet header and record type definitions
$NSARECDEF	Security auditing record definitions
$OBJRECDEF	Object language record definition
$OPCDEF	Operator communication manager request type codes–return status codes
$OPCMSG	OPCOM message definitions
$OPRDEF	Operator communications message types and values
$OSSDEF	Object security service definitions
$OTSDEF	Language-independent support procedure (OTS$) return status codes
$PAGEDEF	Page size and limit definitions
$PQLDEF	Quota types for process creation quota list
$PRCDEF	Create process ($CREPRC) system service status flags and item codes
$PRDEF	Processor register definitions
$PRODEF	Global symbol definition record in object file—procedure definition
$PROMDEF	Global section string descriptor entry: procedure definition, version mask symbols
$PROVDEF	Global section string descriptor entry: procedure definition, vectored symbols
$PROWDEF	Global symbol definition record in object file—procedure definition with word of psect value
$PRTDEF	Protection field definitions
$PRVDEF	Privilege bit definitions
$PSCANDEF	Process scan item code definitions
$PSIGDEF	Signature block offset definitions for calling standard
$PSLDEF	Processor status longword (PSL) mask and symbolic names for access modes
$PSMMSGDEF	Print symbiont message definitions
$PTDDEF	Processor status word mask and field definitions
$QUIDEF	Get queue information service definitions
$RABDEF	RMS record access block definitions
$RMEDEF	RMS escape definitions
$RMSDEF	RMS return status codes
$RNHBLKDEF	Get Queue Information Service ($GETQUI) definitions, item codes
$SBKDEF	Open file statistics block
$SCRDEF	Screen package interface definitions
$SDFDEF	Symbol record in object file

Module Name	Description
$SDFMDEF	Symbol definitions
$SDFVDEF	Symbol definitions for vectored symbols
$SDFWDEF	Symbol record in object file with word of psect value
$SECDEF	Attribute flags for private/global section creation and mapping
$SGPSDEF	Global symbol definition record in object file—P-section definition in shareable image
$SHRDEF	Definitions for shared messages
$SJCDEF	Send to job controller service definitions
$SMBMSGDEF	Symbiont manager message definitions
$SMGDEF	Definitions for RTL screen management
$SMGMSG	Messages for the Screen Management facility
$SMGTRMPTR	Terminal capability pointers for RTL SMG$ facility
$SMRDEF	Define symbiont manager request codes
$SNAPEVTDEF	System snapshot/reboot status definitions
$SORDEF	Messages for the Sort/Merge facility
$SRFDEF	Global symbol definition record in object file—symbol reference definitions
$SRMDEF	SRM hardware symbol definitions
$SSDEF	System service failure and status codes
$STENVDEF	Assign symbiont manager print job/record option codes
$STRDEF	String manipulation procedures (STR$) return status codes
$STSDEF	Status codes and error codes
$SYIDEF	Get system information data identifier definitions
$SYSSRVNAM	System service entry point descriptions
$TIRDEF	Text information and relocation record in object file
$TPADEF	TPARSE control block
$TPUDEF	TPU callable interface definitions
$TRMDEF	Define symbols to the item list QIO format
$TT2DEF	Terminal special symbols
$TT3DEF	Terminal special symbols
$TTCDEF	Character code format
$TTDEF	Terminal device characteristic codes
$UAIDEF	User Authorization Information data identifier definitions
$UICDEF	Format of user identification code (UIC)
$UIDDEF	Universal identifier definitions
$USGDEF	Disk usage accounting file produced by ANALYZE/DISK_STRUCTURE
$USRIDEF	User image bit definitions
$VAXDEF	VAX model definitions
$XABALLDEF	Allocation XAB definitions

Module Name	Description
$XABCXFDEF	RMS context XAB associated with FAB
$XABCXRDEF	RMS context XAB associated with RAB
$XABDATDEF	Date/time XAB definitions
$XABDEF	Definitions for all XABs
$XABFHCDEF	File header characteristics XAB definitions
$XABITMDEF	Item list XAB
$XABJNLDEF	Journal XAB definitions
$XABKEYDEF	Key definitions XAB field definitions
$XABPRODEF	Protection XAB field definitions
$XABRDTDEF	Revision date/time XAB definitions
$XABRUDEF	Recovery unit XAB
$XABSUMDEF	Summary XAB field definitions
$XABTRMDEF	Terminal control XAB field definitions
$XADEF	DR11-W definitions for device specific characteristics
$XKDEVDEF	3271 device status block
$XKSTSDEF	Definitions for 3271 line status block (returned by IO$_RDSTATS)
$XMDEF	DMC-11 device characteristic codes
$XWDEF	System definition for software DDCMP
CMA$DEF	DECthreads definitions
CVT$ROUTINES	Floating-point conversion routines
DTK$ROUTINES	Routine definitions for DECtalk facility
LIB$ROUTINES	Routine definitions for general purpose run-time library procedures
MTH$ROUTINES	Routine definitions for mathematics run-time library procedures
NCS$ROUTINES	Routine definitions for National Character set procedure
OTS$ROUTINES	Routine definitions for language-independent support procedures
PPL$DEF	Definitions for Parallel Processing library facility
PPL$MSG	Message definitions for Parallel Processing library facility
PPL$ROUTINES	Routine definitions for Parallel Processing library facility
SMG$ROUTINES	Routine definitions for Screen Management procedures
SOR$ROUTINES	Routine definitions for Sort/Merge procedures
STR$ROUTINES	Routine definitions for string manipulation procedures

Appendix F. Using System Services: Examples

This appendix contains examples that involve accessing OpenVMS system services from VSI Fortran programs. The individual examples address the following operations:

- Section F.1: Calling RMS Procedures
- Section F.2: Using an AST Routine
- Section F.3: Accessing Devices Using Synchronous I/O
- Section F.4: Communicating with Other Processes
- Section F.5: Sharing Data
- Section F.6: Displaying Data at Terminals
- Section F.7: Creating, Accessing, and Ordering Files
- Section F.8: Measuring and Improving Performance
- Section F.9: Accessing Help Libraries
- Section F.10: Creating and Managing Other Processes

Each example includes the free-form source program (with comments), a sample use of the program, and explanatory notes.

F.1. Calling RMS Procedures

When you explicitly call an RMS system service, the order of the arguments in the call must correspond with the order shown in the *VSI OpenVMS Record Management Services Reference Manual*. You must use commas to reserve a place in the call for every argument. If you omit an argument, the procedure uses a default value of zero.

When calling an RMS routine from VSI Fortran, the procedure name format is SYS$procedure_name. The following example shows a call to the RMS procedure SYS$SETDDIR. This RMS procedure sets the default directory for a process.

Source Program:

```
!   File: SETDDIR.F90
!
!   This program calls the RMS procedure $SETDDIR to change
!   the default directory for the process.

    IMPLICIT INTEGER (A - Z)
    CHARACTER(LEN=17) DIR /'[EX.PROG.FOR]'/     ❶
    STAT = SYS$SETDDIR (DIR,,)                  ❷
    IF (.NOT. STAT) TYPE *, 'ERROR'
```

```
    END PROGRAM
```

❶ The default directory name is initialized into a CHARACTER variable.
❷ The call to $SETDDIR contains one argument, the directory name, which is passed by descriptor, the default argument passing mechanism for CHARACTERs. The omitted arguments are optional, but commas are necessary to reserve places in the argument list.

Sample Use:

```
$ DIRECTORY                ❶
 Directory WORK$:[EX.PROG.FOR.CALL]
 BASSUM.BAS;1      BASSUM.OBJ;1      COBSUM.COM;1      DOCOMMAND.F90;2
 GETMSG.EXE;1      GETMSG.F90;4      GETMSG.LIS;2      GETMSG.OBJ;1
 SETDDIR.F90;3     SETDDIR.LIS;1
Total of 10 files.
$ FORTRAN SETDDIR
$ LINK SETDDIR
$ RUN SETDDIR
$ DIRECTORY                ❷
 Directory WORK$:[EX.PROG.FOR]
 CALL.DIR;1      COMU.DIR;1      DEVC.DIR;1      FIL.DIR;1
 HAND.DIR;1      INTR.DIR;1      LNKR.DIR;1      MNAG.DIR;1
 RMS.DIR;1       SHAR.DIR;1      SYNC.DIR;1      TERM.DIR;1
 Total of 12 files.
```

❶ The DIRECTORY command executed *before* the SETDDIR program is run shows that the following directory is the default:

```
WORK$:[EX.V4PROG.FOR.CALL]
```

This directory contains the file SETDDIR.F90.
❷ Another DIRECTORY command *after* the SETDDIR program is run shows that the default directory has changed. The following directory is the new default directory:

```
WORK$:[EX.PROG.FOR]
```

For More Information:

On calling RMS system services, see Chapter 11.

F.2. Using an AST Routine

The following example demonstrates how to request and declare an AST procedure. It consists of the following:

- The main program CTRLC defines a common block AST_COM that contains the logical variable CTRLC_FLAG and integer channel number, calls the ENABLE_AST routine to set up **Ctrl/C** trapping, and contains a DO loop that allows **Ctrl/C** interruption.

- A subroutine named ENABLE_AST that enables **Ctrl/C** trapping using the SYS$QIOW system service. It is called by the main program and the AST_ROUTINE subroutine.

- A subroutine named AST_ROUTINE that gets called when a **Ctrl/C** is pressed. It resets the logical variable CTRLC_FLAG and calls the ENABLE_AST subroutine to allow repetitive **Ctrl/C** use.

For More Information:

On AST routines, see the *HP OpenVMS System Services Reference Manual*.

Source Programs:

```
!  Sample program to show enabling of an AST in Fortran
!
!  The program uses a Ctrl/C AST to interrupt a work loop in the
!  main program.
!
     PROGRAM CTRLC
     IMPLICIT NONE

     LOGICAL CTRLC_FLAG                ! Set to TRUE when Ctrl/C is pressed
     INTEGER (KIND=2) CHANNEL          ! Channel for terminal
     COMMON /AST_COM/ CTRLC_FLAG,CHANNEL
     VOLATILE CTRLC_FLAG               ! Required because variable  (1)
                                       ! can change at any time
     INTEGER ITERATIONS,I

!  Do first-time initialization

     CHANNEL = 0
     CTRLC_FLAG = .FALSE.
     CALL ENABLE_AST

!  Read iteration count

100  WRITE (*,'($,A)') ' Enter iteration count (0 to exit): '
     READ (*,*) ITERATIONS
     DO I=1,ITERATIONS
        IF (CTRLC_FLAG) GOTO 200       ! Was Ctrl/C pressed?
        WRITE (*,*) 'Count is ',I
        CALL LIB$WAIT (2.0)            ! Pause 2 seconds
     END DO
     IF (ITERATIONS .EQ. 0) GOTO 999
     GOTO 100    ! Loop back

200  WRITE (*,*) 'Ctrl/C pressed'
     CTRLC_FLAG = .FALSE.
     GOTO 100

999  END PROGRAM CTRLC

!  Subroutine ENABLE_AST

    SUBROUTINE ENABLE_AST                                  (2)
    IMPLICIT NONE

    INCLUDE '($SYSSRVNAM)'             ! System services
    INCLUDE '($IODEF)'                 ! $QIO function codes

    LOGICAL CTRLC_FLAG
    VOLATILE CTRLC_FLAG                                    (1)
    INTEGER (KIND=2) CHANNEL
    COMMON /AST_COM/ CTRLC_FLAG,CHANNEL
```

```
   EXTERNAL AST_ROUTINE

   INTEGER ASSIGN_STATUS, QIO_STATUS, IOSB(2)

!  Assign channel if not already assigned

   IF (CHANNEL .EQ. 0) THEN
     ASSIGN_STATUS = SYS$ASSIGN ('TT:', CHANNEL,,,)
     IF (.NOT. ASSIGN_STATUS) CALL LIB$SIGNAL(%VAL(ASSIGN_STATUS))
   END IF

!  Enable AST so that AST_ROUTINE is called when Ctrl/C is pressed.


   QIO_STATUS = SYS$QIOW (,   &                                    ❸
                %VAL(CHANNEL), &
                %VAL(IO$_SETMODE .OR. IO$M_CTRLCAST), &
                IOSB,,, &
                AST_ROUTINE,,,,,)

   IF (.NOT. QIO_STATUS) CALL LIB$SIGNAL(%VAL(QIO_STATUS))

   RETURN
   END SUBROUTINE ENABLE_AST

!  Subroutine AST_ROUTINE

   SUBROUTINE AST_ROUTINE                                          ❹
   IMPLICIT NONE

   LOGICAL CTRLC_FLAG
   VOLATILE CTRLC_FLAG                                             ❶
   INTEGER (KIND=2) CHANNEL
   COMMON /AST_COM/ CTRLC_FLAG,CHANNEL

!  Indicate that a CTRL/C has been pressed

   CTRLC_FLAG = .TRUE.

!  Reenable the AST.  This must be done by calling ENABLE_AST rather than
!  doing it here as we would need a recursive reference to AST_ROUTINE,
!  which is disallowed unless /RECURSIVE is used.

   CALL ENABLE_AST                                                 ❺

   RETURN
   END SUBROUTINE AST_ROUTINE
```

Sample Use:

```
$ RUN CTRLC
Enter iteration count (0 to exit):
9
Count is            1
Count is            2
Count is            3
Ctrl/C                    ❺
```

```
Cancel

Ctrl/C pressed
Enter iteration count (0 to exit):
0
$
```

❶ The CTRLC_FLAG logical variable is declared volatile in the routines that reference it because its value could change at any point during program execution (other than an assignment statement or subroutine argument).

❷ By providing two subroutines, you allow the **Ctrl/C** AST routine to be executed repeatedly, rather than just once. The ENABLE_AST subroutine is called by the main program and the AST_ROUTINE subroutine. It enables **Ctrl/C** trapping using the SYS$QIOW system service and sets the CTRLC_FLAGS logical variable. For a subroutine to call itself, it must be recursive.

❸ The call to the SYS$QIOW system service enables **Ctrl/C** AST use by specifying that the subroutine AST_ROUTINE be called when **Ctrl/C** is pressed.

❹ When the AST is delivered, the AST_ROUTINE receives control, resets the CTRLC_FLAG logical variable, and returns control back to where **Ctrl/C** was pressed (main program), which eventually displays "**Ctrl/C** pressed".

The arguments to AST_ROUTINE are platform dependent.

❺ The example shows the program executing within the DO loop in the main program (with a two second delay between DO loop executions). When the user types **Ctrl/C**, control is transferred briefly to the AST_ROUTINE subroutine and it then returns back to the main program. Within the DO loop, the main program tests the value of logical variable CTRLC_FLAG and, if set to .TRUE., transfers control to label 200 which displays "**Ctrl/C** pressed".

For More Information:

On the VOLATILE statement, see the *VSI Fortran Reference Manual* [https://docs.vmssoftware.com/vsi-fortran-for-openvms-language-reference-manual/].

F.3. Accessing Devices Using Synchronous I/O

The following example performs output to a terminal via the SYS$QIOW system service.

Source Program:

```
!  File: QIOW.F90
!
!  This program demonstrates the use of the $QIOW system service to
!  perform synchronous I/O to a terminal.

   IMPLICIT         INTEGER (KIND=4) (A - Z)
   INCLUDE          '($SYSSRVNAM)'
   INCLUDE          '($IODEF)'
   CHARACTER(LEN=24)    TEXT_STRING /'This is from a SYS$QIOW.'/   ❶
   CHARACTER(LEN=11)    TERMINAL /'SYS$COMMAND'/
   INTEGER KIND=2)      TERM_CHAN
   STRUCTURE /TT_WRITE_IOSB/
           INTEGER (KIND=2)   STATUS
           INTEGER (KIND=2)   BYTES_WRITTEN
           INTEGER (KIND=4)   %FILL
```

```
   END STRUCTURE
   RECORD /TT_WRITE_IOSB/  IOSB

!  Assign the channel number

   STAT = SYS$ASSIGN (TERMINAL, TERM_CHAN,,)
   IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT))                          ❷

!  Initialize STATUS to zero (0)

   STATUS = 0

!  Output the message twice

   DO I=1,2
      STAT = SYS$QIOW (%VAL(1),%VAL(TERM_CHAN), &                      ❸
                       %VAL(IO$_WRITEVBLK),IOSB,,, &
                       %REF(TEXT_STRING), &
                       %VAL(LEN(TEXT_STRING)),, &
                       %VAL(32),,)

       IF (.NOT. STAT) CALL LIB$STOP (%VAL(STATUS))
       IF (.NOT. IOSB.STATUS) CALL LIB$STOP (%VAL(IOSB.STATUS))
   ENDDO
   END PROGRAM
```

Sample Use:

```
$ FORTRAN QIOW
$ LINK QIOW
$ RUN QIOW
This is from a SYS$QIOW.
This is from a SYS$QIOW.
```

❶ If SYS$QIO and a SYS$WAITFR are used instead of SYS$QIOW, you must use a VOLATILE declaration for any program variables and arrays that can be changed while the operation is pending.

❷ TERM_CHAN receives the channel number from the SYS$ASSIGN system service.

The process permanent logical name SYS$COMMAND is assigned to your terminal when you log in. The SYS$ASSIGN system service translates the logical name to the actual device name.

❸ SYS$QIO and SYS$QIOW accept the CHAN argument by immediate value, unlike SYS$ASSIGN, which requires that it be passed by reference. Note the use of %VAL in the call to SYS$QIOW but not in the previous call to SYS$ASSIGN.

The function IO$_WRITEVBLK requires values for parameters P1, P2, and P4.

- P1 is the starting address of the buffer containing the message. So, TEXT_STRING is passed by reference.
- P2 is the number of bytes to be written to the terminal. A 24 is passed, since it is the length of the message string.
- P4 is the carriage control specifier; a 32 indicates single space carriage control.

A SYS$QIOW is issued, ensuring that the output operation will be completed before the program terminates.

F.4. Communicating with Other Processes

The following example shows how to create a global pagefile section and how two processes can use it to access the same data. One process executes the program PAGEFIL1, which must first be installed. When run, PAGEFIL1 creates and writes to a global pagefile section. PAGEFIL1 then waits for a second process to update the section. The second process executes PAGEFIL2, which maps and updates the pagefile section.

Because PAGEFIL2 maps to the temporary global pagefile section created in PAGEFIL1, PAGEFIL1 must be run first. The two processes coordinate their activity through common event flags.

Source Program: PAGEFIL1.F90

```
!   File: PAGEFIL1.F90
!
!   This program creates and maps a global page frame section.
!   Data in the section is accessed through an array.

    IMPLICIT INTEGER (KIND=4)     (A-Z)
    INCLUDE                  '($SECDEF)'
    INCLUDE                  '($SYSSRVNAM)'
    INCLUDE                  '($SYIDEF)'
    DIMENSION                MY_ADR(2),OUT_ADR(2)
    COMMON /MYCOM/           IARRAY(50)                    ❶
    CHARACTER(LEN=4)         NAME/'GSEC'/
    VOLATILE /MYCOM/                                       ❷


!   Associate with common cluster MYCLUS

    STATUS = SYS$ASCEFC (%VAL(64),'MYCLUS',,)              ❸

!   To calculate the ending address of the page boundary, call
!   LIB$GETSYIW to get the processor-specific page size, PAGE_MAX

    STATUS = LIB$GETSYI(SYI$_PAGE_SIZE,PAGE_MAX,,,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

    MY_ADR(1) = %LOC(IARRAY(1))                            ❶
    MY_ADR(2) = MY_ADR(1) + PAGE_MAX -1

!   Flags for call to SYS$CRMPSC

    SEC_FLAGS = SEC$M_PAGFIL.OR.SEC$M_GBL.OR.SEC$M_WRT.OR.SEC$M_DZRO

!   Create and map the temporary global section

    STATUS = SYS$CRMPSC(MY_ADR,OUT_ADR,,%VAL(SEC_FLAGS), &  ❹
                        NAME,,,,%VAL(1),,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

!   Manipulate the data in the global section              ❺

    DO 10 I  = 1,50
      IARRAY(I) = I
```

```
    END DO

    STATUS = SYS$SETEF(%VAL(72))
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    TYPE *,'Waiting for PAGEFIL2 to update section'
    STATUS = SYS$WAITFR(%VAL(73))

!   Print the array modified by PAGEFIL2 in the global section

    TYPE *, 'Modified data in the global section:'
    WRITE (6,100) (IARRAY(I), I=1,50)
100 FORMAT(10I5)
    END PROGRAM
```

❶ PAGEFIL1 and PAGEFIL2 are linked with the same options file, which specifies that the COMMON block program section is shareable, can be written to, and starts on a page boundary. The first argument to the SYS$CRMPSC (and SYS$MGBLSC) system service is a two-element array MYADR which specifies the starting and ending address.

❷ If any variables or arrays are used or modified, you should declare them as volatile in the other routines that reference them.

❸ Associate to a common event flag cluster to coordinate activity. The processes must be in the same UIC group.

❹ The $CRMPSC system service creates and maps a global pagefile section.

The starting and ending process virtual addresses of the section are placed in MY_ADR. The SEC$M_PAGFIL flag requests a temporary pagefile section. The flag SEC$M_GBL requests a global section. The flag SEC$M_WRT indicates that the pages should be writable as well as readable. The SEC$M_DZRO flag requests pages filled with zeros.

❺ Data is written to the pagefile section by PAGEFIL1.

Source Program: PAGEFIL2.F90

```
!   File: PAGEFIL2.F90
!
!   This program maps and modifies a global section after PAGEFIL1
!   creates the section.  Programs PAGEFIL1 and PAGEFIL2 synchronize
!   the processing of the global section through the use of common
!   event flags.

    IMPLICIT INTEGER (KIND=4)       (A - Z)
    INCLUDE                  '($SECDEF)'
    INCLUDE                  '($SYSSRVNAM)'
    INCLUDE                  '($SYIDEF)'
    DIMENSION                MY_ADR(2)                    ❶
    COMMON /MYCOM/           IARRAY(50)
    VOLATILE /MYCOM/                                      ❷

!   Call LIB$GETSYIW to get page size, PAGE_MAX

    STATUS = LIB$GETSYI(SYI$_PAGE_SIZE,PAGE_MAX,,,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

    MY_ADR(1) = %LOC(IARRAY(1))                           ❶
    MY_ADR(2) = MY_ADR(1) + PAGE_MAX -1

!   Associate with common cluster MYCLUS and wait for
```

```
!   event flag to be set

    STATUS = SYS$ASCEFC(%VAL(64),'MYCLUS',,)                ❸
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    STATUS = SYS$WAITFR (%VAL(72))
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

!   Set flag to allow section to be written

    FLAGS = SEC$M_WRT

!   Map the global section

    STATUS = SYS$MGBLSC(MY_ADR,,,%VAL(FLAGS),'GSEC',,)      ❹
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

!   Print out the data in the global section and            ❺
!   multiply each value by two

    TYPE *, 'Original data in the global section:'
    WRITE (6,100) (IARRAY(I), I=1,50)
100 FORMAT (10I5)
    DO I=1,50                                               ❻
      IARRAY(I) = IARRAY(I) * 2
    END DO

!   Set an event flag to allow PAGEFIL1 to continue execution

    STATUS = SYS$SETEF(%VAL(73))
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    END PROGRAM
```

❶ PAGEFIL1 and PAGEFIL2 are linked with the same options file, which specifies that the COMMON block program section is shareable, can be written to, and starts on a page boundary. The first argument to the SYS$CRMPSC (and SYS$MGBLSC) system service is a two-element array MYADR which specifies the starting and ending address.

❷ If any variables or arrays are used or modified, you should declare them as volatile in the other routines that reference them.

❸ Associate to a common event flag cluster to coordinate activity. The processes must be in the same UIC group.

❹ PAGEFIL2 maps the existing section as writable by specifying the SEC$M_WRT flag.

❺ PAGEFIL2 reads from the pagefile section.

❻ PAGEFIL2 modifies the data in the pagefile section.

The options file PAGEFIL.OPT contains the following line of source text:

```
PSECT_ATTR=MYCOM,PAGE,SHR,WRT,SOLITARY                      ❶
COLLECT=SHARED_CLUS,MYCOM                                   ❷
```

❶ PAGEFIL1 and PAGEFIL2 are linked with the same options file, which specifies that the COMMON block program section is shareable, can be written to, and starts on a page boundary. The first argument to the SYS$CRMPSC (and SYS$MGBLSC) system service is a two-element array MYADR which specifies the starting and ending address.

❷ The COLLECT option instructs the linker to create a cluster named SHARED_CLUS and to put the PSECT MYCOM into that cluster. This prevents the problem of inadvertently mapping another PSECT in a page containing all or part of MYCOM. Clusters are always positioned on page boundaries.

Sample Use:

```
$ FORTRAN /ALIGN=NATURAL PAGEFIL1
$ FORTRAN/ALIGN=NATURAL PAGEFIL2
$ LINK PAGEFIL1,PAGEFIL/OPTIONS                    ❶
$ LINK PAGEFIL2,PAGEFIL/OPTIONS                    ❶
$ RUN PAGEFIL1                !***Process 1***
Waiting for PAGEFIL2 to update section                ❷
Modified data in the global section:
    2    4    6    8   10   12   14   16   18   20
   22   24   26   28   30   32   34   36   38   40
   42   44   46   48   50   52   54   56   58   60
   62   64   66   68   70   72   74   76   78   80
   82   84   86   88   90   92   94   96   98  100

$
$

RUN PAGEFIL2                !***Process 2***
Original data in the global section:                  ❷
    1    2    3    4    5    6    7    8    9   10
   11   12   13   14   15   16   17   18   19   20
   21   22   23   24   25   26   27   28   29   30
   31   32   33   34   35   36   37   38   39   40
   41   42   43   44   45   46   47   48   49   50
$
```

❶ PAGEFIL1 and PAGEFIL2 are linked with the same options file, which specifies that the COMMON block program section is shareable, can be written to, and starts on a page boundary. The first argument to the SYS$CRMPSC (and SYS$MGBLSC) system service is a two-element array MYADR which specifies the starting and ending address.

❷ After PAGEFIL1 is run, creates the global section, writes out the data, and then displays:

```
Waiting for PAGEFIL2 to update section
```

A separate terminal is used to run PAGEFIL2, which displays the original data written by PAGEFIL1 and then modifies that data and exits. Once modified, PAGEFIL1 displays the data modified by PAGEFIL2 and exits.

For More Information:

On the VOLATILE statement, see the *VSI Fortran for OpenVMS Language Reference Manaul*.

F.5. Sharing Data

The program called SHAREDFIL is used to update records in a relative file. The SHARE qualifier is specified on the OPEN statement to invoke the RMS file sharing facility. In this example, the same program is used to access the file from two processes:

Source Program:

```
!   File: SHAREDFIL.F90
!
!   This program can be run from two or more processes to demonstrate the
!   use of an RMS shared file to share data.  The program requires the
!   relative file named REL.DAT.
```

```
    IMPLICIT       INTEGER (KIND=4) (A - Z)
    CHARACTER(LEN=20)   RECORD
    INCLUDE        '($FORIOSDEF)'                                          ❶

    OPEN (UNIT=1, FILE='REL', STATUS='OLD', SHARED,  &                     ❷
        ORGANIZATION='RELATIVE', ACCESS='DIRECT', FORM='FORMATTED')        ❸

!   Request record to be examined

100 TYPE 10
 10 FORMAT ('$Record number (Ctrl/Z to quit): ')
    READ (*,*, END=999) REC_NUM

!   Get record from file

    READ (1,20, REC=REC_NUM, IOSTAT=STATUS), REC_LEN, RECORD
 20 FORMAT (Q, A)

!   Check I/O status

    IF (STATUS .EQ. 0) THEN
        TYPE *, RECORD(1:REC_LEN)                                          ❹
      ELSE IF (STATUS .EQ. FOR$IOS_ATTACCNON) THEN
        TYPE *,  'Nonexistent record.'
        GOTO 100
      ELSE IF (STATUS .EQ. FOR$IOS_RECNUMOUT) THEN
        TYPE *, 'Record number out of range.'
        GOTO 100
      ELSE IF (STATUS .EQ. FOR$IOS_SPERECLOC) THEN
        TYPE *, 'Record locked by someone else.'                           ❺
        GOTO 100
      ELSE
        CALL ERRSNS (, RMS_STS, RMS_STV,,)
        CALL LIB$SIGNAL (%VAL(RMS_STS), %VAL(RMS_STV))
    ENDIF

!  Request updated record

   TYPE 30
30 FORMAT ('$New Value or CR: ')
   READ (*,20) REC_LEN, RECORD
   IF (REC_LEN .NE. 0) THEN
     WRITE (1,40, REC=REC_NUM, IOSTAT=STATUS) RECORD(1:REC_LEN)
40   FORMAT (A)
     IF (STATUS .NE. 0) THEN
       CALL ERRSNS (, RMS_STS, RMS_STV,,)
       CALL LIB$SIGNAL(%VAL(RMS_STS),%VAL(RMS_STV))
     ENDIF
   ENDIF

!  Loop

    GOTO 100

999 END PROGRAM
```

Sample Use:

```
$ FORTRAN SHAREDFIL
$ LINK SHAREDFIL
$ RUN SHAREDFIL
Record number (Ctrl/Z to quit): 2
MSPIGGY
New Value or CR: FOZZIE
Record number (Ctrl/Z to quit): 1
KERMIT
New Value or CR:
Record number (Ctrl/Z to quit): Ctrl/Z
$
$ RUN SHAREDFIL
Record number (Ctrl/Z to quit): 2            ❺
Record locked by someone else.
Record number (Ctrl/Z to quit): 2
Record locked by someone else.
Record number (Ctrl/Z to quit): 2
FOZZIE
New Value or CR: MSPIGGY
Record number (Ctrl/Z to quit): Ctrl/Z       ❹
$
```

❶ The library module FORIOSDEF must be included to define the symbolic status codes returned by VSI Fortran I/O statements.
❷ This program requires a relative file named REL.DAT.
❸ The SHARED qualifier is used on the OPEN statement to indicate that the file can be shared. Because manual locking was not specified, RMS automatically controls access to the file. Only read and update operations are allowed in this example. No new records can be written to the file.
❹ Once the first process has finished with record #2, the second process can update it.
❺ The second process is not allowed to access record #2 while the first process is accessing it.

F.6. Displaying Data at Terminals

The following example calls SMG routines to format screen output.

No sample run is included for this example because the program requires a video terminal in order to execute properly.

Source Program:

```
!   File: SMGOUTPUT.F90
!
!   This program calls Run-Time Library Screen Management routines
!   to format screen output.

    IMPLICIT INTEGER (KIND=4) (A-Z)
    INCLUDE          '($SMGDEF)'                                    ❶

!   Establish terminal screen as pasteboard

    STATUS = SMG$CREATE_PASTEBOARD (NEW_PID,,,)                     ❷
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
```

```
!   Establish a virtual display region

    STATUS = SMG$CREATE_VIRTUAL_DISPLAY (15,30,DISPLAY_ID,,,)          ❸
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

!   Paste the virtual display to the screen, starting at
!   row 2, column 15

    STATUS = SMG$PASTE_VIRTUAL_DISPLAY(DISPLAY_ID,NEW_PID,2,15)        ❹
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

!   Put a border around the display area

    STATUS = SMG$LABEL_BORDER(DISPLAY_ID,'This is the Border',,,,,)    ❺
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

!   Write text lines to the screen

    STATUS = SMG$PUT_LINE (DISPLAY_ID,' ',,,,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    STATUS = SMG$PUT_LINE (DISPLAY_ID,'Howdy, pardner',2,,,,)          ❻
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    STATUS = SMG$PUT_LINE (DISPLAY_ID,'Double spaced lines...',2,,,,)  ❻
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

    STATUS = SMG$PUT_LINE (DISPLAY_ID,'This line is blinking',2, &     ❼
                           SMG$M_BLINK,0,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    STATUS = SMG$PUT_LINE (DISPLAY_ID,'This line is reverse video',2, & ❼
                           SMG$M_REVERSE,0,,)
    IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    DO I = 1, 5                                                        ❽
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'Single spaced lines...',,,,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
    ENDDO

    END PROGRAM
```

❶ The INCLUDE statement incorporates the $SMGDEF library module from FORSYSDEF.TLB into the source program. This library module contains symbol definitions used by the screen management routines.

❷ The call to SMG$CREATE_PASTEBOARD creates a pasteboard upon which output will be written. The pasteboard ID is returned in the variable NEW_PID.

No value is specified for the output device parameter, so the output device defaults to SYS$OUTPUT. Also, no values are specified for the PB_ROWS or PB_COLS parameters, so the pasteboard is created with the default number of rows and columns. The defaults are the number of rows and the number of columns on the physical screen of the terminal to which SYS$OUTPUT is assigned.

❸ The created virtual display is 15 lines long and 30 columns wide. The virtual display initially contains blanks.

❹ The virtual display is pasted to the pasteboard, with its upper left corner positioned at row 2, column 15 of the pasteboard. Pasting the virtual display to the pasteboard causes all data written to the virtual display to appear on the pasteboard's output device, which is SYS$OUTPUT—the terminal screen.

At this point, nothing appears on the screen because the virtual display contains only blanks. However, because the virtual display is pasted to the pasteboard, the program statements described below cause text to be written to the screen.

❺ A labeled border is written to the virtual display.

❻ Using a call to the RTL routine SMG$PUT_LINE, the text line (“Howdy, pardner” is written to the virtual display.

To specify double spacing, a call to SMG$PUT_LINE displays “Double spaced lines...” by specifying the line-adv (third) argument to SMG$PUT_LINE as 2.

❼ Two subsequent calls to SMG$PUT_LINE specify the SMG$M_BLINK and SMG$M_REVERSE parameters (rendition-set argument) display the double-spaced lines “This line is blinking” as blinking and “This line is reverse video” in reverse video. The parameter mask constants like SMG$M_BLINK are defined in the $SMGDEF library module in FORSYSDEF.TLB.

❽ The program displays single-spaced text by omitting a value for the line-adv argument (third argument) to SMG$PUT_LINE. The DO loop displays the line “Single spaced lines...” five times.

F.7. Creating, Accessing, and Ordering Files

In the following example, each record in a relative file is assigned to a specific cell in that file. On sequential write operations, the records are written to consecutive empty cells. Random write operations place the records into cell numbers as provided by the REC=n parameter.

Source Program:

```
!   File: RELATIVE.F90
!
!   This program demonstrates how to access a relative file
!   randomly. It also performs some I/O status checks.

    IMPLICIT          INTEGER (KIND=4) (A - Z)
    STRUCTURE /EMPLOYEE_STRUC/
      CHARACTER(LEN=5)       ID_NUM
      CHARACTER(LEN=6)       NAME
      CHARACTER(LEN=3)       DEPT
      CHARACTER(LEN=2)       SKILL
      CHARACTER(LEN=4)       SALARY
    END STRUCTURE
    RECORD /EMPLOYEE_STRUC/ EMPLOYEE_REC
    INTEGER (KIND=4) REC_LEN
    INCLUDE   '($FORIOSDEF)'                              ❶

    OPEN (UNIT=1, FILE='REL', STATUS='OLD', ORGANIZATION='RELATIVE',  & ❷
         ACCESS='DIRECT', FORM='UNFORMATTED',RECORDTYPE='VARIABLE')

!  Get records by record number until e-o-f
!  Prompt for record number

100 TYPE 10
 10 FORMAT ('$Record number: ')
    READ (*,*, END=999) REC_NUM                           ❸

!   Read record by record number

    READ (1,REC=REC_NUM,IOSTAT=STATUS) EMPLOYEE_REC
```

```
!   Check I/O status

    IF (STATUS .EQ. 0) THEN
       WRITE (6) EMPLOYEE_REC                          ❹
    ELSE IF (STATUS .EQ. FOR$IOS_ATTACCNON) THEN
       TYPE *,  'Nonexistent record.'
    ELSE IF (STATUS .EQ. FOR$IOS_RECNUMOUT) THEN
       TYPE *, 'Record number out of range.'
    ELSE
       CALL ERRSNS (, RMS_STS, RMS_STV,,)              ❺
       CALL LIB$SIGNAL (%VAL(RMS_STS), %VAL(RMS_STV))
    ENDIF

!   Loop

    GOTO 100
999 END
```

Sample Use:

```
$ FORTRAN RELATIVE
$ LINK RELATIVE
$ RUN RELATIVE
Record number: 7
08001FLANJE119PL1920
Record number: 1
07672ALBEHA210SE2100
Record number: 30
Nonexistent record.
Record number: Ctrl/Z
$
```

❶ The INCLUDE statement defines all Fortran I/O status codes.

❷ The OPEN statement defines the file and record processing characteristics. Although the file organization is specified as relative, RMS would in fact obtain the file organization from an existing file. If the file's organization were not relative, the file OPEN statement would fail.

The file is being opened for unformatted I/O because the data records will be read into a VSI Fortran record (EMPLOYEE_REC), and VSI Fortran does not allow records to be used in formatted I/O.

❸ The READ statement reads the record specified in REC_NUM, rather than the next consecutive record. The status code for the record operation is returned in the variable STATUS.

❹ These statements test the record operation status obtained in comment 3. Note, the status codes returned by RMS and VSI Fortran are not numerically or functionally similar.

❺ RMS status codes actually require two parameters. These values can be obtained using the ERRSNS subroutine.

F.8. Measuring and Improving Performance

This example demonstrates how to adjust the size of the process working set from a program.

Source Program:

```
!   File: ADJUST.F90
```

```
!
!   This program demonstrates how a program can control
!   its working set size using the $ADJWSL system service.

    IMPLICIT      INTEGER (A-Z)
    INCLUDE       '($SYSSRVNAM)'
    INTEGER (KIND=4)    ADJUST_AMT      /0/
    INTEGER (KIND=4)    NEW_LIMIT       /0/

    CALL LIB$INIT_TIMER

    DO ADJUST_AMT= -50,70,10

!   Modify working set limit

      RESULT = SYS$ADJWSL( %VAL(ADJUST_AMT), NEW_LIMIT)     ❶
      IF (.NOT. RESULT) CALL LIB$STOP(%VAL(RESULT))

      TYPE 50, ADJUST_AMT, NEW_LIMIT
 50   FORMAT(' Modify working set by', I4, '   New working set size =', I5)
    END DO
    CALL LIB$SHOW_TIMER
    END PROGRAM
```

❶ The call to SYS$ADJWSL call uses a function invocation.

Sample Use:

```
$ SET WORKING_SET/NOADJUST                    ❶
$ SHOW WORKING_SET
  Working Set      /Limit=2000  /Quota=4000  /Extent=98304
  Adjustment disabled   Authorized Quota=4000  Authorized Extent=98304

  Working Set (8Kb pages) /Limit=125  /Quota=250  /Extent=6144
                          Authorized Quota=250  Authorized Extent=6144
$ FORTRAN ADJUST
$ LINK ADJUST

$ RUN ADJUST
Modify working set by -50    New working set size = 1936          ❷
Modify working set by -40    New working set size = 1888
Modify working set by -30    New working set size = 1856
Modify working set by -20    New working set size = 1824
Modify working set by -10    New working set size = 1808
Modify working set by   0    New working set size = 1808
Modify working set by  10    New working set size = 1824
Modify working set by  20    New working set size = 1856
Modify working set by  30    New working set size = 1888
Modify working set by  40    New working set size = 1936
Modify working set by  50    New working set size = 2000
Modify working set by  60    New working set size = 2064
Modify working set by  70    New working set size = 2144
ELAPSED:  0 00:00:00.01  CPU: 0:00:00.01  BUFIO: 13  DIRIO: 0  FAULTS: 24
$
```

❶ The DCL SHOW WORKING_SET command displays the current working set limit and the maximum quota.

❷ The SYS$ADJWSL is used to increase or decrease the number of pages in the process working set.

The program cannot decrease the working set limit beneath the minimum established by the operating system, nor can the process working set be expanded beyond the authorized quota.

F.9. Accessing Help Libraries

The following example demonstrates how to obtain text from a help library. After the initial help request has been satisfied, the user is prompted and can request additional information.

Source Program:

```
!   File: HELPOUT.F90
!
!   This program satisfies an initial help request and enters interactive
!   HELP mode.  The library used is SYS$HELP:HELPLIB.HLB.

    IMPLICIT  INTEGER (KIND=4) (A - Z)
    CHARACTER(LEN=32)   KEY
    EXTERNAL       LIB$PUT_OUTPUT,LIB$GET_INPUT          ❶

!   Request a HELP key

    WRITE (6,200)
200 FORMAT(1X,'What Topic would you like HELP with? ',$)
    READ (5,100) KEY
100 FORMAT (A32)

!   Locate and print the help text

    STATUS = LBR$OUTPUT_HELP(LIB$PUT_OUTPUT,,KEY,   &     ❷
                             'HELPLIB',,LIB$GET_INPUT)
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    END PROGRAM
```

Sample Use:

```
$ FORTRAN HELPOUT
$ LINK HELPOUT
$ RUN HELPOUT
What topic would you like HELP with? TYPE
TYPE
    Displays the contents of a file or a group of files on the
    current output device.

      Format:
          TYPE file-spec[,...]
      Additional information available:
      Parameters  Qualifiers
     /BACKUP    /BEFORE    /BY_OWNER  /CONFIRM   /CONTINUOUS           /
CREATED
     /EXACT     /EXCLUDE   /EXPIRED   /HEADER    /HIGHLIGHT /MODIFIED  /
OUTPUT
     /PAGE      /SEARCH    /SINCE     /TAIL      /WRAP
     Examples
TYPE Subtopic? /HIGHLIGHT
TYPE
```

```
  /HIGHLIGHT

          /HIGHLIGHT[=keyword]
          /NOHIGHLIGHT (default)

    Use with the /PAGE=SAVE and /SEARCH qualifiers to specify the
    type of highlighting you want when a search string is found. When
    a string is found, the entire line is highlighted. You can use
    the following keywords: BOLD, BLINK, REVERSE, and UNDERLINE. BOLD
    is the default highlighting.

TYPE Subtopic? Ctrl/Z
$
```

❶ To pass the address of LIB$PUT_OUTPUT and LIB$GET_INPUT, they must be declared as EXTERNAL. You can supply your own routines for handling input and output.

❷ The address of an output routine is a required argument. When requesting prompting mode, the default mode, an input routine must be specified.

F.10. Creating and Managing Other Processes

The following example demonstrates how a created process can use the SYS$GETJPIW system service to obtain the PID of its creator process. It also shows how to set up an item list to translate a logical name recursively.

Source Program:

```
!   File: GETJPI.F90
!   This program demonstrates process creation and control.
!   It creates a subprocess then hibernates until the subprocess wakes it.

    IMPLICIT        INTEGER (KIND=4) (A - Z)
    INCLUDE         '($SSDEF)'
    INCLUDE         '($LNMDEF)'
    INCLUDE         '($SYSSRVNAM)'
    CHARACTER(LEN=255)    TERMINAL       /'SYS$OUTPUT'/
    CHARACTER(LEN=9)      FILE_NAME      /'GETJPISUB'/
    CHARACTER(LEN=5)      SUB_NAME       /'OSCAR'/
    INTEGER (KIND=4)      PROCESS_ID     /0/
    CHARACTER(LEN=17)     TABNAM         /'LNM$PROCESS_TABLE'/
    CHARACTER(LEN=255)    RET_STRING
    CHARACTER(LEN=2)      ESC_NULL
    INTEGER (KIND=4)      RET_ATTRIB
    INTEGER (KIND=4)      RET_LENGTH      /10/
    STRUCTURE /ITMLST3_3ITEMS/
      STRUCTURE    ITEM(3)
        INTEGER (KIND=2)    BUFFER_LENGTH
        INTEGER (KIND=2)    CODE
        INTEGER (KIND=4)    BUFFER_ADDRESS
        INTEGER (KIND=4)    RETLEN_ADDRESS
      END STRUCTURE
      INTEGER (KIND=4)      END_OF_LIST
    END STRUCTURE
    RECORD /ITMLST3_3ITEMS/  TRNLST

!   Translate SYS$OUTPUT
!   Set up TRNLST, the item list for $TRNLNM
```

```
    TRNLST.ITEM(1).CODE = LNM$_STRING
    TRNLST.ITEM(1).BUFFER_LENGTH = 255
    TRNLST.ITEM(1).BUFFER_ADDRESS = %LOC(RET_STRING)
    TRNLST.ITEM(1).RETLEN_ADDRESS = 0

    TRNLST.ITEM(2).CODE = LNM$_ATTRIBUTES
    TRNLST.ITEM(2).BUFFER_LENGTH = 4
    TRNLST.ITEM(2).BUFFER_ADDRESS = %LOC(RET_ATTRIB)
    TRNLST.ITEM(2).RETLEN_ADDRESS = 0

    TRNLST.ITEM(3).CODE = LNM$_LENGTH
    TRNLST.ITEM(3).BUFFER_LENGTH = 4
    TRNLST.ITEM(3).BUFFER_ADDRESS = %LOC(RET_LENGTH)
    TRNLST.ITEM(3).RETLEN_ADDRESS = 0

    TRNLST.END_OF_LIST = 0

!   Translate SYS$OUTPUT

100 STATUS = SYS$TRNLNM (,TABNAM,TERMINAL(1:RET_LENGTH),,TRNLST)
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    IF (IAND(LNM$M_TERMINAL, RET_ATTRIB).EQ. 0) THEN
        TERMINAL = RET_STRING(1:RET_LENGTH)
        GO TO 100
    ENDIF

!   Check if process permanent file

    ESC_NULL(1:2) = char('1B'x)//char('00'x)
    IF (RET_STRING(1:2) .EQ. ESC_NULL) THEN
         RET_STRING = RET_STRING(5:RET_LENGTH)
         RET_LENGTH = RET_LENGTH - 4
    ENDIF

!   Create the subprocess

    STATUS = SYS$CREPRC (PROCESS_ID, FILE_NAME,,   &          ❶
                         RET_STRING(1:RET_LENGTH),,,, &
                         SUB_NAME,%VAL(4),,,)
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    TYPE 10, PROCESS_ID
10  FORMAT (' PID of subprocess OSCAR is ', Z)

!   Wait for wakeup by subprocess

    STATUS = SYS$HIBER ()                                     ❷
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

    TYPE *, 'GETJPI has been awakened.'
    END PROGRAM

!   File: GETJPISUB.F90
!   This separately compiled program is run in the subprocess OSCAR
!   which is created by GETJPI.  It  obtains its creator's PID and then
!   wakes it.

    IMPLICIT       INTEGER (KIND=4) (A - Z)                   ❸
```

```
    INCLUDE         '($JPIDEF)'
    INCLUDE         '($SYSSRVNAM)'
    STRUCTURE /GETJPI_IOSB/
      INTEGER(KIND=4)  STATUS
      INTEGER(KIND=4)  %FILL
    END STRUCTURE
    RECORD /GETJPI_IOSB/  IOSB
    STRUCTURE /ITMLST3_1ITEM/
      STRUCTURE    ITEM
        INTEGER (KIND=2)    BUFFER_LENGTH
        INTEGER (KIND=2)    CODE
        INTEGER (KIND=4)    BUFFER_ADDRESS
        INTEGER (KIND=4)    RETLEN_ADDRESS
      END STRUCTURE
      INTEGER (KIND=4)      END_OF_LIST
    END STRUCTURE
    RECORD /ITMLST3_1ITEM/  JPI_LIST

!   Set up buffer address for GETJPI

    JPI_LIST.ITEM.CODE = JPI$_OWNER                          ❹
    JPI_LIST.ITEM.BUFFER_LENGTH = 4
    JPI_LIST.ITEM.BUFFER_ADDRESS = %LOC(OWNER_PID)
    JPI_LIST.ITEM.RETLEN_ADDRESS = 0

!   Get PID of creator

    STATUS = SYS$GETJPIW (%VAL(1),,, JPI_LIST,IOSB,,)        ❺
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
    IF (.NOT. IOSB.STATUS) CALL LIB$STOP (%VAL(IOSB.STATUS))

!   Wake creator

    TYPE *, 'OSCAR is waking creator.'
    STATUS = SYS$WAKE (OWNER_PID,)
    IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

    END PROGRAM
```

Sample Use:

```
$ FORTRAN GETJPI,GETJPISUB
$ LINK GETJPI
$ LINK GETJPISUB
$ RUN GETJPI
PID of subprocess OSCAR is 2120028A
OSCAR is waking creator.
GETJPI has been awakened.
```

❶ The subprocess is created using SYS$CREPRC.
❷ The process hibernates.
❸ The INCLUDE statement defines the value of all JPI$ codes including JPI$_OWNER. JPI$_OWNER is the item code which requests the PID of the owner process. If there is no owner process (that is, if the process about which information is requested is a detached process), the system service $GETJPIW returns a PID of zero.

❹ Because of the item code JPI$_OWNER in the item list, $GETJPIW returns the PID of the owner of the process about which information is requested. If the item code were JPI$_PID, $GETJPIW would return the PID of the process about which information is requested.

Because the default value of 0 is used for arguments PIDADR and PRCNAM, the process about which information is requested is the requesting process, namely, OSCAR.

❺ The item list for SYS$GETJPIW consists of a single item descriptor followed by a zero longword.