VMS Software

VSI Pascal User Manual

Document Number: DO-DPASUM-01A

Publication Date: April 2024

Operating System and Version: VSI OpenVMS x86-64 Version 9.2-1 or higher
VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI Pascal Version 6.3 for OpenVMS x86-64
VSI Pascal Version 6.2 for OpenVMS I64
VSI Pascal Version 6.2 for OpenVMS Alpha

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

VSI Pascal User Manual

VMS Software

Copyright © 2024 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

Intel, Itanium and IA64 are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java, the coffee cup logo, and all Java based marks are trademarks or registered trademarks of Oracle Corporation in the United States or other countries.

Kerberos is a trademark of the Massachusetts Institute of Technology.

Microsoft, Windows, Windows-NT and Microsoft XP are U.S. registered trademarks of Microsoft Corporation. Microsoft Vista is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Motif is a registered trademark of The Open Group.

UNIX is a registered trademark of The Open Group.

Preface ..... **vii**
1. About VSI vii
2. Intended Audience vii
3. Document Structure vii
4. Related Documents viii
5. OpenVMS Documentation viii
6. VSI Encourages Your Comments viii
7. Conventions viii

Chapter 1. Compiling VSI Pascal for OpenVMS Programs **1**
1.1. PASCAL Command 1
1.1.1. PASCAL Command Examples 2
1.1.2. PASCAL Qualifiers 2
1.1.3. Contents of the Compilation Listing File 19
1.1.3.1. Source Code 19
1.1.3.2. Cross-Reference Section 19
1.1.3.3. Machine Code Section 20
1.1.3.4. Structured Layout Section 20
1.1.3.5. Compilation Statistics 20
1.1.4. Text Libraries 20
1.1.4.1. Using the %INCLUDE Directive for Text Libraries 21
1.1.4.2. Specifying Text Libraries on the Command Line 21
1.1.4.3. Defining Default Libraries 22
1.2. LINK Command 22
1.2.1. LINK Command Examples 23
1.2.2. LINK Qualifiers 24
1.2.3. Object Module Libraries 26
1.3. RUN Command 26
1.4. Error Messages 27

Chapter 2. Separate Compilation **29**
2.1. ENVIRONMENT, HIDDEN, and INHERIT Attributes 29
2.2. Interfaces and Implementations 33
2.3. Data Models 36
2.4. Separate Compilation Examples 39

Chapter 3. Program Correctness, Optimization, and Efficiency **43**
3.1. Compiler Optimizations 43
3.1.1. Compile-Time Evaluation of Constants 44
3.1.2. Elimination of Common Subexpressions 45
3.1.3. Elimination of Unreachable Code 46
3.1.4. Code Hoisting from Structured Statements 46
3.1.5. Inline Code Expansion for Predeclared Functions 47
3.1.6. Inline Code Expansion for User-Declared Routines 47
3.1.7. Operation Rearrangement 47
3.1.8. Partial Evaluation of Logical Expressions 47
3.1.9. Value Propagation 48
3.1.10. Strength Reduction (VSI OpenVMS I64 and VSI OpenVMS Alpha systems) 49
3.1.11. Split Lifetime Analysis 49
3.1.12. Code Scheduling 49
3.1.13. Loop Unrolling 49
3.1.14. Alignment of Compiler-Generated Labels 50
3.1.15. Error Reduction Through Optimization 50

3.1.16. Processor Selection and Tuning (VSI OpenVMS Alpha systems) 50
3.1.17. Compiling for Optimal Performance 51
3.2. Programming Considerations 51
3.3. Implementation-Dependent Behavior 52
3.3.1. Subexpression Evaluation Order 53
3.3.2. MAXINT and MAXINT64 Predeclared Constants 53
3.3.3. Pointer References 53
3.3.4. Variant Records 54
3.3.5. Atomicity, Granularity, Volatility, and Write Ordering 55
3.3.6. Debugging Considerations 56
Chapter 4. Programming Tools 59
4.1. Debugger Support for VSI Pascal for OpenVMS 59
4.2. Language-Sensitive Editor/Source Code Analyzer Support for VSI Pascal for OpenVMS 60
4.2.1. Programming Language Placeholders and Tokens 60
4.2.2. Placeholder and Design Comment Processing 60
4.2.3. LSE and SCA Examples 61
4.3. Accessing CDD/Repository from VSI Pascal for OpenVMS 62
4.3.1. Equivalent VSI Pascal for OpenVMS and CDDL Data Types 63
4.3.2. CDD/Repository Example 65
Chapter 5. Calling Conventions 67
5.1. VSI OpenVMS Calling Standard 67
5.1.1. Parameter Lists 67
5.1.2. Function Return Values 68
5.1.3. Contents of the Call Stack 68
5.1.4. Unbound Routines 70
5.2. Parameter-Passing Semantics 71
5.3. Parameter-Passing Mechanisms 71
5.3.1. By Immediate Value 72
5.3.2. By Reference 72
5.3.3. By Descriptor 73
5.3.3.1. CLASS_S Attribute 75
5.3.3.2. CLASS_A and CLASS_NCA Attributes 75
5.3.3.3. %STDESCR Mechanism Specifier 75
5.3.3.4. %DESCR Mechanism Specifier 76
5.3.4. Summary of Passing Mechanisms and Passing Semantics 77
5.4. Passing Parameters between VSI Pascal for OpenVMS and Other Languages 77
5.4.1. Parameter Mechanisms Versus Parameter Semantics 78
5.4.2. Passing Nonroutine Parameters between VSI Pascal for OpenVMS and Other Languages 78
5.4.3. Passing Routine Parameters between VSI Pascal and Other Languages 80
Chapter 6. Programming on VSI OpenVMS Systems 83
6.1. Using System Definitions Files 83
6.2. Declaring System Routines 85
6.2.1. Methods Used to Obtain VSI OpenVMS Data Types 86
6.2.2. Methods Used to Obtain Access Methods 86
6.2.3. Methods Used to Obtain Passing Mechanisms 87
6.2.4. Data Structure Parameters 88
6.2.5. Default Parameters 89
6.2.6. Arbitrary Length Parameter Lists 90
6.3. Calling System Routines 91

6.4. Using Attributes 91
6.5. Using Item Lists 92
6.6. Using Foreign Mechanism Specifiers on Actual Parameters 93
6.7. Using 64-Bit Pointer Types 94
6.7.1. Pascal Language Features Not Supported with 64-Bit Pointers 94
6.7.2. Using 64-Bit Pointers with System Definition Files 95
Chapter 7. Input and Output Processing 99
7.1. Environment I/O Support 99
7.1.1. Indexed Files 99
7.1.2. VSI OpenVMS Components and RMS Records 100
7.1.3. Count Fields for Variable-Length Components 100
7.1.4. Variable-Length with Fixed-Length Control Field (VFC) Component Format 100
7.1.5. Random Access by Record File Address (RFA) 100
7.1.6. OPEN Procedure 100
7.1.6.1. OPEN Defaults 101
7.1.6.2. OPEN and RMS Data Structures 101
7.1.7. Default Line Limits 105
7.2. User-Action Functions 105
7.3. File Sharing 108
7.4. Record Locking 109
Chapter 8. Error Processing and Condition Handling 111
8.1. Condition Handling Terms 111
8.2. Overview of Condition Handling 112
8.2.1. Condition Signals 112
8.2.2. Handler Responses 112
8.3. Writing Condition Handlers 113
8.3.1. Establishing and Removing Handlers 113
8.3.2. Declaring Parameters for Condition Handlers 114
8.3.3. Handler Function Return Values 115
8.3.4. Condition Values and Symbols 116
8.3.5. Using Condition Handlers that Return SS$_CONTINUE 116
8.4. Examples of Condition Handlers 117
Chapter 9. Migrating Between Different Architectures 123
9.1. Sharing Environment Files Across Platforms 123
9.2. Default Size for Enumerated Types and Booleans 123
9.3. Default Data Layout for Unpacked Arrays and Records 123
9.4. Overflow Checking 123
9.5. Bound Procedure Values 124
9.6. Different Descriptor Classes for Conformant Array Parameters 124
9.7. Data Layout and Conversion 124
9.7.1. Natural Alignment, VAX Alignment, and Enumeration Sizes 125
9.7.2. VSI Pascal for OpenVMS Features Affecting Data Alignment and Size 126
9.7.3. Optimal Record Layout 126
9.7.4. Optimal Data Size 128
9.7.5. Converting Existing Records 128
9.7.6. Applications with No External Data Dependencies 128
9.7.7. Applications with External Data Dependencies 129
Appendix A. Errors Returned by STATUS and STATUSV Functions 133
Appendix B. Entry Points to VSI Pascal for OpenVMS Utilities 137
B.1. PAS$FAB (f) 137

B.2. PAS$RAB (f) 137
B.3. PAS$MARK2 (s) 138
B.4. PAS$RELEASE2 (p) 138

Appendix C. Diagnostic Messages 141

C.1. Compiler Diagnostics 141
C.2. Run-Time Diagnostics 205

Preface

This manual describes selected programming tasks using the VSI Pascal programming language. It contains information on using some VSI Pascal language elements in combination, and it provides examples of how to improve programming efficiency.

You can use the information in this manual to write programs or modules for the OpenVMS operating system. If you need to write VSI Pascal programs or modules that must be compiled by another compiler, see the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for a checklist of language extensions not included in the Pascal standard. The *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] also provides information on the Pascal standard.

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 is intended for experienced applications programmers with a firm understanding of the Pascal language. Some familiarity with the operating system is helpful.

3. Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 provides information on compiling programs, linking programs, running programs, and using text and object-module libraries.
- Chapter 2 describes the use of separately compiled modules.
- Chapter 3 describes programming techniques that improve the efficiency of compilation and execution.
- Chapter 4 provides information on tools that you may want to use to develop VSI Pascal programs.
- Chapter 5 provides information on the *VSI OpenVMS Calling Standard* as applied to VSI Pascal programs.
- Chapter 6 provides information on VSI Pascal system definitions files, and declaring and calling system routines.
- Chapter 7 provides information on the relationship between VSI Pascal input and output, and the underlying OpenVMS VAX Record Management Services (RMS) constructs.
- Chapter 8 provides information on error processing and writing condition handlers.
- Chapter 9 provides information on migrating between different architectures running OpenVMS.
- Appendix A provides a list of possible error values returned by the STATUS and STATUSV functions.

- Appendix B provides a list of entry points to utilities in the OpenVMS Run-Time Library that can be called as external routines by an VSI Pascal program.
- Appendix C provides descriptions of diagnostic messages that can be generated by an VSI Pascal program at compile time and run time.

4. Related Documents

The following manuals are also part of the VSI Pascal documentation set:

- *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] — Provides information on the syntax and semantics of the VSI Pascal programming language, including information about the alignment, allocation, and internal representation of data types supported by VSI Pascal.
- *VSI Pascal Installation Guide* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-installation-guide/] — Provides information on how to install VSI Pascal on your operating system.

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 conventions are used in this manual:

Convention	Meaning
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.

Convention	Meaning
( )	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.
`monospace`	Bold monospace type indicates a command line, command verb, or a qualifier.
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.

Chapter 1. Compiling VSI Pascal for OpenVMS Programs

After you use a text editor to write code to a file, you use Digital Command Language (DCL) commands to compile modules and programs, and to link and run programs. This chapter covers the **PASCAL**, **LINK**, and **RUN** commands, as well as describes possible error messages.

For information on DCL syntax, see **HELP** or the VSI OpenVMS documentation set.

1.1. PASCAL Command

The **PASCAL** command invokes the VSI Pascal compiler, which verifies program source statements, issues error messages, and generates and groups machine language instructions into an object module for the linker.

The default for compiler output files (object modules) is the .OBJ file type.

The Pascal command format is as follows:

```
PASCAL [[{/command-qualifier} ...]]
       {file-spec[[{/file-qualifier} ...]]} {+|,} ...
/qualifier [[= {file-spec | identifier | (identifier, ...) | n |
           name=identifier, … | directory, …} ]]
```

/command-qualifier

The name of a qualifier that indicates special processing to be performed by the compiler on all files listed.

file-spec

The name of one of the following:

- The input source file that contains the program or module to be compiled. If you separate multiple source file specifications with commas, the programs are compiled separately. If you separate the file specifications with plus signs, the files are concatenated and compiled as one program.

  The default file type for an input file is either .PAS (source file) or .TLB (text-library module).

- The output file used only with the **/ANALYSIS_DATA**, **/ENVIRONMENT**, **/LIST**, **/OBJECT**, or **/DIAGNOSTICS** qualifiers.

/file-qualifier

The name of a qualifier that indicates special processing to be performed by the compiler on the files to which the qualifier is attached.

identifier

The name of one or more options that modify the **/ALIGN**, **/CHECK**, **/DEBUG**, **/DESIGN**, **/FLOAT**, **/OPTIMIZE**, **/SHOW**, **/STANDARD**, **/TERMINAL**, and **/USAGE** qualifiers.

n

The value of an integer constant. When used with the **/ERROR_LIMIT** qualifier, this parameter indicates the maximum number of errors to be detected before compilation ceases. When used with the **/OPTIMIZATION=LEVEL=*n*** qualifier, this parameter indicates a specific level of optimization.

name=value

The definition of a constant or name with the specified value when used with the **/CONSTANT** qualifier.

directory

The input directory that contains the include file, environment file, or text library processed by a %INCLUDE directive or [INHERIT] attribute when used with the **/INCLUDE** qualifier.

1.1.1. PASCAL Command Examples

This section contains examples of PASCAL command lines.

```
$ PASCAL/LIST [DIR]M
```

The source file M.PAS in directory [DIR] is compiled, producing an object file named M.OBJ and a listing file named M.LIS. The compiler places the object and listing files in the default directory.

```
$ PASCAL/LIST A, B, C
```

Source files A.PAS, B.PAS, and C.PAS are compiled as separate files, producing object files named A.OBJ, B.OBJ, and C.OBJ, and listing files named A.LIS, B.LIS, and C.LIS.

```
$ PASCAL X + Y + Z
```

Source files X.PAS, Y.PAS, and Z.PAS are concatenated and compiled as one file producing an object file named X.OBJ. By default, batch mode produces a listing file, which takes its name from the name of the first file on the command line. In this example, the name of the listing file would be X.LIS.

```
$ PASCAL A, B, C+D/LIST, F
```

When a qualifier follows the file specification, it applies only to the file immediately preceding it. Files concatenated with plus signs are considered one file. This command line produces four object files, A.OBJ, B.OBJ, C.OBJ, and F.OBJ, and one listing file, D.LIS.

```
$ PASCAL A + CIRC/NOOBJECT + X
```

This command completely suppresses the object file; that is, source files A.PAS, CIRC.PAS, and X.PAS are concatenated and compiled, but no object file is produced.

1.1.2. PASCAL Qualifiers

This section describes the command and file qualifiers that you can use when compiling code.

/ALIGN= *option*

Controls the default alignment rules. Note that specifying the ALIGN attribute overrides any value that was previously specified for the **/ALIGN** qualifier.

Table 1.1 lists the options for the **/ALIGN** qualifier..

Table 1.1. /ALIGN Qualifier Options

Option	Action	Default Information
NATURAL [1]	Uses natural alignment when positioning record fields or array components. Natural alignment is when a record field or an array component is positioned on a boundary based on its size. For example, 32-bit integers are aligned on the nearest 32-bit boundary.	Default on all OpenVMS systems if **/ALIGN** is not specified.
VAX	Uses byte alignment when positioning record fields or array components. Record fields or array components larger than 32 bits are positioned on the nearest byte boundary.	

[1]Previous versions of VSI Pascal used ALPHA_AXP for this option. The NATURAL option is now the recommended spelling for the same behavior. The ALPHA_AXP option will continue to be recognized for compatibility with old command lines.

/[NO]ANALYSIS_DATA
/NOANALYSIS_DATA (default)

Creates a file containing source code analysis information. If you omit the file specification, the analysis file defaults to the name of your source file with a .ANA file type. The source code analysis file is used with products such as the Language-Sensitive Editor/Source Code Analyzer (LSE/SCA).

/[NO]ARCHITECTURE
/ARCHITECTURE=GENERIC (default) (VSI OpenVMS Alpha systems only, ignored on other systems)

Specifies which version of the Alpha architecture instructions should be generated for. All Alpha processors implement a core set of instructions and, in some cases, the following extensions: BWX (byte- and word- manipulation instructions) and MAX (multimedia instructions). See the *Alpha Architecture Reference Manual* for additional information. Table 1.2 lists the available options:

Table 1.2. /ARCHITECTURE Qualifier Options

Option	Action
GENERIC (default)	Generate instructions that are appropriate for all Alpha processors.
HOST	Generate instructions for the processor on which the compiler is running (for example, EV56 instructions on an EV56 processor, and EV4 instructions on an EV4 processor).
EV4, EV5	Generate instructions for the EV4 processor (21064, 20164A, 21066, and 21068 chips) and EV5 processor (some 21164 chips). (Note that the EV5 and EV56 processors both have the same chip number - 21164.)
EV56	Generate instructions for EV56 processors (some 21164 chips). This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX extension. Applications compiled with this option may incur emulation overhead on EV4 and EV5 processors.

Option	Action
PCA56	Generate instructions for PCA56 processors (21164PC chips). This option permits the compiler to generate any EV4 instruction, plus any instructions contained in the BWX and MAX extensions. However, VSI Pascal does not generate any of the instructions in the MAX (multimedia) extension to the Alpha architecture.
EV6	Generate instructions for EV6 processors (21264 chips). This option permits the compiler to generate any EV4 instruction, any instruction contained in the BWX and MAX extensions, plus any instructions added for the EV6 chip. These new instructions include a floating-point square root instruction (SQRT), integer/floating-point register transfer instructions, and additional instructions to identify extensions and processor groups. Applications compiled with this option may incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.
EV6	Generate instructions for EV7 processors (21364 chips). This option permits the compiler to generate any EV67 instruction. There are no additional instructions available on the EV7 processor, but the compiler does have different instruction scheduling and prefetch rules for tuning code for the EV7. Applications compiled with this option may incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.
EV67, **EV68**	Generate instructions for EV67 and EV68 processors (21264A chips). This option permits the compiler to generate any EV6 instruction, plus the new bit count instructions (CTLZ, CTPOP, and CTTZ). However, VSI Pascal does not currently generate any of the new bit count instructions, and the EV67 and EV68 have identical instruction scheduling models, so the EV67 and EV68 are essentially identical to the EV6. Applications compiled with this option may incur emulation overhead on EV4, EV5, EV56, and PCA56 processors.

Beginning with VSI OpenVMS Alpha V7.1 and continuing with subsequent versions, the operating system includes an instruction emulator. This capability allows any Alpha chip to execute and produce correct results from Alpha instructions, regardless of whether some of the instructions are not implemented on the chip. Applications using emulated instructions will run correctly, but may incur significant emulation overhead at run time.

/ASSUME=*option*
/ASSUME=ACCURACY_SENSITIVE (default)

Specifies whether certain code transformations that affect floating-point operations are allowed. These changes may or may not affect the accuracy of the program's results. You cannot specify **/ASSUME** without options.

If you specify **NOACCURACY_SENSITIVE**, the compiler is free to reorder floating-point operations based on algebraic identities (inverses, associativity, and distribution). This allows the

compiler to move additional floating-point operations outside of loops or reduce or remove floating-point operations totally, thereby improving performance.

The default, **ACCURACY_SENSITIVE**, directs the compiler to avoid certain floating-point transformations that might slightly affect the program's accuracy.

Normally, the compiler assumes that pointer variables are initialized by a call to the NEW predeclared routine. The memory returned by NEW is at least quadword aligned. The compiler can take advantage of that alignment to generate better code. However, if the program initializes the pointer by some other means such as IADDRESS or typecasting with values that are not quadword aligned, then the generated code may produce alignment faults. While the alignment faults are silently handled by OpenVMS, the resulting performance loss might be significant.

By specifying **BYTE_ALIGNED_POINTERS**, the compiler will generate slightly slower code to fetch the value. However, compared to the overhead of correcting the alignment faults, this additional overhead is very small.

The preferred solution is to ensure that all pointers contain quadword aligned addresses and use the default of **NOBYTE_ALIGNED_POINTERS**.

Table 1.3 lists the available options, their corresponding actions, and default information.

Table 1.3. /ASSUME Qualifier Options

Option	Action
[NO]ACCURACY_SENSITIVE	Specifies whether certain code transformations that affect floating-point operations are allowed.
[NO]BYTE_ALIGNED_POINTERS (OpenVMS Alpha and OpenVMS I64 systems)	Specifies that the compiler should assume that all pointers point to memory that is only aligned on byte boundaries.
[NO]LONG_CALLS (OpenVMS I64 systems)	Specifies that the compiler should generate the longer 'brl.call' instruction for calls to external routines. This option is only needed when the linker is unable to resolve a PCREL21B relocation for the default 'br.call' instruction.
[NO]REDUCED_RELOCATIONS (OpenVMS I64 systems)	Specifies that the compiler should generate additional instructions to reduce the number of address constants requested from the linker. This option is only needed when the linker is unable to generate all the requested address constants.

/CDD_QUAD_TYPE=*option*
/CDD_QUAD_TYPE=EMPTY_RECORD (default)

Directs the compiler how the %DICTIONARY directive translates quadword and octaword sized items from the CDD Dictionary.

Table 1.4 lists the available options, their corresponding actions, and default information.

Table 1.4. /CDD_QUAD_TYPE Qualifier Options

Option	Action
EMPTY_RECORD	The compiler will translate quadword and octaword sized items (including both CDD date/time datatypes) into "[BYTE(*n*)]

Option	Action
	RECORD END", where "*n*" is 8 or 16. This syntax reserves the appropriate amount of memory for the item, but does not provide any direct method to fetch or store the item. Programs must use explicit typecasts to properly manipulate the empty records. This is the default and is how all prior compilers have translated quadword and octaword sized items.
INTEGER64	The compiler will translate signed quadwords (including both CDD date/time datatypes) into INTEGER64 and unsigned quadwords into UNSIGNED64. Octaword values are still translated into empty records as described above.
RDML_QUAD_TYPE	The compiler will translate quadword sized items (including both CDD date/time datatypes) into "[BYTE(8),UNSAFE] RECORD L0:UNSIGNED; L1:INTEGER END" and octaword sized items into "[BYTE(16),UNSAFE] RECORD L0,L1,L2:UNSIGNED; L3:INTEGER END". This matches the behavior of the RDML preprocessor.

`/[NO]CHECK`
`/CHECK=(BOUNDS,DECLARATIONS)` (default)

Directs the compiler to generate code to perform run-time checks. A single identifier or a list of identifiers enclosed in parentheses can follow **`/CHECK`**; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to check.

The system issues an error message and normally terminates execution if any of the conditions in the options list occur. Table 1.5 lists the available checking options, their corresponding actions, and their negations.

Table 1.5. `/CHECK` Qualifier Options

Option	Action
ALL	Generates checking code for all options.
NONE	Suppresses all checking code.
[NO]BOUNDS	Verifies that an index expression is within the bounds of an array's index type, that character-string sizes are compatible with the operations being performed, and that schemata are compatible.
[NO]CASE_SELECTORS	Verifies that the value of a case selector is contained in the corresponding case-label list.
[NO]DECLARATIONS	Verifies that schema definitions yield valid types and that uses of GOTO from one block to an enclosing block are correct. Also controls whether the ASSERT statement is processed.
[NO]OVERFLOW	Verifies that the result of an integer computation does not exceed the machine representation.
[NO]POINTERS	Verifies that the value of a pointer variable is not NIL.
[NO]SUBRANGE	Verifies that values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatible with a set variable; verifies that MOD operates on positive numbers.

The **BOUNDS** and **DECLARATIONS** options are the only checking options enabled by default. The **/CHECK** qualifier without options is equivalent to **/CHECK=ALL**. The negation **/NOCHECK** is equivalent to **/CHECK=NONE**.

The CHECK attribute in the source program or module overrides the **/CHECK** qualifier on the command line.

/CONSTANT=(*name=value, ...*)

The **/CONSTANT** qualifier allows a limited set of Pascal constants to be defined from the command line. This capability can be used to augment the conditional-compilation facility.

name is the name of a Pascal constant to create. You cannot define any predeclared Pascal name by the command line.

value can be one of the following:

- integer-literal
- –integer-literal
- TRUE
- FALSE
- "string-literal"
- 'string-literal'

Nonbase-10 integer literals are not supported on the command line. For example:

```
$ PASCAL/CONSTANT=(DEBUG=TRUE,MAXSIZE=10,OFFSET=-10,IDENT="V1.0")
```

Note that the definition of **/CONSTANT** is such that DCL does not remove any double-quote characters used in the *name=value* clauses. All characters are literally passed to the compiler for processing. This behavior is slightly different from the usual behavior of putting items in double quotes on DCL commands to preserve the case, but not to pass the double quotes to the target application.

This definition of **/CONSTANT** allows you to define Pascal string literals with embedded single quotes and for DCL symbol substitutions. For example,

```
$ PASCAL/CONSTANT=MSG="Special compile run for Monday"
$ IDENT = "V1.0"
$ PASCAL/CONSTANT=MSG="''IDENT'"
```

Inserting double-quote characters and inserting adjacent single-quote characters can be accomplished by using the \ escape character allowed in VSI Pascal double-quoted string constants.

By using the \' single-quote escape character, you can insert adjacent single quotes without DCL interpreting it as a symbol substitution, as shown in the following example:

```
$ PASCAL/CONSTANT=MSG="String with 2 \'\' single quote characters"
```

Do not use \" to insert a double-quote character into the string literal, as DCL will interpret the double quote as the end of the string. Instead, use the \x22 character literal (16#22 is the ASCII code for the double-quote character) to insert a double-quote character into the string literal without DCL interpreting it as the end of the string. This is shown in the following example:

```
$ PASCAL/CONSTANT=MSG="String with a \x22 double-quote character"
```

To use a single-quote string literal with **/CONSTANT**, enclose the entire *name=value* clause in double quotes to prevent DCL from trying to perform symbol substitution when it sees the single-quote character. For example:

```
$ PASCAL/CONSTANT="MSG='Single-quoted literal'"
```

In this case, the double quotes are discarded by the compiler, and the single-quoted string literal is processed. However, using double-quoted literals with **/CONSTANT** is easier and more flexible.

The extended-string syntax for string literals is not supported on the command line. The extended-string syntax is as follows:

```
{'printing-string'({constant-expression},...)}...
{"printing-string"({constant-expression},...)}...
```

These extended-string literals are constant expressions, not simple literals.

To insert nonprintable characters into a string literal from the command line, you can use a double-quoted string literal and the \xnn escape sequence.

/[NO]CROSS_REFERENCE
/NOCROSS_REFERENCE (default)

Produces a cross-reference section within the listing file. The compiler ignores this qualifier if you do not also specify **/LIST** on the same command line.

/[NO]DEBUG
/DEBUG=TRACEBACK (default)

Specifies that the compiler is to generate information for use by the debugger and the run-time error traceback mechanism. A single identifier or a list of identifiers enclosed in parentheses can follow **/DEBUG**; these identifiers are the names of options that inform the compiler which type of information it should generate.

Table 1.6 lists the available options, their corresponding actions, and their negations.

Table 1.6. /DEBUG Qualifier Options

Option	Action
ALL	Specifies that the compiler should include symbol and traceback information in the object module.
NONE	Specifies that symbol and traceback information will not be included in the object module.
[NO]SYMBOLS	Specifies that the compiler should include in the object module symbol definitions for all identifiers in the compilation.
[NO]TRACEBACK	Specifies that the compiler should include in the object module traceback information permitting virtual addresses to be translated into source program routine names and compiler-generated line numbers.

When debugging programs that contain schema, you must use the **/NOOPTIMIZE** qualifier on the **PASCAL** DCL command. If you do not use **/NOOPTIMIZE**, you might receive incorrect debug information or an Internal Debug Error when manipulating schema.

When you specify **SYMBOLS** without **TRACEBACK**, the table of compiler-generated line numbers is omitted from the debugger symbol table.

You should consider using **/NOOPTIMIZE** when you are using **/DEBUG**. Allowing optimizations to occur can make debugging difficult and can obscure some sections of the compilation unit.

The **/DEBUG** qualifier without options is equivalent to **/DEBUG=ALL**. The negation **/NODEBUG** is equivalent to **/DEBUG=NONE**.

/[NO]DESIGN
/NODESIGN (default)

Directs the compiler to accept design phase placeholders on all VSI OpenVMS systems as valid program elements within an VSI Pascal program. Placeholders are produced by you or by LSE/SCA; design comments are intended for use with LSE/SCA. Table 1.7 lists the options, their corresponding action, and their negation.

Table 1.7. /DESIGN Qualifier Options

Option	Action
[NO]PLACEHOLDERS	Directs the compiler to accept placeholders as valid program elements.
[NO]COMMENTS (obsolete)	Directs the compiler to recognize design comments.

The **/DESIGN** qualifier without an option is equivalent to **/DESIGN=(PLACEHOLDERS)**.

/[NO]DIAGNOSTICS
/NODIAGNOSTICS (default)

Creates a file containing compiler messages and diagnostic information. If you omit the file specification, the diagnostics file defaults to the name of your source file with a .DIA file type. The diagnostics file is used by products such as LSE/SCA.

/ENUMERATION_SIZE=*option*

Controls the allocation of unpacked enumerated data types and Boolean data types, which are considered to be an enumerated type containing two elements. Note that specifying the ENUMERATION_SIZE attribute overrides any value you previously specified with this qualifier. Table 1.8 lists the available options for the **/ENUMERATION_SIZE** qualifier.

Table 1.8. /ENUMERATION_SIZE Qualifier Options

Option	Action
BYTE	Allocates unpacked enumerated data types with up to 255 elements in a single byte. Otherwise, enumerated data types are allocated in a 16-bit word.
LONG (default)	Allocates all unpacked enumerated data types in a 32-bit longword.

/[NO]ENVIRONMENT
determined by attributes (default)

Produces an environment file in which declarations and definitions made at the outermost level of a compilation unit are saved. The default file name is the same as the source file name. The default file type is .PEN, an abbreviation for Pascal Environment. You can provide a different name for the environment file by including a file specification after the **/ENVIRONMENT** qualifier, for example, **/ENVIRONMENT=MASTER.PEN**.

The **/ENVIRONMENT** qualifier on the command line overrides the ENVIRONMENT attribute in the source program or module. By default, the attributes of the source program or module determine

whether an environment file is created; however, if the **/ENVIRONMENT** qualifier is specified at compile time, an environment file will always be created.

/[NO]ERROR_LIMIT
/ERROR_LIMIT=30 (default)

Terminates compilation after the occurrence of a specified number of error messages, excluding warning-level and information-level errors. If you specify **/NOERROR_LIMIT**, compilation continues until 500 errors have been detected.

/FLOAT= *floattype*

Selects the default format for REAL and DOUBLE data types. You must specify ***floattype*** if you use the **/FLOAT** qualifier. Table 1.9 lists the available options, their corresponding actions, and default information.

If the source program includes the [NO]G_FLOATING attribute, then the value of the **/FLOAT** qualifier must be in agreement with the value of the attribute.

Table 1.9. /FLOAT Qualifier Options

Option	Action	Default Information
D_FLOAT	REAL data type will be defined in the F_floating-point format; DOUBLE will be defined in the D_floating-point format.	
G_FLOAT	REAL data type will be defined in the F_floating-point format; DOUBLE will be defined in the G_floating-point format.	Default for VSI OpenVMS Alpha systems if **/FLOAT** or **/NOG_FLOATING** is not specified.
IEEE_FLOAT	REAL data type is defined in the IEEE S floating-point format; DOUBLE is defined in the IEEE T floating-point format.	Default for VSI OpenVMS I64 and and OpenVMS x86-64 systems if **/FLOAT** or **/NOG_FLOATING** is not specified.

Routines and compilation units between which double-precision quantities are passed should not mix floating-point formats. On VSI OpenVMS I64 and OpenVMS x86-64 systems, VAX floating-point support is implemented by converting the VAX floating format values to IEEE format, performing the operation, and converting the result back to VAX floating format. Because of the conversion, some programs might get slightly different results for F_floating, D_floating, and G_floating computations than they produce on VSI OpenVMS Alpha systems.

***file-spec*/LIBRARY**
none (default)

Specifies that a file is a text library file. The text library file specification is required. The text library files in a list of source files must be concatenated by plus signs. The default file type is .TLB.

/GRANULARITY=*option*

Directs the compiler to generate additional code to preserve the indicated granularity. Granularity refers to the amount of storage that can be modified when updating a variable. You can specify the following options for the **/GRANULARITY** qualifier:

- **BYTE**

- **LONGWORD**
- **QUADWORD** (default)

To update a variable that is smaller than a longword on older Alpha systems, VSI Pascal must issue multiple instructions to fetch the surrounding longword or quadword, update the memory inside to longword or quadword, and then write the longword or quadword back into memory. If multiple processes are writing into memory that is contained in the same longword or quadword, you might incur inaccurate results, unless **/GRANULARITY=BYTE** or some other synchronization mechanism is used.

On newer Alpha systems, the architecture has additional instructions that can modify byte and word-sized data directly. See the **/ARCHITECTURE** qualifier for additional information.

On OpenVMS I64 systems, the compiler may use quadword instructions to update unaligned variables unless modified by use of the **/GRANULARITY** qualifier.

On OpenVMS x86-64 systems, the x86-64 architecture and instruction set provides for byte granularity such that this qualifier has no impact on the generated code.

/IDENT=*identifer* or *string*
/IDENT=none (default)

Specifies the module-ident to be used in the object file or environment file as needed. This qualifier is equivalent to specifying an explicit [IDENT(quoted-string)] attribute in the source file. An explicit IDENT attribute in the source file will override the qualifier. **/IDENT=ABC** will yield an ident string of ABC. **/IDENT="abc"** will yield an ident string of abc.

/INCLUDE=(*directory*, ...)

Allows you to specify search locations for %INCLUDE directives and [INHERIT] attributes that specify file names without explicit disk or directory specifications.

The qualifier takes a list of directories to search. The compiler applies the **/INCLUDE** information to the following Pascal constructs:

- %INCLUDE 'name' or %INCLUDE 'name.ext'
- [INHERIT('name')] or [INHERIT('name.ext')]
- %INCLUDE 'name(modname)' or %INCLUDE 'name.ext(modname)'

The compiler searches as follows:

1. The current directory with a default extension of .pas for %INCLUDE directives, .pen for [INHERIT] attributes, and .tlb for %INCLUDE from text libraries (just like before)
2. Any directories specified with the **/INCLUDE** qualifier (in the order specified) with the appropriate default extension
3. SYS$LIBRARY: with the appropriate default extension

/[NO]LIST
/NOLIST (interactive default)
/LIST=*input_file_name*.LIS (batch default)

Produces a source listing file with a file type of .LIS. See the **/SHOW** qualifier for more information on controlling the contents of the source listing file.

/[NO]MACHINE_CODE
/NOMACHINE_CODE (default)

Produces a machine code section within the listing file. If the compiler detects errors in the source code, the compiler does not generate this section. The compiler ignores this qualifier if you do not also specify **/LIST** on the same command line.

/MATH_LIBRARY=*option* (VSI OpenVMS Alpha systems only)

Determines whether the compiler uses alternate math library routines that boost performance, but sacrifice accuracy. You can specify the following options for the **/MATH_LIBRARY** qualifier:

- **ACCURATE** (default)
- **FAST**

/[NO]OBJECT
/OBJECT= *input_file_name*.OBJ (default)

Specifies the name of the object file. If the compiler detects errors in the source code, the compiler writes no representation of object code to the listing file.

/[NO]OPTIMIZE
/OPTIMIZE (default)

Directs the compiler to optimize the code for the program or module being compiled so that the compiler generates more efficient code. A single identifier or a list of identifiers enclosed in parentheses can follow **/OPTIMIZE**; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to optimize.

Table 1.10 lists the available options, their corresponding actions, and their negations.

Table 1.10. /OPTIMIZE Qualifier Options

<table>
<tr><th>Option</th><th colspan="2">Action</th></tr>
<tr><td>ALL</td><td colspan="2">Enables all optimization components.</td></tr>
<tr><td>NONE</td><td colspan="2">Disables all /OPTIMIZE options.</td></tr>
<tr><td>[NO]INLINE=keyword</td><td colspan="2">Enables inline expansion of user-defined routines.</td></tr>
<tr><td rowspan="5">LEVEL=num</td><td colspan="2">Controls the optimization level. Values for num are:</td></tr>
<tr><td>0</td><td>Disables all optimizations. Identical in function to /NOOPTIMIZE.</td></tr>
<tr><td>1</td><td>Enables local optimizations and recognition of common subexpressions.</td></tr>
<tr><td>2</td><td>Enables all level 1 optimizations and some global optimizations, including the following: code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.</td></tr>
<tr><td>3</td><td>Enables all level 2 optimizations and some additional global optimizations that improve speed at the cost of extra code size. These optimizations include integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.</td></tr>
</table>

<table>
<tr><th>Option</th><th colspan="2">Action</th></tr>
<tr><td rowspan="3"></td><td></td><td>Identical in function to <code>/OPTIMIZE=NOINLINE</code>.</td></tr>
<tr><td>4</td><td>Enables all level 3 optimizations and inline expansion of procedures and functions. Identical in function to <code>/OPTIMIZE</code>.</td></tr>
<tr><td>5</td><td>Enables software pipelining and additional software dependency analysis, which in certain cases improves run-time performance.</td></tr>
<tr><td><code>UNROLL=num</code></td><td colspan="2">Controls number of times loops are unrolled. The default is 4. <code>/UNROLL=0</code> disables loop unrolling. Loop unrolling is only enabled above optimization level 2. (VSI OpenVMS I64 and VSI OpenVMS Alpha systems only.)</td></tr>
<tr><td><code>TUNE=processor</code></td><td colspan="2">Tune the object code to run best on the processor chosen. The default is Generic. Values for the processor are EV4, EV5, EV56, EV6, EV7, EV67, EV68, Generic, and Host. (VSI OpenVMS Alpha systems only.)</td></tr>
</table>

The **`/OPTIMIZE`** qualifier without options is equivalent to **`/OPTIMIZE=ALL`**. The negation **`/NOOPTIMIZE`** is equivalent to **`/OPTIMIZE=NONE`**.

The OPTIMIZE and NOOPTIMIZE attributes in the source program or module override the **`/OPTIMIZE`** and **`/NOOPTIMIZE`** qualifiers on the command line.

For More Information:

- On compiler optimizations (Section 3.1)

The **`/NOOPTIMIZE`** qualifier guarantees full evaluation of both operands of the AND and OR Boolean operators to aid in diagnosing all potential programming errors. If you wish to have short-circuit evaluation even with the **`/NOOPTIMIZE`** qualifier, use the AND_THEN and OR_ELSE Boolean operators.

You can also specify an optimization level. Optimization levels from level 2 and higher include all optimizations from lower levels.

On OpenVMS x86-64 systems, the optimizer and code-generate is different than Alpha and I64 systems so the exact list of optimizations enabled by an optimization level may be different.

`/PEN_CHECKING_STYLE=`*`option`*
`D=/PEN_CHECKING_STYLE=COMPILATION_TIME`

Specifies the desired environment file checking method. This qualifier is identical to the [PEN_CHECKING_STYLE(keyword)] module-level attribute. It accepts the same keywords as the attribute. An explicit [PEN_CHECKING_STYLE(keyword)] attribute in the source file will override the **`/PEN_CHECKING_STYLE`** DCL qualifier.

Table 1.11. `/PEN_CHECKING_STYLE` Qualifier Options

Option	Action
`COMPILATION_TIME`	Uses the compilation time of the environment file in all subsequent compile-time and link-time checking for users of thisenvironment file.

Option	Action
`IDENT_STRING`	Uses the [IDENT()] string of the environment file in all subsequent compile-time and link-time checking for users of thisenvironment file.
`NONE`	Disables all compile-time and link-time checking for users of this environment file.

`/[NO]PLATFORMS`
`/NOPLATFORMS` (default)

Displays informational messages about nonportable language features for the specified platform.

Table 1.12 lists the supported qualifier options.

Table 1.12. `/PLATFORMS` Qualifier Options

Option	Action
`COMMON`	Displays informational messages for all platforms.
`OpenVMS_I64`	Displays informational messages for the VSI OpenVMS I64 platform.
`OpenVMS_Alpha`	Displays informational messages for the VSI OpenVMS Alpha platform.
`OpenVMS_x86-64`	Displays informational messages for the OpenVMS x86-64 platform.

`/PSECT_MODEL=[NO]MULTILANGUAGE` (VSI OpenVMS Alpha systems only)
`/PSECT_MODEL=NOMULTILANGUAGE` (default)

This qualifier controls whether the compiler pads the size of overlaid PSECTs, so as to ensure compatibility when the PSECT is shared by code created by other VSI OpenVMS Alpha compilers.

When a PSECT generated with a [COMMON] attribute is overlaid with a PSECT consisting of a C struct or a Fortran COMMON block, linker error messages can result due to the inconsistent sizes of the PSECTs; some languages pad the size of PSECTS, while other do not.

`/[NO]SHOW`
`/SHOW=(DICTIONARY,HEADER,INCLUDE,SOURCE,STATISTICS)` (default)

Specifies a list of items to be included in the listing file. A single identifier or a list of identifiers enclosed in parentheses can follow `/SHOW`; these identifiers are the names of options that inform the compiler which type of information it should generate.

Table 1.13 lists the available options, their corresponding actions, and their negations.

Table 1.13. `/SHOW` Qualifier Options

Option	Action
`ALL`	Enables listing of all options.
`NONE`	Disables all `/SHOW` options.
`[NO]DICTIONARY`	Enables listing of %DICTIONARY records.

Option	Action
[NO]HEADER	Enables page headers.
[NO]INCLUDE	Enables page headers.
[NO]SOURCE	Enables listing of VSI Pascal source code.
[NO]STATISTICS	Enables listing of compilation statistics.
[NO]STRUCTURE_LAYOUT	Enables listing of the sizes, record field offsets, and comments about nonoptimal performance for variables and types in your program.

The compiler ignores the **/SHOW** qualifier if you do not also specify the **/LIST** qualifier on the same command line. The negation **/NOSHOW** is equivalent to **/SHOW=NONE**; **/SHOW** is equivalent to **/SHOW=ALL**.

/STANDARD=*option*
/NOSTANDARD (default)

Causes the compiler to generate messages wherever the compilation unit uses VSI Pascal language extensions, which are nonstandard Pascal features. Within the VSI Pascal documentation set, these standards are collectively referred to as the Pascal standard.

Table 1.14 lists the available options and their corresponding actions.

Table 1.14. /STANDARD Qualifier Options

Option	Action
NONE	Disables standards checking.
ANSI	Uses the rules of the ANSI standard.
ISO	Uses the rules of the ISO standard.
EXTENDED	Uses the rules of the Extended standard.
[NO]VALIDATION	Performs validation for the given standard.

The **/STANDARD** qualifier allows you to use only two options. The first option selects the standard to be used (ANSI, ISO or EXTENDED). The second option determines whether the strict validation rules are to be enforced ([NO]VALIDATION). **/STANDARD=(ANSI, ISO, VALIDATION)** is not allowed because both ANSI and ISO are specified.

By default, these information-level messages are written to the error file SYS$ERROR. Using the **VALIDATION** option changes all nonstandard information-level messages to error-level messages.

The **/STANDARD** qualifier without options is equivalent to **/STANDARD=(ANSI, NOVALIDATION)**. **/STANDARD=VALIDATION** is equivalent to **/STANDARD=(ANSI, VALIDATION)**. The negation **/NOSTANDARD** is equivalent to **/STANDARD=NONE**.

/[NO]SYNCHRONOUS_EXCEPTIONS (VSI OpenVMS Alpha systems only)
/NOSYNCHRONOUS_EXCEPTIONS (default)

Specifies that the compiler should generate code to insure that exceptions are reported as near as possible to the instruction that generated the exception. This can avoid confusion in tracing the source of an exception, however, there is a performance penalty for using this qualifier.

/[NO]TERMINAL
/NOTERMINAL (default)

Specifies a list of items to be displayed on the terminal. A single identifier or a list of identifiers enclosed in parentheses can follow the **/TERMINAL** qualifier; these identifiers are options that inform the compiler which type of information to display.

Table 1.15 lists the available options and their corresponding actions.

Table 1.15. /TERMINAL Qualifier Options

Option	Action
ALL	Displays all options.
NONE	Disables all **/TERMINAL** options.
[NO]FILE_NAME	Displays file names on Pascal command line as they are being processed.
[NO]STATISTICS	Displays compiler statistics.

The **/TERMINAL** qualifier without options is equivalent to **/TERMINAL=ALL**. The negation **/NOTERMINAL** is equivalent to **/TERMINAL=NONE**.

/[NO]TIE (VSI OpenVMS Alpha systems only)
/NOTIE (default)

Specifies that the generated code can call images translated by the VAX Environment Software Translator (VEST) utility, which translates OpenVMS VAX system images into functionally equivalent VSI OpenVMS Alpha system images. The Translated Image Environment (TIE) allows translated images to execute as if on an OpenVMS VAX system.

/[NO]USAGE
/USAGE=(EMPTY_RECORDS, NONGRNACC, PACKED_ACTUALS, UNSUPPORTED_CDD, UNINITIALIZED, VOLATILE) (defaults)

Directs the compiler to perform compile-time checks indicated by the chosen options. A single identifier or a list of identifiers enclosed in parentheses can follow **/USAGE**; these identifiers are options that tell the compiler which checks to perform.

Table 1.16 lists the available options, their corresponding actions, and their negations.

Table 1.16. /USAGE Qualifier Options

Option	Action
ALL	Enables checking of all options.
NONE	Disables all **/USAGE** options.
[NO]EMPTY_RECORDS	Checks for fetching records with no fields. Such fields are usually created by the %DICTIONARY directive for unsupported data types.
[NO]NONGRNACC (VSI OpenVMS I64 and VSI OpenVMS Alpha systems)	Specifies that the compiler should issue warning messages for code sequences that might not match your granularity request from the **/GRANULARITY** qualifier. When the compiler cannot guarantee that the generated code matches the granularity setting, a warning message is issued. You

Option	Action
	should examine your code to make sure that the variable being accessed is quadword-aligned and is a multiple of quadwords in size. In this case, the resulting code will be correct, although the compiler might not be able to determine that at compile time. Such cases involve pointer dereferences or VAR parameters. These messages are enabled by default by the compiler.
[NO]PACKED_ACTUALS	Checks for passing components of packed structures to VAR parameters.
[NO]PERFORMANCE	Checks for variables and record fields that are poorly sized or aligned on inefficient boundaries. This provides the same information that is found in the **/SHOW=STRUCTURE_LAYOUT** listing section.
[NO]UNCALLABLE (VSI OpenVMS I64 and VSI OpenVMS Alpha systems)	Specifies whether the compiler should issue informational messages for routines that are declared but never called.
[NO]UNCERTAIN	Checks for variables that can be uninitialized depending on program flow.
[NO]UNINITIALIZED	Checks for variables that are known to be uninitialized.
[NO]UNSUPPORTED_CDD	Checks for usage of CDD/Repository constructs that do not correspond to VSI Pascal data types.
[NO]UNUSED	Checks for variables that are declared but never referenced.
[NO]VOLATILE (VSI OpenVMS I64 and VSI OpenVMS Alpha systems)	Checks for VOLATILE variables that are not aligned properly. On VSI OpenVMS I64 and VSI OpenVMS Alpha systems, certain unaligned VOLATILE variables cannot be updated in an atomic fashion.
[NO]64BIT_TO_DESCR	Specifies that the compiler should disable the checking of passing 64-bit pointer expressions to parameters passed by 32-bit descriptors. Normally, the compiler will flag that as an error but certain P2 64-bit addresses can be passed in the descriptor if the address is treated as an unsigned integer.

The following types of variables are not checked for uninitialization:

- Variables that have a file component
- Predeclared INPUT or OUTPUT identifiers
- Variables that have global, external, or inherited visibility
- Variables declared with the AT attribute
- Variables declared with the COMMON attribute
- Variables declared with the READONLY attribute
- Variables declared with the VOLATILE attribute
- Variables used as parameters

- Variables used as function identifiers

The **/USAGE** qualifier without options is equivalent to **/USAGE=ALL**. The negation **/NOUSAGE** is equivalent to **/USAGE=NONE**.

The VSI Pascal compiler can detect when some variables are uninitialized; however, it cannot detect that an uplevel variable is uninitialized at the point at which it was referenced. This is because at the time the routine is lexically scanned, the compiler has not seen any of the calls to that routine.

/[NO]VERSION
/NOVERSION (default)

Controls whether the compiler prints compiler and OpenVMS version information to SYS$OUTPUT and then returns to the operating system. No other command qualifiers or source files are processed when **/VERSION** is used.

/[NO]WARNINGS
/WARNINGS (default)

Directs the compiler to generate diagnostic messages in response to warning-level or informational-level errors.

By default, these messages are written to the error file SYS$ERROR. A warning or informational diagnostic message indicates that the compiler has detected acceptable but unorthodox syntax or has performed some corrective action; in either case, unexpected results can occur.

Note that informational messages generated when you specify the **/STANDARD** qualifier do not appear if **/NOWARNINGS** is enabled.

/[NO]ZERO_HEAP
/ZERO_HEAP (default)

Specifies that heap memory should be zeroed after allocation. By default, the Pascal RTL will return zero-filled memory for each call to the NEW built-in. Using the **/NOZERO_HEAP** qualifier can increase runtime performance.

For More Information:

- On debugging (Section 4.1)
- On text libraries (Section 1.1.4)
- On LSE and SCA information (Section 4.2)
- On error messages (Section 1.4)
- On the contents of a compiler listing (Section 1.1.3)
- On Pascal standards (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On using environment files (Section 2.1)
- On the AND_THEN and OR_ELSE Boolean operators (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

1.1.3. Contents of the Compilation Listing File

You control the contents of a compilation listing by appending qualifiers to the PASCAL command. Table 1.17 lists the parts of a complete compilation listing and the qualifiers that cause them to be generated.

Table 1.17. Compilation Listing Contents and Qualifiers

Section	Generated With
Source code	**/LIST**
Cross-reference	**/LIST /CROSS_REFERENCE**
Machine code	**/LIST /MACHINE_CODE**
Compilation statistics	**/LIST /SHOW=STATISTICS**
Structure layout	**/LIST /SHOW=STRUCTURE_LAYOUT**

A compilation listing file usually contains source code because the **/SHOW=SOURCE** qualifier is enabled by default. The **/LIST** qualifier does not initiate the printing of the listing file. To obtain a line printer copy of your listing file, use the **PRINT** command.

You can control the number of lines that appear on a listing page by defining the SYS$LP_LINES logical name before invoking the compiler. For example:

```
$ DEFINE SYS$LP_LINES 100
$ PASCAL/LIST [DIR]M
```

This set of commands creates a printed page size of 94 lines (the compiler subtracts six lines for margins).

The following sections describe the contents of each part of the listing file.

For More Information:

- On the PASCAL command qualifiers (Section 1.1.2)
- On the SYS$LP_LINES logical (*VSI OpenVMS Programming Concepts Manual*)

1.1.3.1. Source Code

The source code part of a listing file includes indicator flags (I = Include file; D = Dictionary extraction; C = Comment line); a procedure nesting level (PL); a statement nesting level (SL); listing line number; source code; and any diagnostic messages.

```
IDC-PL-SL
     0  0           1 program hw(output);
     0  0           2 begin
  C  0  0           3 ! Comment line
     0  1           4  writeln('Built with ',%compiler_version);
     0  0           5 end.
```

1.1.3.2. Cross-Reference Section

The cross-reference part of a listing file contains a list of all identifiers and labels used within the source code. This list includes the name of the identifier or label, the program element it represents, the source

code line numbers where it appears, and, where applicable, the attributes, declaring block, and function result type associated with it.

1.1.3.3. Machine Code Section

The machine code part of a listing file contains a representation of the object code generated by the compiler. Information is organized by program section and, within each program section, by executable block.

For each program section, the compiler generates the program section name and properties, hexadecimal representation of the code, computer-generated labels, symbolic opcode, and symbolic operands (if needed). The listing format is similar to the native assembler for the target architecture, but it will not assemble. Some formats have hexadecimal notation on the left side and line numbers on the right side.

1.1.3.4. Structured Layout Section

This listing section gives the sizes, record field offsets, and comments about nonoptional performance for variables and types in your program.

This section is useful to aid in restructuring data types for optimal performance.

1.1.3.5. Compilation Statistics

The compilation statistics part of a listing file contains the following categories of summary information:

- Psect Summary, listing the program section name, number of bytes, and attributes of all program sections created during compilation.
- Environment Statistics, listing the names of all environment files inherited by the compilation and symbol information. This information includes the total number of symbols in the environment file, the number of symbols actually used by the compilation, and the percentage of used symbols versus defined symbols.

  Note that the VSI Pascal compiler defines symbol in terms of internal representation. This definition can not reflect the complexity of the environment source; that is, the number of symbols shown loaded can not reflect the number of symbols in your program.
- Command Qualifiers and Options List, containing the exact command line passed by DCL to the VSI Pascal compiler, and the qualifier options in effect during compilation.
- Compiler Internal Timing Statistics, noting the number of page faults and amount of elapsed time and CPU time required for each phase of the compilation.
- Compilation Statistics, listing the total number of messages generated at each level—informational, warning, error, and fatal; the time and speed of compilation; and the number of page faults that occurred. The last line is a message indicating that the compilation of the source code is complete.

1.1.4. Text Libraries

A text library contains modules of source text that you can incorporate in a program by using the %INCLUDE directive. This directive indicates the module and, optionally, the text library in which the module can be found. Text library names can be specified in the following ways:

- In the %INCLUDE directive

- On the PASCAL command line
- In a DEFINE default library command (DCL)

1.1.4.1. Using the %INCLUDE Directive for Text Libraries

The %INCLUDE directive has the following form:

```
%INCLUDE '[[file-spec]] (module-name) [[/[[NO]]LIST ]]'
```

file-spec

The name of the text library containing a module to be included in the compilation.

module-name

The name of a text module, located in a text library, that is to be included in the source file. The name of the module must be enclosed in parentheses. The module names can include any printable character except a space, horizontal tab, comma, or exclamation point. The maximum length of the module name is determined when the text library is created. Module names are also case insensitive.

/[NO]LIST

Indicates that the included module should be printed in the listing of the program if a listing is being generated. If not specified, the default is determined by the [NO]INCLUDE option on the **/SHOW** qualifier. The **INCLUDE** option enables the listing of %INCLUDE files and is enabled by default.

For example, the following %INCLUDE directive specifies both the text library DATAB.TLB and the module External_Declarations:

```
%INCLUDE 'DATAB.TLB (External_Declarations)'
```

If the text library is not specified in the %INCLUDE directive, its name must appear on the PASCAL command line or it must be specified by a DCL **DEFINE** command.

For More Information:

- On **/LIST** and **/SHOW** qualifiers (Section 1.1.2)
- On default libraries (Section 1.1.4.3)

1.1.4.2. Specifying Text Libraries on the Command Line

The **/LIBRARY** qualifier identifies text libraries specified on the PASCAL command line. When you compile a source file that includes a module from a text library, concatenate the name of the text library to the name of the source file and append the **/LIBRARY** qualifier. You specify concatenation with a plus sign. For example:

```
$ PASCAL APPLIC+DATAB/LIBRARY
```

This command instructs the compiler to search the DATAB text library each time it encounters an %INCLUDE directive within the APPLIC source file.

If more than one library is specified, the compiler searches the libraries in the order they appear on the command line. For example:

```
$ PASCAL APPLIC+DATAB/LIBRARY+DATAC/LIBRARY+DATAD/LIBRARY
```

If you request multiple compilations, the **/LIBRARY** qualifier must appear after each compilation in which it is needed. For example:

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

If you are concatenating source files, the **/LIBRARY** qualifier can appear only after the last source file. For example:

```
$ PASCAL METRIC.PAS+APPLIC.PAS+DATAB/LIBRARY
```

Any Pascal output qualifiers that appear after the **/LIBRARY** qualifier, such as **/OBJECT** or **/LISTING**, apply to the last source file name that you specified. For example, the following **PASCAL** command creates APPLIC.OBJ:

```
 $ PASCAL METRIC+APPLIC+DATAB/LIBRARY/OBJECT
```

For More Information:

- On the **PASCAL** command and qualifiers (Section 1.1)

1.1.4.3. Defining Default Libraries

You can define one of your private text libraries as a default text library for the Pascal compiler to search. The VSI Pascal compiler searches the default library after it searches libraries specified in the PASCAL command.

To establish a default library, define the logical name PASCAL$LIBRARY, as in the following example of the DEFINE command:

```
$ DEFINE PASCAL$LIBRARY DISK$:[LIB]DATAB
```

While this assignment is in effect, the compiler automatically searches the library DISK$:[LIB]DATAB.TLB for any included modules that it cannot locate in libraries explicitly specified on the **PASCAL** command.

The VSI Pascal compiler uses PASCAL$LIBRARY as the file name for the default text library; the location and search order of the logical name tables are controlled by Record Management Services (RMS).

If PASCAL$LIBRARY is defined as a search list, the compiler opens the first item specified in the list. If the include module is not found there, the search is terminated and an error message is issued.

For More Information:

- On the DCL command **DEFINE** (*VSI OpenVMS DCL Dictionary*)

1.2. LINK Command

The LINK command invokes the OpenVMS Linker, which combines object modules into one executable image, which can then be executed by the VSI OpenVMS operating system.

The linker uses the name of the input file that you specified first on the command line for the name of the output file. The default for linker output files (executable images) is the .EXE file type.

The **LINK** command format is as follows:

```
LINK [[{/command-qualifier} ...]]
```

```
     {file-spec[[{/file-qualifier} ...]]} , ...
/qualifier [[= {file-spec | library-module | (library-module, ...)} ]]
```

/command-qualifier

The name of a qualifier that indicates special processing to be performed by the linker on all files listed.

file-spec

The name of one of the following:

- The input file (which can be the name of an object module library) that contains the object code to be linked.
- The options file, used only with the **/OPTIONS** qualifier.
- The output file, used only with the **/EXECUTABLE** and **/MAP** qualifiers.

/file-qualifier

The name of a qualifier (the **/INCLUDE**, **/LIBRARY**, or **/OPTIONS** qualifier) that indicates special processing to be performed by the linker on the files to which the qualifier is attached.

library-module

The name of one or more object modules or shareable image libraries that you can only specify using the **/INCLUDE** or **/LIBRARY** qualifiers.

A source program or module cannot run on the system until it is linked. If you are using .PEN (Pascal Environment) files that include variables, procedures, or functions, make sure you link the object file into the .EXE file. When you execute the LINK command, the OpenVMS Linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference

1.2.1. LINK Command Examples

This section contains examples of **LINK** command lines.

```
$ LINK DANCE.OBJ, CHACHA.OBJ, SWING.OBJ
```

This command links the object files DANCE.OBJ, CHACHA.OBJ, and SWING.OBJ to produce one executable image called DANCE.EXE.

```
$ LINK/EXECUTABLE=TEST CIRCLE
```

This command links CIRCLE.OBJ and then causes the executable image generated by the linker to be named TEST.EXE.

```
$ LINK SCHEDULE,COURSES/INCLUDE=(HISTORY,ALGEBRA,PHILOSOPHY)
```

This example shows the use of the **/INCLUDE** qualifier with a library named COURSES. The linker extracts the modules HISTORY, ALGEBRA, and PHILOSOPHY from the library COURSES and includes them in the executable image SCHEDULE.EXE.

```
$ LINK SCHEDULE,COURSES/LIBRARY/INCLUDE=(HISTORY,ALGEBRA, PHILOSOPHY)
```

This example also causes the linker to include the modules HISTORY, ALGEBRA, and PHILOSOPHY in the image file SCHEDULE.EXE. However, the **/LIBRARY** qualifier causes the linker to search the rest of the library COURSES and link in any other modules needed to resolve symbolic references in SCHEDULE, HISTORY, ALGEBRA, and PHILOSOPHY.

```
$ LINK UPDATE/EXE=[PROJECT.EXE]/MAP=[PROJECT.MAP]
```

This command produces these files:

- [PROJECT.EXE]UPDATE.EXE
- [PROJECT.MAP]UPDATE.MAP

For More Information:

- On debugging (Section 4.1)
- On error messages (Section 1.4)
- On including modules from object module libraries (Section 1.2.3)
- On messages generated by the linker (*VSI OpenVMS Linker Utility Manual*)
- On DCL syntax (HELP)
- On the OpenVMS Linker (*VSI OpenVMS Linker Utility Manual*)

1.2.2. LINK Qualifiers

The following are command and file qualifiers that you can use when linking object modules:

/[NO]DEBUG
/NODEBUG (default)

Indicates that the VMS Debugger is to be included in the executable image and that a symbol table is to be generated. If you specify **LINK/DEBUG**, the program links and then executes under the control of the debugger.

/[NO]EXECUTABLE
/EXECUTABLE (default)

Produces an executable image. A file specification can follow **/EXECUTABLE** to designate a name for the image file. The **/NOEXECUTABLE** qualifier, which suppresses production of the image file, is useful when you want to verify the results of linking an object file before the image is produced.

/INCLUDE
none (default)

Specifies that the input file is an object module or a shareable image library,and that the modules named are the only ones in the library to be explicitly included as input. In the case of shareable image libraries, the module is the shareable image name. You must specify at least one module name with the **/INCLUDE** qualifier. The default for library modules is the .OLB file type.

This qualifier is a file qualifier and cannot be used directly on the **LINK** command.

`/LIBRARY`
none (default)

Specifies that the input file is an object module or shareable image library,which the linker must search to resolve undefined symbols within other input modules specified on the same command line.

You can use the **`/LIBRARY`** qualifier with the **`/INCLUDE`** qualifier to modify the same input file specification. In that case, the same library is searched for unresolved references.

This qualifier is a file qualifier and cannot be used directly on the **`LINK`** command. The default for the file to which this qualifier is applied is the .OLB file type.

`/[NO]MAP`
`/NOMAP` (interactive default)
`/MAP/NOCROSS_REFERENCE` (batch default)

Controls the generation of a map file and its contents. The **`/MAP`** qualifier produces a map file, which you can name by including a file specification.

The map file is stored on the default device in the default directory. If you do not include a file specification with **`/MAP`**, the map file is given the name of the first input file and a .MAP file type.

With the **`/MAP`** qualifier, you can use the qualifiers **`/BRIEF`**, **`/FULL`**, and **`/CROSS_REFERENCE`** to define the type of information included in the map file.

***`filename`*`/OPTIONS`**
none (default)

Specifies that the input file is a linker options file, which can contain input file specifications as well as special instructions recognized only by the linker. You can also use options files to create shareable images.

`/[NO]SHAREABLE`
`/NOSHAREABLE` (default)

Creates a shareable image. 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. For example, a shareable image can contain a library of routines or can be used by the system manager to create a global section for all users.

To include a shareable image as input to the linker, you can 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. You can also include a shareable image by using a linker options file.

The **`/NOSHAREABLE`** qualifier specifies that the image produced cannot be linked with other images.

`/[NO]TRACEBACK`
`/TRACEBACK` (default)

Causes the generation of error messages to be accompanied by symbolic traceback information. This information shows the sequence of calls that transferred control to the program in which the error occurred. **`/NOTRACEBACK`** suppresses production of traceback information.

The traceback capability is automatically included with the **/DEBUG** qualifier; therefore, if you specify both **/DEBUG** and **/NOTRACEBACK**, **/NOTRACEBACK** has no effect.

For More Information:

- On debugging (Section 4.1)
- On object-module libraries (Section 1.2.3)
- On shareable images, options files, and contents of map files (*VSI OpenVMS Linker Utility Manual*)

1.2.3. Object Module Libraries

An object module library contains modules of program text that have been successfully compiled. To link modules contained in a object module library, use the **/INCLUDE** qualifier and specify the modules you want to link. For example:

```
$ LINK GARDEN,VEGETABLES/INCLUDE=(EGGPLANT,TOMATO, BROCCOLI,ONION)
```

This example directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN.

Besides program modules, an object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the **/LIBRARY** qualifier. When you use the **/LIBRARY** qualifier during a link operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

Also, 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.

In the following example, the linker uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL.

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command **DEFINE**. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the **LINK** command.

For More Information:

- On the **LINK** command and qualifiers (Section 1.2)
- On the OpenVMS Linker (*VSI OpenVMS Linker Utility Manual*)
- On the DCL command **DEFINE** (*VSI OpenVMS DCL Dictionary*)

1.3. RUN Command

The RUN command executes programs that have been linked into an executable image by the OpenVMS Linker. This command has the following format:

```
RUN [/command-qualifier] file-spec
```

/command-qualifier

The name of a qualifier that indicates special processing to be performed by the linker on all files listed.

file-spec

The name of the executable image you want to run. The default file type for executable images is .EXE.

The image activator accepts one command qualifier, as follows:

/[NO]DEBUG
depends on linking (default)

> The **/[NO]DEBUG** qualifier is optional. Specify the **/DEBUG** qualifier to request the debugger, if the image was not linked with it. You cannot use **/DEBUG** on images linked with the **/NOTRACEBACK** qualifier. If the image was linked with the **/DEBUG** qualifier and you do not want the debugger to prompt, use the **/NODEBUG** qualifier. The default action depends on whether you specified **/DEBUG** on the **LINK** command line.

Consider the following examples:

```
$ RUN PROG
```

This example executes the image PROG.EXE. If you specified **/DEBUG** to the linker while creating PROG.EXE, the image activator passes control to the debugger upon execution. If you did not specify **/DEBUG** to the linker while creating PROG.EXE, the image activator executes the program.

```
$ RUN/NODEBUG PROG
```

This example executes the image PROG.EXE without invoking the debugger.

For More Information:

- On debugging (Section 4.1)
- On messages generated by the image activator (Section 1.4)
- On the DCL command **RUN** (*VSI OpenVMS DCL Dictionary*)

1.4. Error Messages

During program development, you can have to respond to messages regarding possible syntax or logic errors in your program. These messages have the following form:

```
%SOURCE-CLASS-MNEMONIC, message_text
```

SOURCE

A code that identifies the origin of the message. For example, the PASCAL code identifies the VSI Pascal compiler, and the PAS code identifies the VSI Pascal run-time system.

CLASS

A single character that determines message severity. The four classes of error messages are: Informational (I), Warning (W), Error (E), and Fatal (F). The definition for each class depends on the source of the message, but execution of your request does not continue when E- or F-level errors occur.

MNEMONIC

A name that is unique to that message.

message_text

Explains the event that caused the message to be generated.

For example, a common linker error occurs when you omit required file or library names from the command line, and the linker cannot locate the definition for a specified global symbol reference. The following error messages appear when a main program in OCEAN.OBJ calls a subprogram in SEAWEED.OBJ that is not specified in the **LINK** command:

```
%LINK-W-NUDFSYMS, 1 undefined symbol
%LINK-I-UDFSYMS,        SEAWEED
%LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
%LINK-W-DIAGISUED, completed but with diagnostics
```

For More Information:

- On the complete list of VSI Pascal compile-time and run-time errors (Appendix C)
- On the complete list of linker messages (*VSI OpenVMS Linker Utility Manual*)

Chapter 2. Separate Compilation

Pascal allows you to divide your application into subprograms by creating procedures and functions. VSI Pascal allows you further modularity by allowing you to create compilation units, called programs and modules, that can be compiled separately.

Note

The sections at the beginning of this chapter use code fragments from the examples in this chapter and in the online example directory, which by default is PASCAL$EXAMPLES.

For More Information:

- On the ENVIRONMENT, HIDDEN, and INHERIT attributes (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On compiling and executing programs and modules (Chapter 1)

2.1. ENVIRONMENT, HIDDEN, and INHERIT Attributes

To divide your program into a program and a series of modules, you need to decide, according to the needs of your application, which data types, constants, variables, and routines need to be shared either by other modules or by the program. To share data, create an environment file by using the ENVIRONMENT attribute in a module. Consider the following example:

```
{
Source File: share_data.pas
This program initializes data to be shared with another compilation
unit.
}
[ENVIRONMENT( 'share_data' )]
Module Share_Data;
CONST
   Rate_For_Q1 = 0.1211;
   Rate_For_Q2 = 0.1156;
   Rate_For_Q3 = 0.1097;
   Rate_For_Q4 = 0.11243;
TYPE
   Initialized_Type = ARRAY[1..10] OF INTEGER VALUE
                      [1..5: 67; 6,9: 105; OTHERWISE 33];
END.
```

If you do not specify a file name, VSI Pascal creates an environment file using the file name of the source file and a default extension of .PEN. Another compilation unit can access the types and constants in the previous example by inheriting the environment file, as follows:

```
{
Source File: program.pas
This code inherits data declarations and uses them in a program.
}
[INHERIT( 'share_data' )]
PROGRAM Use_Data( OUTPUT );
```

```
VAR
   a, b, c          : Initialized_Type;
   Total            : REAL VALUE 0.0;
BEGIN
Total := Total + ( Total * Rate_For_Q3 );
WRITELN( b[7] );    {b is of an initialized type}
END.
```

To build and run the application made up of the code in the previous examples, use the following commands:

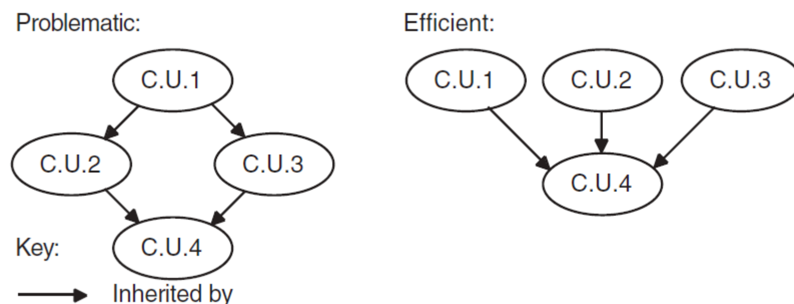
```
$ PASCAL SHARE_DATA
$ PASCAL PROGRAM
$ LINK PROGRAM
$ RUN PROGRAM
    33
```

If a module contains variable declarations, routine declarations, schema types, or module initialization or finalization sections, you must link the program with the module that created the environment file to resolve external references. To prevent errors, you may wish to link programs with modules of inherited environment files as standard programming practice. For example, if SHARE_DATA contained a variable declaration, you must enter the following to resolve the external reference:

```
$ PASCAL SHARE_DATA
$ PASCAL PROGRAM
$ LINK PROGRAM, SHARE_DATA
$ RUN PROGRAM
    33
```

For many applications, it is a good idea to place all globally accessible data into one module, create a single environment file, and inherit that module in other compilation units that need to make use of that data. Using environment files in this way reduces the difficulties in maintaining the data (it is easier to maintain one file) and it eliminates problems that can occur when you **cascade** environment files. If compilation unit A inherits an environment file from compilation unit B, and if unit B inherits a file from unit C, then inheritance is cascading. Figure 2.1 shows a cascading inheritance path and a noncascading inheritance path.

Figure 2.1. Cascading Inheritance of Environment Files



Cascading is not always undesirable; it depends on your application and on the nature of the environment files. For example, if cascading occurs for a series of constant and type definitions that are not likely to change, cascading may require very little recompiling and relinking. However, if the constant and type definitions change often or if environment files contain routines and variables, you may find it easier to redesign the inheritance paths of environment files due to the recompiling and relinking involved.

Also, the inheritance path labeled Efficient in Figure 2.1 is not immune to misuse. That inheritance path, although it avoids the problems of cascading, may still involve multiply declared identifiers (identical

identifiers contained in several of the compilation units whose environment files are inherited by compilation unit 4).

In many instances, VSI Pascal does not allow multiply declared identifiers in one application. For example, a compilation unit cannot inherit two environment files that declare the same identifier; also, a compilation unit usually cannot inherit an environment file that contains an identifier that is identical to an identifier in the outermost level of the unit (one exception, for example, is the redeclaration of a redefinable reserved word or of an identifier predeclared by VSI Pascal). Also, VSI Pascal allows the following exceptions to the rules concerning multiply declared identifiers:

- A variable identifier can be multiply declared if all declarations of the variable have the same type and attributes, and if all but one declaration at most are external.
- A procedure identifier can be multiply declared if all declarations of the procedure have congruent parameter lists and if all but one declaration at most are external.
- A function identifier can be multiply declared if all declarations of the function have congruent parameter lists and identical result types, and if all but one declaration at most are external.

If a compilation unit creates an environment file and if it contains data that you do not want to share with other compilation units, you can use the HIDDEN attribute. Consider the following example:

```
[ENVIRONMENT]
MODULE Example;
TYPE
   Array_Template( Upper : INTEGER ) =
      [HIDDEN] ARRAY[1..Upper] OF INTEGER;
   Global_Type : Array_Template( 10 );
VAR
   i : [HIDDEN] INTEGER;   {Used for local incrementing}

PROCEDURE x;
   BEGIN
      i := i + 1;
   END;

PROCEDURE y;
   BEGIN
      FOR i := i + 1;
   END;
END.
```

The code in the previous example hides the schema type, preventing the schema type from being used in inheriting modules. (Whether to hide the type depends on the requirements of a given application.) Also, VSI Pascal does not include the `variable i` in the environment file; this allows inheriting modules to declare the identifier `variable i` as an incrementing variable without being concerned about generating errors for a multiply defined identifier.

VSI Pascal performs compile-time and link-time checks to ensure that all compilations that inherit environment files actually used the same environment file definition. Information is placed in the object file such that the VSI OpenVMS Linker performs the same check between each object file that inherited environment files.

By default, compilation units that inherit an environment file compare the embedded compilation time inside the environment file:

- Uses found in any other environment files that are also inherited.

If the times are different, a compile-time message is displayed.

This checking can be disabled or modified by using the PEN_CHECKING_STYLE attribute in the Pascal source file that created the environment file. Once the environment file exists, its selected checking style will be performed at each use.

The PEN_CHECKING_STYLE attribute is valid at the beginning of a MODULE that creates an environment. The syntax is:

```
PEN_CHECKING_STYLE(keyword)
```

In this format "keyword" is:

- COMPILATION_TIME

  Uses the compilation time of the environment file in all subsequent compile-time checking for users of this environment file. This is the default.

- IDENT_STRING

  Uses the [IDENT()] string of the environment file in all subsequent compile-time checking for users of this environment file.

- NONE

  Disables all compile-time checking for users of this environment file.

When VSI Pascal compiles a module with the **/ENVIRONMENT** qualifier or [ENVIRONMENT] attribute, it generates an environment file and an object file. This is also true when compiling a or program, although it is not customary to generate an environment from a program. The PEN file contains compressed symbol table information and is used by subsequent Pascal compilations with the INHERIT attribute.

Neither the VSI OpenVMS Linker or ANALYZE/OBJECT reads environment files. Only the VSI Pascal compiler uses these files. The OBJ file contains the following:

- Storage for variables declared at the outermost level of the module
- Code for procedures/functions contained in the module
- Linker timestamp verification for all inherited environment files and for the environment file being created.
- Compiler-generated variables and routines to support schema types declared or discriminated at the outermost level of the module
- Code for the TO BEGIN DO or TO END DO sections (the module will have an invisible TO BEGIN DO section if it inherits an environment whose creating module also had a TO BEGIN DO or TO END DO section)

It is possible for the OBJ file to contain just the linker timestamp record for the environment being created. Since the linker does not require the timestamp record for correct behavior, you may not need the OBJ file created when generating a PEN file however, if the module is used inherited variables or routines, then the OBJ file is needed at link time. This applies not only to the environments inherited by the program, but also to any environments inherited by modules which create subsequent environments.

With the addition of schema types and TO BEGIN DO and TO END DO sections, the decision on which OBJ files to include on the **LINK** command becomes more complicated. This is due to:

- The compiler-generated variables and routines for schema types
- Any module initialization or finalization sections (TO BEGIN DO and TO END DO sections)
- The fact that currently PROGRAMs that inherit modules with initialization routines do not call the module's initialization routines directly but rely on the VSI OpenVMS LIB$INITIALIZE feature to activate them.

The rule is still basically "Use any OBJ whose module contained variables or routines" but now you must consider compiler-generated variables or routines, as well as user-defined variables or routines. Depending on the modules involved, missing OBJ files might not be detected by the linker, since PROGRAMs do not directly call their inherited initialization routines.

Compiling with the **/DEBUG** qualifier might require more OBJ files to be included on the **LINK** command. Normally, the compiler only uses the inherited compiler-generated variables and routines if the corresponding schema types are used in certain Pascal constructs. However, the Debug information generated to describe the schema types always requires the compiler-generated variables and routines. This is because a user might ask the Debugger to perform some operation on the variable that did not appear in the source code. Therefore, if you omitted an OBJ file on the **LINK** command and did not get a linker error, you might get an error if the **/DEBUG** qualifier was used.

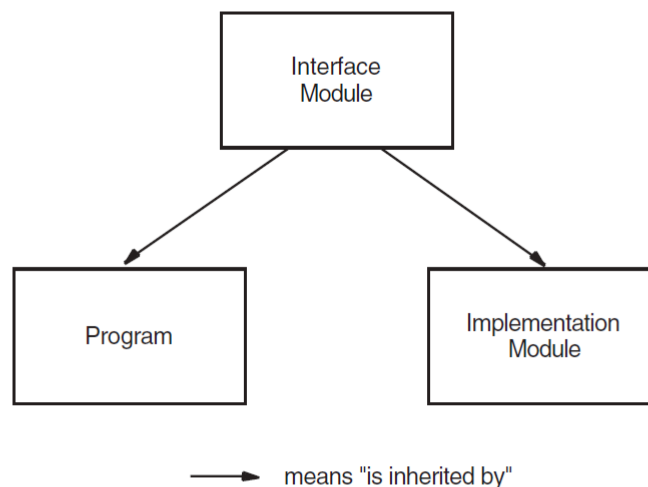
All OBJ files from modules creating environments must be included on the **LINK** command line if they contain variables or routines (either user-generated or compiler-generated). To receive the full benefit of the linker's timestamp verification, you should include all OBJ files on the **LINK** command.

2.2. Interfaces and Implementations

If your application requires, you can use a method of creating and inheriting environment files that minimizes the number of times you have to recompile compilation units. This method involves the division of module declarations into two separate modules: an interface module and an implementation module. The **interface module** contains data that is not likely to change: constant definitions, variable declarations, and external routine declarations. The **implementation module** contains data that may change: bodies of the routines declared in the interface module, and private types, variables, routines, and so forth.

The interface module creates the environment file that is inherited by both the implementation module and by the program. Figure 2.2 shows the inheritance process.

Figure 2.2. Inheritance Path of an Interface, an Implementation, and a Program



Consider this code fragment from the interface module in Example 2.1 (see Section 2.4):

```
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface( OUTPUT );

   {Globally accessible type}

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;

   {Module initialization section}

END.
```

The code contained in the interface is not likely to change often. The implementation code can change without requiring recompilation of the other modules in the application. Consider this code fragment from the implementation module in Example 2.2 (see Section 2.4):

```
[INHERIT( 'Interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;
```

To compile, link, and run the code in Examples 2.1, 2.2, and 2.3 (the main program), use the following commands:

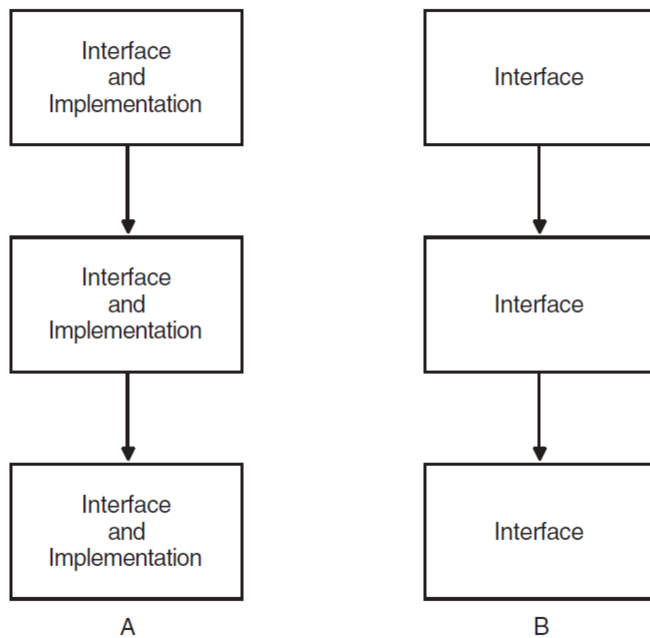
```
$ PASCAL GRAPHICS_INTERFACE
$ PASCAL GRAPHICS_IMPLEMENTATION
$ PASCAL GRAPHICS_MAIN_PROGRAM
$ LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$ GRAPHICS_INTERFACE
$ RUN GRAPHICS_MAIN_PROGRAM
```

If you need to change the code contained in any of the routine bodies declared in the implementation module, you do not have to recompile the program to reflect the changes. For example, if you have to edit the implementation module, you can regenerate the application with the following commands:

```
$ EDIT GRAPHICS_IMPLEMENTATION
$ PASCAL GRAPHICS_IMPLEMENTATION
$ LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$ GRAPHICS_INTERFACE
$ RUN GRAPHICS_MAIN_PROGRAM
```

In this manner, interfaces and implementations can save you maintenance time and effort. In addition, the interface and implementation design allows you to better predict when cascading inheritance may provide maintenance problems. Figure 2.3 shows two forms of cascading.

Figure 2.3. Cascading Using the Interface and Implementation Design



If the compilation units creating environment files are designed to contain both interface and implementation declarations, the cascading in column A may lead to more recompiling, more relinking, and more multiply declared identifiers. The design shown in column B does not always provide easy maintenance, but it is more likely to do so. For example, if each interface provided a different kind of constant or type (as determined by your application) and if the constants and types are not derived from one another, the inheritance path in column B may be quite efficient and orderly, and may require little recompiling and relinking.

Do not place the following in an implementation module:

- Nonstatic types and variables at the module level
- A module initialization section (TO BEGIN DO)
- A module finalization section (TO END DO)

These restrictions are necessary because VSI Pascal cannot determine the order of activation of initialization and finalization sections that do not directly follow an environment-file inheritance path. Since implementation modules do not create environment files, the initialization and finalization sections in those modules are effectively outside of any inheritance path. Also, if you use the previously listed objects in implementation modules, there may be attempts to access data that has not yet been declared. Consider the following example:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
PROCEDURE x; EXTERNAL;
END.


{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );
VAR
```

```
   My_String : STRING( 10 );

[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

TO BEGIN DO
   My_String := 'Okay';
END.
```

In the previous example, it is possible for you to call procedure `x` (in some other module that also inherits INTERFACE.PEN) before the creation and initialization of the `variable My_String`. You can circumvent this problem by using a routine call to initialize the variable and by moving the code to the interface module, as shown in the next example:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
VAR
   My_String : STRING( 10 );

PROCEDURE x; EXTERNAL;
PROCEDURE Initialize; EXTERNAL;

TO BEGIN DO
   Initialize;
END.


{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );

[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

[GLOBAL] PROCEDURE Initialize;
   BEGIN
   My_String := 'Okay';
   END;
END.
```

2.3. Data Models

Using separate compilation and a few other features of VSI Pascal (including initial states, constructors, the HIDDEN attribute, and TO BEGIN DO and TO END DO sections), you can construct models for creating, distributing, isolating, and restricting data in an application.

Of course, the design of the data model depends on the needs of a particular application. However, to show some of the power of VSI Pascal features used in conjunction, Examples 2.1, 2.2, and 2.3 in Section 2.4 create a generic graphics application. Consider the following code fragment from Example 2.1:

```
TYPE
```

```
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                    {Starting coordinate points}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                           {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
```

The interface module provides an interface to the rest of the application. This module contains types and external procedure declarations that the data model chooses to make available to other compilation units in the application; other units can access these types and routines by inheriting the generated environment file.

The type `Shape_Types` defines two legal graphical objects for this application: a circle and a rectangle. The type `Shape` can be used by other units to create circles and rectangles of specified dimensions. This code uses a variant record to specify the different kinds of data needed for a circle (a radius value) and a rectangle (height and width values).

Since the type has initial-state values, any variable declared to be of this type receives these values upon declaration. Providing initial states for types that are included in environment files can prevent errors when other compilation units try to access uninitialized data.

The initial states in this code are specified for the individual record values. You can also provide an initial state for this type using a constructor, as follows:

```
   Shape( t : Shape_Types ) = RECORD
      Coordinate_X, Coordinate_Y : REAL;
      CASE t OF
         Square : ( Height, Width : REAL );
         Circle : ( Radius        : REAL );
      END VALUE [ Coordinate_X : 50.0; Coordinate_Y : 50.0;
                  CASE Circle OF [ Radius : 5.0 ] ];
```

If you use constructors for variant records, you can only specify an initial state for one of the variant values. If you need to specify initial states for all variant values, you must specify the initial states on the individual variants, as shown in Example 2.1.

The interface module also declares routines that can draw, rotate, scale, and delete an object of type `Shape`. The bodies of these routines are located in the implementation module. The interface module also contains a TO BEGIN DO section, as shown in the following code fragment:

```
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
```

As with the other routines, the body of `Draw_Logo` is located in the implementation module. The HIDDEN attribute prevents compilation units that inherit the interface environment file from calling the `Draw_Logo` routine. This ensures that the application only calls `Draw_Logo` once at the beginning of the application.

Using this design, the interface module can provide graphical data and tools to be used by other compilation units without the other units having to worry about implementation details. The actual details are contained in one implementation module. For example, the routine bodies are contained in the implementation module. Consider the following code fragment from Example 2.2:

```
{Declare routine bodies for declarations in the interface}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}
```

The routine bodies of the external routines declared in the interface module are located in the implementation module. The code in each of the routines uses the actual discriminant of parameter `s` to determine if the shape is a circle or a rectangle and draws the shape. If this code needs to change, it does not require that you recompile the code in Examples 2.1 or 2.3 in Section 2.4.

Example 2.2 also contains code that is isolated and hidden from other compilation units that inherit the interface environment file. Consider the following code fragment from the interface module:

```
[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius       : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo.}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin.}
   END;
```

In the graphical data model, you may wish to define a company logo, and you may wish to display that logo on the screen before any other graphical data is drawn or displayed. This code declares the `variable Initial_Shape`. Since this variable is declared locally to `Draw_Logo` and since `Draw_Logo` is contained in a module that does not produce an environment file, other modules that may have access to the interface environment file do not have access to this variable. In this application, you may not wish to give other compilation units the power to alter the company logo.

The code in the interface's TO BEGIN DO section, which executes before any program code, displays the company logo and deletes it to begin the application. Consider again the compilation process for interfaces, implementations, and programs:

```
$ PASCAL GRAPHICS_INTERFACE
$ PASCAL GRAPHICS_IMPLEMENTATION
$ PASCAL GRAPHICS_MAIN_PROGRAM
```

```
$ LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$ GRAPHICS_INTERFACE
$ RUN GRAPHICS_MAIN_PROGRAM
```

VSI Pascal executes the TO BEGIN DO section according to the inheritance order of environment files. Remember that VSI Pascal cannot determine the order of execution for TO BEGIN DO sections contained in implementation modules, so do not use them there.

Using this design, you can allow different sites that run the graphics application to access global data through the interface module. One location can maintain and control the contents of the implementation module, shipping the implementation's object module for use at other sites. You can use this method for other types of sensitive data or data that needs to be maintained locally.

2.4. Separate Compilation Examples

Example 2.1 shows an interface module that creates the environment file INTERFACE.PEN. This environment file is inherited in Examples 2.2 and in 2.3.

Example 2.1. An Interface Module for Graphics Objects and Routines

```
{
Source File: graphics_interface.pas
This module creates an interface to graphical data and routines.
}
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface;
TYPE
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                    {Starting coordinate points:}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                           {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
END.
```

Example 2.2 shows the implementation of the routines declared in Example 2.1.

Example 2.2. An Implementation Module for Graphics Objects and Routines

```
{
```

```
Source File: graphics_implementation.pas
This module implements the graphics routines and data declarations
made global by the interface module.
}
[INHERIT( 'Interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );

{Declare routine bodies for declarations in the interface:}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Scale( s : Shape );
   BEGIN
   WRITELN( 'Scaling the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Delete( s : Shape );
   BEGIN
   WRITELN( 'Deleting the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius       : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo.}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin.}
   END;
END.
```

Example 2.3 shows a main program and its use of the types and routines provided by the interface module.

Example 2.3. A Graphics Main Program

```
{
Source File: graphics_main_program.pas
This program inherits the interface environment file, which gives it
access to the implementation's declarations.
}
```

```
[INHERIT( 'Interface' )]  {Types and routines in interface module}
PROGRAM Graphics_Main_Program( OUTPUT );

VAR
   My_Shape : Shape( Rectangle )
      VALUE [ Coordinate_X : 25.0;
              Coordinate_Y : 25.0;
              CASE Rectangle OF
              [Height : 12.50; Width : 25.63]];
BEGIN
{
You cannot access the variable Initial_Shape, because it is in the
implementation module, and that module does not create an environment
file.

You can work with My_Shape.  If you did not provide initial values in
this declaration section, the module Graphics_Interface provided
initial values for the schema type Shape.
}
Draw(   My_Shape );
Scale(  My_Shape );
Rotate( My_Shape );
Delete( My_Shape );
END.
```

To compile, link, and run the code in Examples 2.1, 2.2, and 2.3, enter the following:

```
$ PASCAL GRAPHICS_INTERFACE
$ PASCAL GRAPHICS_IMPLEMENTATION
$ PASCAL GRAPHICS_MAIN_PROGRAM
$ LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$ GRAPHICS_INTERFACE
$ RUN GRAPHICS_MAIN_PROGRAM
Drawing a company logo
Code that draws a circle
Deleting the shape :    CIRCLE
Code that draws a rectangle
Scaling the shape : RECTANGLE
Rotating the shape : RECTANGLE
Deleting the shape : RECTANGLE
```

Chapter 3. Program Correctness, Optimization, and Efficiency

This chapter discusses the following topics: compiler optimizations, programming considerations, and implementation-dependent behavior.

The objective of optimization is to produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory. Realizing this objective requires programs that are carefully designed and written, and compilation techniques, such as those used by VSI Pascal, that take advantage of the operating system and machine architecture environment. (The benefits of portable code and program efficiency depend on the requirements of your application.)

3.1. Compiler Optimizations

By default, programs compiled with the VSI Pascal compiler undergo optimization. An optimizing compiler automatically attempts to remove repetitious instructions and redundant computations by making assumptions about the values of certain variables. This, in turn, reduces the size of the object code, allowing a program written in a high-level language to execute at a speed comparable to that of a well-written assembly language program. Optimization can increase the amount of time required to compile a program, but the result is a program that may execute faster and more efficiently than a nonoptimized program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization. In addition, this awareness often makes it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The compiler performs the following optimizations:

- Compile-time evaluation of constant expressions
- Elimination of some common subexpressions
- Partial elimination of unreachable code
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predeclared functions
- Inline code expansion for user-declared routines
- Rearrangement of unary minus and NOT operations to eliminate unary negation and complement operations
- Partial evaluation of logical expressions
- Propagation of compile-time known values
- Strength reduction
- Split lifetime analysis

- Code scheduling
- Loop unrolling

These optimizations are described in the following sections. In addition, the compiler performs the following optimizations, which can be detected only by a careful examination of the machine code produced by the compiler:

- Global assignment of variables to registers

  If possible, reduce the number of memory references needed by assigning frequently referenced variables to registers.

- Reordering the evaluation of expressions

  This minimizes the number of temporary values required.

- Peephole optimization of instruction sequences

  The compiler examines code a few instructions at a time to find operations that can be replaced by shorter and faster equivalent operations.

Not all optimizations are available on all targets. Some of them depend on the target architecture and underlying code generator technology.

For More Information:

- On VSI Pascal language elements (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

3.1.1. Compile-Time Evaluation of Constants

The compiler performs the following computations on constant expressions at compile time:

- Negation of constants

  The value of a constant preceded by unary minus signs is negated at compile time. For example:

  ```
  x := -10.0;
  ```

- Type conversion of constants

  The value of a lower-ranked constant is converted to its equivalent in the data type of the higher-ranked operand at compile time. For example:

  ```
  x := 10 * y;
  ```

  If x and y are both real numbers, then this operation is compiled as follows:

  ```
  x := 10.0 * y;
  ```

- Arithmetic on integer and real constants

  An expression that involves +, –, *, or / operators is evaluated at compile time. For example:

  ```
  CONST
  ```

```
   nn = 27;
{In the executable section:}
i := 2 * nn + j;
```

This is compiled as follows:

```
i := 54 + j;
```

- Array address calculations involving constant indexes

  These are simplified at compile time whenever possible. For example:

```
VAR
   i : ARRAY[1..10, 1..10] OF INTEGER;
{In the executable section:}
i[1,2] := i[4,5];
```

- Evaluation of constant functions and operators

  Arithmetic, ordinal, transfer, unsigned, allocation size, CARD, EXPO, and ODD functions involving constants, concatenation of string constants, and logical and relational operations on constants, are evaluated at compile time.

For More Information:

- On the complete list of compile-time operations and routines (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

3.1.2. Elimination of Common Subexpressions

The same subexpression often appears in more than one computation within a program. For example:

```
a := b * c + e * f;

h := a + g - b * c;

IF ((b * c) - h) <> 0 THEN ...
```

In this code sequence, the subexpression b * c appears three times. If the values of the operands b and c do not change between computations, the value b * c can be computed once and the result can be used in place of the subexpression. The previous sequence is compiled as follows:

```
t := b * c;

a := t + e * f;

h := a + g - t;

IF ((t) - h) <> 0 THEN ...
```

Two computations of b * c have been eliminated. In this case, you could have modified the source program itself for greater program optimization.

The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect:

```
VAR
```

```
   a, b : ARRAY[1..25, 1..25] OF REAL;
{In the executable section:}
a[i,j] := b[i,j];
```

Without optimization, this source program would be compiled as follows:

```
t1 := (j - 1) * 25 + i;
t2 := (j - 1) * 25 + i;
a[t1] := b[t2];
```

Variables t1 and t2 represent equivalent expressions. The compiler eliminates this redundancy by producing the following optimization:

```
t = (j - 1) * 25 + i;
a[t] := b[t];
```

3.1.3. Elimination of Unreachable Code

The compiler can determine which lines of code, if any, are never executed and eliminates that code from the object module being produced. For example, consider the following lines from a program:

```
CONST
   Debug_Switch = FALSE;
{In the executable section:}
IF  Debug_Switch  THEN  WRITELN( 'Error found here' );
```

The IF statement is designed to write an error message if the value of the symbolic constant Debug_Switch is TRUE. Suppose that the error has been removed, and you change the definition of Debug_Switch to give it the value FALSE. When the program is recompiled, the compiler can determine that the THEN clause will never be executed because the IF condition is always FALSE; no machine code is generated for this clause. You need not remove the IF statement from the source program.

Code that is otherwise unreachable, but contains one or more labels, is not eliminated unless the GOTO statement and the label itself are located in the same block.

3.1.4. Code Hoisting from Structured Statements

The compiler can improve the execution speed and size of programs by removing invariant computations from structured statements. For example:

```
FOR j := 1 TO i + 23 DO
   BEGIN
   IF Selector THEN a[i + 23, j - 14] := 0
   ELSE b[i + 23, j - 14] := 1;
   END;
```

If the compiler detected this IF statement, it would recognize that, regardless of the Boolean value of Selector, a value is stored in the array component denoted by [i + 23, j – 14]. The compiler would change the sequence to the following:

```
t := i + 23;
FOR j := 1 TO t DO
   BEGIN
   u := j - 14;
   IF Selector THEN a[t,u] := 0
   ELSE b[t,u] := 1;
```

```
END;
```

This removes the calculation of j – 14 from the IF statement, and the calculation of i + 23 from both the IF statement and the loop.

3.1.5. Inline Code Expansion for Predeclared Functions

The compiler can often replace calls to predeclared routines with the actual algorithm for performing the calculation. For example:

```
Square := SQR( a );
```

The compiler replaces this function call with the following, and generates machine code based on the expanded call:

```
Square := a * a;
```

The program executes faster because the algorithm for the SQR function has already been included in the machine code.

3.1.6. Inline Code Expansion for User-Declared Routines

Inline code expansion for user-declared routines performs in the same manner as inline code expansion for predeclared functions: the compiler can often replace calls to user-declared routines with an inline expansion of the routine's executable code. Inline code expansion is useful on routines that are called only a few times. The overhead of an actual procedure call is avoided, which increases program execution. The size of the program, however, may increase due to the routine's expansion.

To determine whether or not it is desirable to inline expand a routine, compilers use a complex algorithm.

3.1.7. Operation Rearrangement

The compiler can produce more efficient machine code by rearranging operations to avoid having to negate and then calculate the complement of the values involved. For example:

```
(-c) * (b - a)
```

If a program includes this operation, the compiler rearranges the operation to read as follows:

```
c * (a - b)
```

These two operations produce the same result, but because the compiler has eliminated negation or complement operations, the machine code produced is more efficient.

3.1.8. Partial Evaluation of Logical Expressions

The Pascal language does not specify the order in which the components of an expression must be evaluated. If the value of an expression can be determined by partial evaluation, then some subexpressions may not be evaluated at all. This situation occurs most frequently in the evaluation of logical expressions. For example:

```
WHILE ( i < 10 ) AND ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

In this WHILE statement, the order in which the two subexpressions (i < 10) and (a[i] <> 0) are evaluated is not specified; in fact, the compiler may evaluate them simultaneously. Regardless of which subexpression is evaluated first, if its value is FALSE the condition being tested in the WHILE statement is also FALSE. The other subexpression need not be evaluated at all. In this case, the body of the loop is never executed.

To force the compiler to evaluate expressions in left-to-right order with short circuiting, you can use the AND_THEN operator, as shown in the following example:

```
WHILE ( i < 10 ) AND_THEN ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

3.1.9. Value Propagation

The compiler keeps track of the values assigned to variables and traces the values to most of the places that they are used. If it is more efficient to use the value rather than a reference to the variable, the compiler makes this change. This optimization is called value propagation. Value propagation causes the object code to be smaller, and may also improve run-time speed.

Value propagation performs the following actions:

- It allows run-time operations to be replaced with compile-time operations. For example:

  ```
  Pi := 3.14;
  Pi_Over_2 := Pi/2;
  ```

  In a program that includes these assignments, the compiler recognizes the fact that Pi's value did not change between the time of Pi's assignment and its use. So, the compiler would use Pi's value instead of a reference to Pi and perform the division at compile time. The compiler treats the assignments as if they were as follows:

  ```
  Pi := 3.14;
  Pi_Over_2 := 1.57;
  ```

  This process is repeated, allowing for further constant propagation to occur.

- It allows comparisons and branches to be avoided at run time. For example:

  ```
  x := 3;
  IF x <> 3 THEN  y := 30
  ELSE y := 20;
  ```

  In a program that includes these operations, the compiler recognizes that the value of x is 3 and the THEN statement cannot be reached. The compiler will generate code as if the statements were written as follows:

  ```
  x := 3;
  y := 20;
  ```

3.1.10. Strength Reduction (VSI OpenVMS I64 and VSI OpenVMS Alpha systems)

Strength reduction speeds computations by replacing a multiply operation with a more efficient add instruction when computing array addresses each time around a loop.

3.1.11. Split Lifetime Analysis

Split lifetime analysis improves register usage by determining if the lifetime of a variable can be broken into multiple, independent sections. If so, the variable may be stored in different registers for each section. The registers can then be reused for other purposes between sections. Therefore, there may be times when the value of the variable does not exist anywhere in the registers. For example:

```
v:= 3.0 *q;
    .
    .
    .
x:= SIN(y)*v:
    .
    .
    .
v:= PI*x:
    .
    .
    .
y:= COS(y)*v;
```

This example shows that the variable v has two disjoint usage sections. The value of v in the first section does not affect the value of v in the second section. The compiler may use different registers for each section.

3.1.12. Code Scheduling

Code scheduling is a technique for reordering machine instructions to maximize the amount of overlap of the multiple execution units inside the CPU. The exact scheduling algorithms depend on the implementation of the target architecture.

3.1.13. Loop Unrolling

Loop unrolling is a technique for increasing the amount of code between branch instructions and labels by replicating the body of a loop. Increasing the code optimizes instruction scheduling. The following code shows such a transformation:

Original Code

```
FOR i:= 1 to 12 DO
    a[i]:= b[i] + c[i]
```

Unrolled Loop Code

```
i:= 1
WHILE i < 12 DO
    BEGIN
    a[i]:= b[i] + c[i];
```

```
    a[i+1]:= b[i+1] + c[i+1];
    a[i+2]:= b[i+2] + c[i+2];
    a[i+3]:= b[i+3] + c[i+3];
    i:= i+4;
    END;
```

In this example, the loop body was replicated four times, allowing the instruction scheduler to overlap the fetching of array elements with the addition of other array elements.

By default, loop unrolling makes 4 copies of an unrolled loop. You can change the number of copies from 1 to 16. This is controlled by:

`/OPTIMIZE=UNROLL="number"`

Numbers larger than 4 may improve performance at a cost of additional code size. However, larger numbers may decrease performance due to cache requirements, register conflicts, and other factors.

3.1.14. Alignment of Compiler-Generated Labels

The compiler aligns the labels it generates for the top of loops, the beginnings of ELSE branches, and others, on machine-specific boundaries by filling in unused bytes with NO-OP instructions.

A branch to a longword-aligned address is faster than a branch to an unaligned address. This optimization may increase the size of the generated code; however, it increases run-time speed.

3.1.15. Error Reduction Through Optimization

An optimized program produces results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different statements in the source program. For example:

Unoptimized Code	Optimized Code
`a := x/y;`	`t := x/y;`
`b := x/y;`	`a := t;`
`FOR i := 1 TO 10 DO`	`b := t;`
`     c[i] := c[i] * x/y;`	`FOR i := 1 TO 10 DO` `          c[i] := c[i] * t;`

If the value of y is 0.0, the unoptimized program would produce 12 divide-by-zero errors at run time; the optimized program produces only one. (The variable t is a temporary variable created by the compiler.) Eliminating redundant calculations and removing invariant calculations from loops can affect the detection of such arithmetic errors. You should keep this in mind when you include error-detection routines in your program.

3.1.16. Processor Selection and Tuning (VSI OpenVMS Alpha systems)

VSI Pascal provides support for generating code for specific Alpha processors and for tuning code for a preferred processor. The supported Alpha processors are EV4, EV5, EV56, EV6, EV7, EV67, and EV68.

The EV4 and EV5 processors are basically identical, with the only difference in the preferred instruction scheduling phase. The EV56 processor added byte and word opcodes. The EV6 processor added a

SQRT instruction, instructions to move data directly between floating and integer registers, and a few other instructions. The EV7 processor is similar to the EV6 processor with differences only in the instruction scheduling phase.

The default architecture (see the **/ARCHITECTURE** qualifier) is for the EV4 processor. This restricts the compiler to instructions that exist on the EV4 processor. It essentially tells the compiler the earliest Alpha processor that will execute the code. If you run the code on earlier Alpha systems, you might get invalid opcode errors or OpenVMS might attempt to emulate the instructions at a severe performance penalty.

The default tuning (see the **/OPTIMIZE=TUNE** qualifier) is "generic." The tuning is for an average Alpha processor. You can achieve better performance it you allow the compiler to tune the code for a specific processor.

Specifying an explicit **/ARCHITECTURE** setting also defaults the **/OPTIMIZE=TUNE** setting to the same processor.

For example, specifying **/ARCHITECTURE=EV56/OPTIMIZE=TUNE=EV7** tells the compiler to use instructions that the generated code should be able to run on an EV56 system, but that it should tune the generated code for best performance on an EV7 system. In these situations, the compiler can actually generate multiple code sequences, one using only EV56 instructions, and the other using EV7 instructions and the AMASK instruction to dynamically execute the faster sequence based on the system executing the program.

Since most Alpha systems are EV56 or later, you might see a significant improvement by specifying **/ARCHITECTURE=EV56** on the command line.

3.1.17. Compiling for Optimal Performance

The following command lines will result in producing the fastest code from the compiler. Depending on the system, use one of the following:

For VSI OpenVMS I64 and OpenVMS x86-64 systems, use:

```
PASCAL /NOZERO_HEAP /OPT=LEVEL=4 /NOCHECK
```

For VSI OpenVMS Alpha systems, use:

```
PASCAL /NOZERO_HEAP /MATH_LIBRARY=FAST /OPT=LEVEL=4 /NOCHECK /ARCH=HOST
/ASSUME=NOACCURACY_SENSITIVE
```

In both cases, you may also want to use the performance flagger to identify datatypes that could be modified for additional performance.

For More Information:

- On performance flagger (Section 1.1.2)

3.2. Programming Considerations

The language elements that you use in a source program directly affect the compiler's ability to optimize the resulting object program. Therefore, you should be aware of the following ways in which you can assist compiler optimization and obtain a more efficient program:

- Define constant identifiers to represent values that do not change during your program. The use of constant identifiers generally makes a program easier to read, understand, and later modify. In addition, the resulting object code is more efficient because symbolic constants are evaluated only once, at compile time, while variables must be reevaluated whenever they are assigned new values.
- Whenever possible, use the structured control statements CASE, FOR, IF-THEN-ELSE, REPEAT, WHILE, and WITH rather than the GOTO statement. You can use the GOTO statement to exit from a loop, but careless use of it interferes with both optimization and the straightforward analysis of program flow.
- Enclose in parentheses any subexpression that occurs frequently in your program. The compiler checks whether any assignments have affected the subexpression's value since its last occurrence. If the value has not changed, the compiler recognizes that a subexpression enclosed in parentheses has already been evaluated and does not repeat the evaluation. For example:

  ```
  x := SIN( u + (b - c) ); y := COS( v + (b - c) );
  ```

  The compiler evaluates the subexpression (b – c) as a result of performing the SIN function. When it is encountered again, the compiler checks to see whether new values have been assigned to either b or c since they were last used. If their values have not changed, the compiler does not reevaluate (b – c).
- Once your program has been completely debugged, disable all checking with [CHECK(NONE)] or with the appropriate compilation switch. Recall that VSI Pascal enables bounds and declaration checking by default. When no checking code is generated, more optimizations can occur, and the program executes faster.

  Integer overflow checking is disabled by default. If you are sure that your program is not in danger of integer overflow, you should not enable overflow checking. Because overflow checking precludes certain optimizations, you can achieve a more efficient program by leaving it disabled.
- When a variable is accessed by a program block other than the one in which it was declared, the variable should have static rather than automatic allocation. An automatically allocated variable has a varying location in memory; accessing it in another block is time-consuming and less efficient than accessing a static variable.
- When creating schema records (or records with nonstatic fields), place the fields with run-time size at the end of the record. The generated code has to compute the offset of all record fields after a field with run-time size, and this change minimizes the overhead.

For More Information:

- On VSI Pascal language elements and on attributes (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On compilation switches (Chapter 1)

3.3. Implementation-Dependent Behavior

The Pascal language has several implementation-dependent behaviors that a program must not rely upon. Relying on these behaviors for correct behavior is illegal and is not portable to other platforms or other compiler versions.

Refer to the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for a list of the implementation-dependent behaviors.

For More Information:

- On attributes and on static and automatic variables (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On compilation switches (Chapter 1)

3.3.1. Subexpression Evaluation Order

The compiler can evaluate subexpressions in any order and may even choose not to evaluate some of them. Consider the following subexpressions that involve a function with side effects:

```
IF f( a ) AND f( b ) THEN ...
```

This IF statement contains two designators for function f with the same parameter a. If f has side effects, the compiler does not guarantee the order in which the side effects will be produced. In fact, if one call to f returns FALSE, the other call to f might never be executed, and the side effects that result from that call would never be produced. For example:

```
q := f( a ) + f( a );
```

The Pascal standard allows a compiler to optimize the code as follows:

```
Q := 2 * f( a )
```

If the compiler does so, and function f has side effects, the side effects would occur only once because the compiler has generated code that evaluates f(a) only once.

If you wish to ensure left-to-right evaluation with short circuiting, use the AND_THEN and OR_ELSE Boolean operators.

For More Information:

- On the order of expression evaluation, see the description of the NOOPTIMIZE attribute (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

3.3.2. MAXINT and MAXINT64 Predeclared Constants

The smallest possible value of the INTEGER type is represented by the predeclared constant −MAXINT. The largest possible value of the INTEGER type is represented by the predeclared constant MAXINT. However, the underlying architecture supports an additional integer value, which is (−MAXINT −1). If your program contains a subexpression with this value, the program's evaluation might result in an integer overflow trap. Therefore, a computation involving the value (−MAXINT −1) might not produce the expected result. To evaluate expressions that include (−MAXINT −1), you should disable either optimization or integer overflow checking.

Similarly, (−MAXINT64 −1) might not produce the expected results.

3.3.3. Pointer References

The compiler assumes that the value of a pointer variable is either the constant identifier NIL or a reference to a variable allocated in heap storage by the NEW procedure. A variable allocated in heap storage is not declared in a VAR section and has no identifier of its own; you can refer to it only by the name of a pointer variable followed by a circumflex (^). Consider the following example:

```
VAR
   x : INTEGER;
   p : ^INTEGER;
{In the executable section:}
NEW( p );
p^ := 0;
x  := 0;
IF p^ = x THEN  p^ := p^ + 1;
```

If a pointer variable in your program must refer to a variable with an explicit name, that variable must be declared VOLATILE or READONLY. The compiler makes no assumptions about the value of volatile variables and therefore performs no optimizations on them.

Use of the ADDRESS function, which creates a pointer to a variable, can result in a warning message because of optimization characteristics. By passing a nonread-only or nonvolatile static or automatic variable as the parameter to the ADDRESS function, you indicate to the compiler that the variable was not allocated by NEW but was declared with its own identifier. Because the compiler's assumptions are incorrect, a warning message occurs. You can also use IADDRESS, which functions similarly to the ADDRESS function except that IADDRESS returns an INTEGER_ADDRESS value and does not generate any warning messages. Use caution when using IADDRESS.

Similarly, when the parameter to ADDRESS is a formal VAR parameter or a component of a formal VAR parameter, the compiler issues a warning message that not all dynamic variables allocated by NEW may be passed to the function.

For More Information:

- On attributes and on predeclared routines (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

3.3.4. Variant Records

Because all the variants of a record variable are stored in the same memory location, a program can use several different field identifiers to refer to the same storage space. However, only one variant is valid at a given time; all other variants are undefined. You must store a value in a field of a particular variant before you attempt to use it. For example:

```
VAR
   x : INTEGER;
   a : RECORD
      CASE t : BOOLEAN OF
         TRUE   : ( b : INTEGER );
         FALSE  : ( c : REAL );
      END;
{In the executable section:}
x := a.b + 5;
a.c := 3.0;
x := a.b + 5;
```

Record a has two variants, b and c, which are located at the same storage address. When the assignment a.c := 3.0 is executed, the value of a.b becomes undefined because TRUE is no longer the currently valid variant. When the statement x := a.b + 5 is executed for the second time, the value of a.b is unknown. The compiler may choose not to evaluate a.b a second time because it has retained the field's previous value. To eliminate any misinterpretations caused by this assumption, variable a should be associated with the VOLATILE attribute. The compiler makes no assumptions about the value of VOLATILE objects.

For More Information:

- On variant records or on the VOLATILE attribute (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

3.3.5. Atomicity, Granularity, Volatility, and Write Ordering

When data is shared by multiple code streams (either multiple processes, multiple threads, or asynchronous events such as AST routines or condition handlers), you need to be aware of certain issues to guarantee correct sharing of data.

You must inform the compiler that the data being shared may change in an asynchronous fashion. By default, the compiler assumes that data is only modified by assignment statements, routine calls, etc. If the data is being changed in a way that the compiler does not know about you must use the VOLATILE attribute to tell the compiler that it must fetch the data in an atomic fashion from memory at each reference and the compiler must store the data in an atomic fashion back into memory at each assignment.

To accomplish atomic access on VSI OpenVMS x86-64 systems often requires no additional work by the compiler. The underlying x86-64 architecture's memory model provides atomic-like access in many situations. In other places, the compiler might have to use the x86-64 lock instruction prefix or the cmpxchg64 instruction.

To accomplish atomic access on VSI OpenVMS I64 systems for volatile objects 64 bits or smaller, fetches and stores are done with the normal `ld`*n* and `st`*n* instructions.

To accomplish atomic access on VSI OpenVMS Alpha systems for volatile objects smaller than 32 bits, fetches and stores are done with the LDx_L/STx_C instruction sequence. This pair of instructions ensures that the volatile data is accessed in an atomic fashion. Without the VOLATILE attribute, you will not get this special instruction sequence, and the data might become corrupted if two writers are trying to store to the shared data at the same time. Items of 32 bits or 64 bits are accessed with single longword and quadword instructions and do not use the LDx_L/STx_C sequence. Newer Alpha systems include byte and word instructions. See the **`/ARCHITECTURE`** qualifier for more information. Only aligned data objects are guaranteed to be accessed atomically. Larger objects that are manipulated with run-time routines are not atomic, as those routines may be interrupted.

Granularity is a term on Alpha machines to describe the situation where two threads update nearby data at the same time. Because the compiler on the older Alpha must fetch the surrounding longword or quadword, modify it, and store it back, the two threads could possibly overwrite each others data. For these situations, the nearby data should be moved to separate quadwords or use the **`/GRANULARITY`** qualifier to tell the compiler that you want longword or byte granularity at the expense of additional LDx_L/STx_C sequences. (See the **`/ARCHITECTURE`** qualifier for more information on the byte and word instructions available on newer Alpha systems.)

Besides atomic accesses, many programs want to perform atomic operations on shared data. To facilitate this, VSI Pascal provides the following built-in routines:

- ADD_INTERLOCKED (expr,variable)

  This routine adds the expression to the aligned word variable and returns -1 if the new value is negative, 0 if it is zero, or 1 if it is positive. On VSI OpenVMS I64 systems, it uses the cmpxchg instruction. On VSI OpenVMS Alpha systems, it uses the LDx_L/STx_C instructions. On VSI OpenVMS x86-64 systems, it uses the cmpxchgw instruction along with a lock prefix.

- CLEAR_INTERLOCKED (Boolean-variable)

  SET_INTERLOCKED(Boolean-variable)

  These routines clear or set a Boolean variable, respectively, and return the original value. On VSI OpenVMS I64 systems, they use the cmpxchg instruction. On VSI OpenVMS Alpha systems, they use the LDx_L/STx_C instructions. On VSI OpenVMS x86-64 systems, they use the cmpxchgq instruction along with a lock prefix.

- ADD_ATOMIC(expr,variable)

  AND_ATOMIC(expr,variable)

  OR_ATOMIC(expr,variable)

  These routines atomically add/and/or the value of the expression with the variable and return the original value. On VSI OpenVMS I64 systems, they use the cmpxchg instruction. On VSI OpenVMS Alpha systems, they use the LDx_L/STx_C instructions. On VSI OpenVMS x86-64 systems, they use the cmpxchgq instruction along with a lock prefix.

The Alpha, Itanium, and Intel-64 (x86-64) architectures do not guarantee that independent writes will complete in the order in which they were issued. These architectures provide a special instruction to serialize write operations. VSI Pascal provides the BARRIER built-in routine on these systems to generate the MB instruction on Alpha systems. The mf instruction on Itanium systems or the mfence instruction on x86-64 systems are used to preserve write ordering.

If your code uses a higher-level synchronization scheme to guard critical regions (such as a lock manager or a semaphore package), then using the VOLATILE attribute, the **/GRANULARITY** qualifier, and the INTERLOCKED/ATOMIC built-ins may not be necessary; you have already ensured that there are only single readers/writers in the critical section.

3.3.6. Debugging Considerations

Some of the effects of optimized programs on debugging are as follows:

- Use of registers

  When the compiler determines that the value of an expression does not change between two given occurrences, it may save the value in a register. In such a case, it does not recompute the value for the next occurrence, but assumes that the value saved in the register is valid. If, while debugging the program, you attempt to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register is not. When execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

  When the value of a variable is being held in a register, its value in memory is generally invalid; therefore, a spurious value may be displayed if you try to examine a variable under these circumstances.

- Coding order

  Some of the compiler optimizations cause code to be generated in a order different from the way it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when you try to step by line, use source display features, or examine or deposit variables.

- Inline code expansion on user-declared routines

  There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline user-declared routine is difficult and is not recommended.

To prevent conflicts between optimization and debugging, you should always compile your program with a compilation switch that deactivates optimization until it is thoroughly debugged. Then you can recompile the program (which by default is optimized) to produce efficient code.

For More Information:

- On debugging tools (Chapter 4)
- On compilation switches (Chapter 1)

Chapter 4. Programming Tools

This chapter describes some Pascal-specific assistance provided for selected tools. For general information on each tool, see the documentation for the tool.

4.1. Debugger Support for VSI Pascal for OpenVMS

In general, the debugger supports the data types and operators of VSI Pascal and the other debugger-supported languages. However, there are important language-specific limitations. (To get information on the supported data types and operators of any of the languages, enter the **`HELP LANGUAGE`** command at the `DBG>` prompt.)

In general, you can examine, evaluate, and deposit into variables, record fields, and array components. An exception to this occurs under the following circumstances: if a variable is not referenced in a program, the VSI Pascal compiler may not allocate the variable. If the variable is not allocated and you try to examine it or deposit into it, you will receive an error message.

When depositing data into variables, the debugger truncates the high-order bits if the value being deposited is larger than the variable; it fills the high-order bits with zeros if the value being deposited is smaller than the variable. If the deposit violates the rules of assignment compatibility, the debugger displays an informational message.

Automatic variables (within any active block) can be examined and can have values deposited into them; however, since automatic variables are allocated in stack storage and are contained in registers, their values are considered undefined until the variables are initialized or assigned a value. For example:

```
DBG> EXAMINE X
MAINP\X: 2147287308
```

In this example, the value of variable X should be considered undefined until after a value has been assigned to X.

In addition, you may examine a VARYING OF CHAR string, but it is not possible to examine the LENGTH field. For example, the following is not supported:

```
DBG> EXAMINE VARY_STRING.LENGTH
```

Because the current LENGTH of a VARYING string is the first word, you should do the following to examine the LENGTH:

```
DBG> EXAMINE/WORD VARY_STRING
```

It should also be noted that the type cast operator (::) is not permitted when evaluating VSI Pascal expressions.

Pointers to undiscriminated schema cannot be correctly described to the debugger at this time since the type of the pointer is dependent upon the value pointed to by the pointer. They are described as pointers to UNSIGNED integers. For example,

```
 TYPE S(I:INTEGER) = ARRAY [1..I] OF INTEGER;
   VAR P : ^S;
   BEGIN
```

```
NEW(P,expression);
END;
```

4.2. Language-Sensitive Editor/Source Code Analyzer Support for VSI Pascal for OpenVMS

This section describes VSI Pascal specific information for the following Language-Sensitive Editor/Source Code Analyzer (LSE/SCA) features:

- Programming language placeholders and tokens
- Placeholder processing
- Design comment processing (OpenVMS VAX systems)

4.2.1. Programming Language Placeholders and Tokens

LSE accepts keywords, or tokens, for all languages with LSE support, but the specific tokens themselves are language-defined. For example, you can expand the %INCLUDE token only when using VSI Pascal.

Likewise, LSE provides placeholders, or prompt markers, for all languages with LSE support, but the specific text or choices these markers call for are language-defined. For example, you see the %{environ_name_string}% placeholder only when using VSI Pascal.

Some VSI Pascal keywords, like TYPE, VAR, IF, and FOR, can be placeholders as well as tokens. LSE supplies language constructs for these keywords when they appear on your screen as placeholders. You can also type the keywords into the buffer yourself, enter the **EXPAND** command, and see the same language constructs appear on your screen.

You can use the **SHOW TOKEN** and **SHOW PLACEHOLDER** commands to display a list of all VSI Pascal tokens and placeholders, or a particular token or placeholder. For example:

```
LSE> SHOW TOKEN IF         {lists the token IF}
LSE> SHOW TOKEN            {lists all tokens  }
```

To copy the listed information into a separate file, first enter the appropriate **SHOW** command to put the list into the $SHOW buffer. Then enter the following command:

```
LSE> GOTO BUFFER $SHOW
LSE> SAVE FILE filename.filetype
```

4.2.2. Placeholder and Design Comment Processing

While all languages with LSE support provide placeholder processing, each language defines specific contexts in which placeholders can be accepted as valid program code. VSI Pascal defines contexts for declaration section placeholders and executable section placeholders. Table 4.1 lists the valid contexts within an VSI Pascal declaration section where you can insert placeholders.

Table 4.1. Placeholders Within the Declaration Section

Can Replace	Cannot Replace
PROGRAM or MODULE identifier	Directive

Can Replace	Cannot Replace
Program parameter	Attribute
Identifier	Declaration-begin reserved word
Data type	Complete declaration
Value	
Complete variant within the variant part of record	

Table 4.2 lists valid contexts within an VSI Pascal executable section where you can insert placeholders.

Table 4.2. Placeholders Within the Executable Section

Can Replace	Cannot Replace
Statement	LABEL identifier
Variable	TO | DOWNTO within a FOR statement
Expression	
Case label	
Complete case expression	
Iteration variable within a FOR statement	

VSI Pascal support for placeholder and design comment processing includes the following language-specific stipulations:

- Pseudocode placeholders are designated with double left- and right-angle brackets (<< >>) or the 8 bit format (« »).
- The compiler produces an empty object file when it encounters pseudocode or LSE placeholders within a source program.
- Comment processing is limited to the declaration section.

4.2.3. LSE and SCA Examples

Example 4.1 shows how you can use LSE tokens and placeholders to create a FOR statement within an VSI Pascal program. The callout numbers identify the steps in this process, which are detailed in the notes appearing after the example.

Example 4.1. Using LSE to Create a FOR Statement

```
  BEGIN
❶ %[statement_list]%...
 END.
     .
     .
     .
 BEGIN
❷ FOR %{control_var}% %{iteration_clause}% DO
     %{statement}%;
 %[statement_list]%...
 END.
```

```
    .
    .
    .
 BEGIN
 FOR INDEX := 1 TO MAX DO
❸     %{statement}%;
 %[statement_list]%...
 END.
    .
    .
    .
 BEGIN
 FOR INDEX := 1 TO MAX DO
❹     %{variable | func_id}% := %{value_expr}%;
 %[statement_list]%...
 END.
    .
    .
    .
 BEGIN
❺ FOR INDEX := 1 TO MAX DO
     ARR[INDEX] := 0;
 %[statement_list]%...
 END.
```

❶ As you begin the executable section of your program, the cursor rests on the placeholder %[statement_list]%. Type the token FOR over this placeholder and expand FOR.

❷ LSE provides the FOR statement template. Select a FOR variable option from the menu. Expand the %{iteration_clause}% placeholder and expand the %{statement}% placeholder.

❸ LSE displays a menu, from which you can select the %{simple_statement}% option. A further menu appears, from which you select the ASSIGNMENT statement option.

❹ LSE provides the assignment statement template. Type an appropriate identifier or value expression over each placeholder.

❺ The completed FOR statement appears in your buffer.

4.3. Accessing CDD/Repository from VSI Pascal for OpenVMS

The Oracle CDD/Repository (CDD/Repository) must be purchased separately. The CDD/Repository allows language-independent structure declarations that can be shared by many VSI OpenVMS layered products. VSI Pascal support of the CDD/Repository allows VSI Pascal programmers to share common record and data definitions with other VSI languages and data management products.

A system manager or data administrator creates the CDD/Repository's directory hierarchies, history lists, and access control lists with the Dictionary Management Utility (DMU). Once record paths are established, you can enter data definitions into and extract them from the CDD/Repository.

To enter data definitions into the CDD/Repository, you first create CDD/Repository source files written in the Common Data Dictionary Language (CDDL). The CDDL compiler converts the definitions to an internal form – making them independent of the language used to access them.

To extract data definitions from the CDD/Repository, include the %DICTIONARY directive in your VSI Pascal source program. If the data attributes of the data definitions are consistent with VSI Pascal requirements, the data definitions are included in the VSI Pascal program during compilation.

The %DICTIONARY directive incorporates CDD/Repository data definitions into the current VSI Pascal source file during compilation.

This directive can appear only in the TYPE section of an VSI Pascal program, not in the executable section. For example:

```
PROGRAM SAMPLE1;

TYPE
   %DICTIONARY 'Pascal_SALESMAN_RECORD/LIST'
   .
   .
   .
```

A **/LIST** option in the %DICTIONARY directive (or the **/SHOW=DICTIONARY** qualifier on the Pascal command line) includes the translated record in the program's listing. For example:

```
TYPE
   %DICTIONARY 'PASCAL_SALESMAN_RECORD/LIST'
   { CDD Path Name => PASCAL_SALESMAN_RECORD }

   PAYROLL_RECORD = PACKED RECORD
      SALESMAN    :  PACKED RECORD
      NAME        :  PACKED ARRAY [1..30] OF CHAR;
      ADDRESS     :  PACKED ARRAY [1..40] OF CHAR;
      SALESMAN_ID :  [BYTE(5)] RECORD END; { numeric string, unsigned }
      END; { record salesman }
   END; { record payroll_record }
```

The option (**/LIST** or **/NOLIST**) overrides the qualifier (**/SHOW=NODICTIONARY** or **/SHOW=DICTIONARY**).

For More Information:

- On CDD/Repository (CDD/Repository CDO Reference Manual, *Using CDD/Repository on VMS Systems*, and *CDD/Administrator User's Guide*)
- On the VSI Pascal %DICTIONARY directive (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

4.3.1. Equivalent VSI Pascal for OpenVMS and CDDL Data Types

The CDD/Repository supports some data types that are not native to VSI Pascal. If a data definition contains a field declared with an unsupported data type, VSI Pascal replaces the field with one declared as a [BYTE(n)] RECORD END, where n is the appropriate length in bytes. By making the data addressable in this way, you are able to manipulate the data either by passing it to external routines as variables or by using the VSI Pascal type casting capabilities to perform an assignment.

However, because these empty records do not have fields, the size of the record is 0 bits. They should not be used in expressions or passed to formal value parameters. Recall that a size attribute used on a type definition has no effect on fetches. When fetching from these records, the compiler will fetch the actual size of the record, 0 bits.

Table 4.3 summarizes the mapping between CDDL data types and the corresponding VSI Pascal data types.

Note

Although this practice is discouraged, you can use both D_floating and G_floating data types in the same compilation unit; however, both types cannot be handled in the same expression. Not all processors support the G_floating and H_floating types.

Table 4.3. Equivalent CDD/Repository Language and VSI Pascal for OpenVMS Data Types

CDDL Data Type	VSI Pascal for OpenVMS Data Type
Unspecified	[BYTE(n)] RECORD END
Byte logical	[BYTE] 0..255
Word logical	[WORD] 0..65535
Longword logical	UNSIGNED
Quadword logical	[BYTE(8)] RECORD END
Octaword logical	[BYTE(16)] RECORD END
Byte integer	[BYTE] –128..127
Word integer	[WORD] –32768..32767
Longword integer	INTEGER
Quadword integer	[BYTE(8)] RECORD END
Octaword integer	[BYTE(16)] RECORD END
F_floating	SINGLE
D_floating	DOUBLE (/NOG_FLOATING)
G_floating	DOUBLE (/G_FLOATING)
H_floating (OpenVMS VAX systems)	QUADRUPLE
F_floating complex	[BYTE (8)] RECORD END
D_floating complex	[BYTE(16)] RECORD END
G_floating complex	[BYTE(16)] RECORD END
H_floating complex	[BYTE(32)] RECORD END
Text	PACKED ARRAY [l..u] OF CHAR
Varying text	VARYING [u] OF CHAR
Numeric string, unsigned	[BYTE(n)] RECORD END
Numeric string, left separate	[BYTE(n)] RECORD END
Numeric string, left overpunch	[BYTE(n)] RECORD END
Numeric string, right separate	[BYTE(n)] RECORD END
Numeric string, right overpunch	[BYTE(n)] RECORD END
Numeric string, zoned sign	[BYTE(n)] RECORD END
Bit	[BIT(n)] 0..((2^n)–1) or [BIT(32)]UNSIGNED or [BIT(N)] RECORD END or ignored
Bit unaligned	[BIT(n), POS(x)] 0..((2^n)n–1) or

CDDL Data Type	VSI Pascal for OpenVMS Data Type
	[BIT(32), POS(x)] UNSIGNED or [BIT(n), POS(x)] RECORD END or ignored
Date and time Date	[BYTE(n)] RECORD END [BYTE(n)] RECORD END
Virtual field	Ignored
Varying string	VARYING [u] OF CHAR
Overlay	Variant record
Pointer	Pointer type

4.3.2. CDD/Repository Example

In Example 4.2, the %DICTIONARY directive is used to access the CDD/Repository record definition Mail_Order_Info. With this definition, the VSI Pascal program Show_Keys performs ISAM file manipulation on an existing indexed file, CUSTOMERS.DAT. Assume that CUSTOMERS.DAT has the primary key Order_Num and a field name called Zip_Code.

Note

Oracle CDD/Repository has no equivalent for the VSI Pascal KEY attribute, which is required to create new indexed files. You can use Oracle CDD/Repository data definitions to open existing indexed files (as in this example) but not new indexed files.

Example 4.2. Using %DICTIONARY to Access a Oracle CDD/Repository Record Definition

```
Program Show_Keys(OUTPUT);

TYPE
    %DICTIONARY 'Mail_Order_Info/LIST'

VAR
    Old_Customer_File   : FILE OF Mail_Order;
    Order_Rec           : Mail_Order;
    Continue            : BOOLEAN;

BEGIN
OPEN( File_Variable := Old_Customer_File,
      File_Name     := 'Customers.Dat',
      History       := OLD,
      Organization  := Indexed,
      Access_Method := Keyed );
FINDK(Old_Customer_File, 1, '1000', NXTEQL);
Continue := TRUE;
WHILE Continue and NOT UFB(Old_Customer_File) DO
    BEGIN
    READ(Old_Customer_File, Order_Rec);
    IF Order_Rec.Zip_Code < '5000'
    THEN
        WRITELN('Order number', Order_Rec.Order_Num, 'has zip code',
```

```
        Order_Rec.Zip_Code)
    ELSE
        Continue := False;
    END;
END.
```

During the compilation of Show_Keys, the record definition Mail_Order_Info is extracted from the CDD/Repository. Show_Keys prints the order number and zip code of each file component that has a zip code greater than or equal to 1000 but less than 5000.

Chapter 5. Calling Conventions

This chapter describes how VSI Pascal passes parameters and calls routines. It discusses the following topics: VSI OpenVMS calling standard, parameter-passing semantics and mechanisms, and passing parameters between VSI Pascal for OpenVMS and other languages.

For More Information:

- On declaring and calling VSI Pascal routines (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On procedure-calling and argument-passing mechanisms (*VSI OpenVMS Calling Standard*)

5.1. VSI OpenVMS Calling Standard

Programs compiled by the VSI Pascal compiler conform to the VSI OpenVMS calling standard. This standard describes how parameters are passed, how function values are returned, and how routines receive and return control. Because VSI Pascal conforms to the calling standard, you can call and pass parameters to routines written in other VSI languages from VSI Pascal programs.

For More Information:

- See the *VSI OpenVMS Calling Standard*

5.1.1. Parameter Lists

Each time a routine is called, the VSI Pascal compiler constructs a parameter list.

On VSI OpenVMS x86-64 systems, the parameters are a sequence of quadword (8-byte) entries. The first 6 integer parameters are located in integer registers designated as %rdi, %rsi, %rdx, %rcx, %r8, and %r9; the first 8 floating-pointer parameters are located in floating-point registers designated as %xmm0 to %xmm7. The packing of smaller values into these registers are further described in the *VSI OpenVMS Calling Standard*. Information about the parameter list is passed in the argument information register (%rax). The second byte of the register specifies the parameter count. Additional arguments are passed on the top of the stack.

On VSI OpenVMS I64 systems, the parameters are a sequence of quadword (8-byte) entries. The first 8 integer parameters are located in integer registers designated as R32 to R39; the first 8 floating-point parameters are located in floating-point registers designated as F8 to F15. Information about the parameter list is passed in the argument information register (R25). The first byte of the register specifies the parameter count. Arguments beyond 8 are passed on the stack starting at offset +16.

On VSI OpenVMS Alpha systems, the parameters are a sequence of quadword (8-byte) entries. The first 6 integer parameters are located in integer registers designated as R16 to R21; the first 6 floating-point parameters are located in floating-point registers designated as F16 to F21. Information about the parameter list is passed in the argument information register (R25). The first byte of the register specifies the parameter count. Arguments beyond 6 are passed on the top of the stack.

The form in which the parameters in the list are represented is determined by the passing mechanisms you specify in the formal parameter list and the values you pass in the actual parameter list. The parameter list contains the actual parameters passed to the routine.

5.1.2. Function Return Values

In VSI Pascal, a function returns to the calling block the value that was assigned to its identifier during execution. VSI Pascal chooses one of three methods for returning this value. The method chosen depends on the amount of storage required for values of the type returned, as follows:

On VSI OpenVMS x86-64 Systems:

- A nonfloating-point scalar type, a schematic subrange, an array, or set with size less than 64 bits, is returned in the first integer register, designated as %rax. If the value is less than 64 bits, %rax is sign-extended or zero-extended depending on the type.
- A record with size less than 128 bits will be returned in a combination of %rax, %rdx, or %xmm0 depending on the types of the record fields.
- If the value is too large to be returned with the first two bullets, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine’s actual parameter list.

On VSI OpenVMS I64 Systems:

- A nonfloating-point scalar type, a schematic subrange, an array, a record, or set with size less than 64 bits, is returned in the first integer register, designated as `r8`. If the value is less than 64 bits, `r8` is sign-extended or zero-extended depending on the type.
- A floating-point value that can be represented in 64 bits of storage is returned in the first floating-point register, designated as `f8`.
- If the value is too large to be represented in 64 bits, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine's actual parameter list.

On VSI OpenVMS Alpha Systems:

- A nonfloating-point scalar type, a schematic subrange, an array, a record, or set with size less than 64 bits, is returned in the first integer register, designated as `r0`. If the value is less than 64 bits, `r0` is sign-extended or zero-extended depending on the type.
- A floating-point value that can be represented in 64 bits of storage is returned in the first floating-point register, designated as `f0`.
- If the value is too large to be represented in 64 bits, if its type is a string type (PACKED ARRAY OF CHAR, VARYING OF CHAR, or STRING), or if the type is nonstatic, the calling routine allocates the required storage. An extra parameter (a pointer to the location where the function result will be stored) is added to the beginning of the calling routine's actual parameter list.

Note that functions that require the use of an extra parameter can have no more than 254 parameters; functions that store their results in registers can have 255 parameters.

5.1.3. Contents of the Call Stack

The VSI OpenVMS I64, VSI OpenVMS Alpha, and VSI OpenVMS x86-64 system conventions define three types of procedures. The calling process does not need to know what type it is calling; the compiler chooses which type to generate based on the requirements of the procedure.

On VSI OpenVMS x86-64 Systems:

On VSI OpenVMS x86-64 systems, the types of procedures are:

- Variable-size stack procedure (sometimes known as a normal procedure in industry x86-64 documentation) - allocates a memory stack that is addressable using either %rbp (the frame pointer register) or %rsp (the stack pointer register). The size of the stack may vary during the procedure execution. The called procedure may maintain a part or the whole context of its caller on that stack.
- Fixed-size stack procedure (sometimes known as a framepointerless procedure in industry x86-64 documentation) - allocates a memory stack that is addressable only using %rsp (the stack pointer register). The size of the stack is fixed during the procedure execution. The called procedure may maintain a part or the whole context of its caller on that stack.
- Fixed-size stack procedure (sometimes known as a framepointerless procedure in industry x86-64 documentation) - allocates a memory stack that is addressable only using %rsp (the stack pointer register). The size of the stack is fixed during the procedure execution. The called procedure may maintain a part or the whole context of its caller on that stack.

In order to provide accurate exception handling, the compiler always generates a frame pointer in the %rbp register. The compiler does not create routines without a frame pointer. The compiler also does not use the "red zone" as described in the OpenVMS Calling Standard.

On VSI OpenVMS x86-64 systems, the compiler determines the exact contents of the stack frame, but all stack frames have common characteristics:

- Saved frame pointer
- Return address
- Fixed temporary locations
- Argument home area if needed
- Register save area
- Arguments passed in memory

On VSI OpenVMS I64 Systems:

On VSI OpenVMS I64 systems, the types of procedures are:

- Memory stack procedures

  These procedures allocate a memory stack and may maintain part or all of its caller's context on that stack.

- Register stack procedures

  These procedures allocate only a register stack and maintains its caller's context in registers.

- Null frame procedures

  These procedures do not allocate a memory stack or a register stack and therefore preserve no context of its caller. However, unlike an VSI OpenVMS Alpha null frame procedure, these procedures do not execute in the context of its caller.

On VSI OpenVMS I64 systems, the compiler determines the exact contents of the memory stack frame, but all memory stack frames have common characteristics:

- Scratch area: A 16-byte region is provided as scratch storage for procedures that are called by the current procedure. Leaf procedures need not allocate this area. A procedure can use the 16 bytes pointed to by the stack pointer as scratch memory, but the contents of this area are not preserved by a procedure call.
- Arguments passed in memory: Parameters beyond those passed in registers are stored in this area of the memory stack frame. A procedure accesses its incoming parameters in the outgoing parameter area of its caller's memory stack frame.
- Local storage: A procedure can store local variables, temporaries, and spilled registers in this area. There are specific conventions that affect the layout for spilled registers.

On VSI OpenVMS Alpha Systems:

On VSI OpenVMS Alpha systems, the types of procedures are:

- Stack frame procedures, in which the calling context is placed on the stack
- Register frame procedures, in which the calling context is in registers
- No frame procedures, for which the compiler does not establish a context and which, therefore, execute in the context of the caller

If a stack frame is required, it consists of a fixed part (which is known at compile time) and an optional variable part.

The compiler determines the exact contents of the stack frame, but all stack frames have common characteristics:

- Fixed temporary locations: This is an optional section that contains language-specific locations required by the procedure context of some languages
- Register save area: This is a set of consecutive quadwords for storing registers saved and restored by the current procedure
- Argument home area: If allocated, this is a region of memory used by the called process to assemble the arguments passed in registers adjacent to the arguments passed in memory. This allows all arguments to be addressed as a contiguous array. The argument home area is also used to store arguments passed in registers if an address for such an argument is required.
- Arguments passed in memory

For More Information:

- On procedure types and characteristics (*VSI OpenVMS Calling Standard*)

5.1.4. Unbound Routines

The frame pointer of calling routines is stored in an implementation-defined register. If, however, you declare a routine with the UNBOUND attribute, the system does not assume that the frame pointer of the calling routine is stored in a register and there is no link between the calling and the called routines. As a result, an unbound routine has the following restrictions:

- It cannot access automatic variables declared in enclosing blocks.
- It cannot call bound routines declared in enclosing blocks.
- It cannot use a GOTO statement to transfer control to enclosing blocks other than the main program block.

By default, routines declared at program or module level and all other routines declared with the INITIALIZE, GLOBAL, or EXTERNAL attributes have the characteristics of unbound routines. Routines passed by the immediate value mechanism must be UNBOUND.

Asynchronous system trap routines (ASTs) and RMS completion routines must have both the ASYNCHRONOUS and UNBOUND attributes. Because they are asynchronous, such routines can access only volatile variables, predeclared routines, and other asynchronous routines. Note that the VSI Pascal run-time system does not permit a program and an asynchronous routine (such as an AST) to access the same file simultaneously.

For More Information:

- On attributes (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On the immediate value mechanism (Section 5.3.1)

5.2. Parameter-Passing Semantics

Parameter-passing semantics describe how parameters behave when passed between the calling and called routine. VSI Pascal passes parameter values by the following methods:

- Value passing semantics (Standard)
- Variable passing semantics (Standard)
- Foreign passing semantics (VSI Pascal extension)

By default, VSI Pascal passes arguments using value semantics.

For More Information:

- On value, variable, and foreign semantics (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

5.3. Parameter-Passing Mechanisms

The way in which an argument specifies how the actual data to be passed by the called routine is defined by the parameter-passing mechanism. In compliance with the VSI OpenVMS calling standard, VSI Pascal supports the basic parameter-passing mechanisms, shown in Table 5.1.

Table 5.1. Parameter-Passing Descriptions

Mechanism	Description
By immediate value	The argument contains the value of the data item.
By reference	The argument contains the address of the data to be used by the routine.

Mechanism	Description
By descriptor	The argument contains the address of a descriptor, which describes type of the data and its location.

By default, VSI Pascal uses the by reference mechanism to pass all actual parameters except those that correspond to conformant parameters and undiscriminated schema parameters, in which case the by descriptor mechanism is used. Table 5.2 describes the syntax you use in VSI Pascal to obtain the desired parameter-passing mechanism.

Table 5.2. Parameter-Passing Syntax on VSI Pascal

Mechanism	Syntax Used by VSI Pascal for OpenVMS
By immediate value	%IMMED or [IMMEDIATE]
By reference	Default for nonconformant and nonschema parameters or %REF
By descriptor	Default for conformant and schema parameters or %DESCR, %STDESCR, [CLASS_S],[CLASS_A], or [CLASS_NCA]

A mechanism specifier usually appears before the name of a formal parameter, or if a passing attribute is used it appears in the attribute list of the formal parameter. However, in VSI Pascal, a mechanism specifier can also appear before the name of an actual parameter. In the latter case, the specifier overrides the type, passing semantics, passing mechanism, and the number of formal parameters specified in the formal parameter declaration.

For More Information:

- On passing mechanisms and passing semantics (Section 5.3.4)

5.3.1. By Immediate Value

The by immediate value passing mechanism passes a copy of a value instead of the address. VSI Pascal provides the %IMMED foreign passing mechanism and the IMMEDIATE attribute in order to pass a parameter by immediate value. You cannot use variable semantics with the by immediate value passing mechanism.

Values that are less than or equal to 64 bits in size can be passed by immediate value.

On OpenVMS VAX systems, values that are less than or equal to 32 bits in size can be passed by immediate value.

5.3.2. By Reference

The by reference mechanism passes the address of the parameter to the called routine. This is the default parameter-passing mechanism for non-conformant and non-schematic parameters.

When using the by reference mechanism, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

In addition to using the defaults, the VSI Pascal compiler provides the %REF foreign passing mechanism and the REFERENCE attribute, which has more than one interpretation for the passing semantics depending on the data item represented by the actual parameter. This allows you to have the called

routine use either variable semantics or true foreign semantics. The mechanism specifier appears before the name of a formal parameter. The parameter passing attribute appears in the attribute list of the formal parameter.

5.3.3. By Descriptor

There are several types of descriptors. Each descriptor contains a value that identifies the descriptor's type. The called routine then uses the information held in the descriptor to identify its type and size. This is the default parameter-passing mechanism for conformant and schematic parameters.

When you use one of the VSI Pascal by descriptor mechanisms, the compiler passes the address of a string, array, or scalar descriptor. The VSI Pascal compiler generates the descriptor supplying the necessary information.

VSI Pascal provides three attributes for the by descriptor passing mechanism: [CLASS_S], [CLASS_A], and [CLASS_NCA]. With these three attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used. The parameter-passing attribute appears in the attribute list of the formal parameters.

Sometimes you may want to choose either variable semantics or true foreign semantics. In these cases, the VSI Pascal compiler provides two foreign passing mechanism specifiers, %DESCR and %STDESCR. These specifiers have more than one interpretation for the passing semantics depending on the data type of the actual parameter. The mechanism specifier appears before the name of a formal parameter.

Table 5.3 lists the class and type of descriptor generated for parameters that can be passed using the by descriptor mechanism.

Table 5.3. Parameter Descriptors

Parameter Type	Descriptor Class and Type		
	%DESCR	**%STDESCR**	**Value or VAR Semantics**
Ordinal	DSC$K_CLASS_S[1]	—	—
SINGLE	DSCK_CLASS_S, DSCK_DTYPE_F, DSC$K_DTYPE_FS	—	—
DOUBLE	DSCK_CLASS_S, DSCK_DTYPE_D, DSCK_DTYPE_G, DSCK_DTYPE_FT	—	—
QUADRUPLE	DSC$K_CLASS_S DSC$K_DTYPE_FX	—	—
RECORD	—	—	—
ARRAY	DSC$K_CLASS_NCA	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	—
ARRAY OF VARYING OF CHAR	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	—
Conformant ARRAY	DSC$K_CLASS_NCA [2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	DSC$K_CLASS_NCA

Parameter Type	Descriptor Class and Type		
Conformant ARRAY OF VARYING OF CHAR[4]	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	DSC$K_CLASS_VSA DSC$K_DTYPE_VT
VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	—
Conformant VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	DSC$K_CLASS_VS DSC$K_DTYPE_VT
STRING	—	—	DSC$K_CLASS_VS DSC$K_DTYPE_VT
Schema name	—	—	Internal VSI Pascal descriptor
Discriminated schema	—	—	—
SET	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	—
FILE	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	
Pointer	DSC$K_CLASS_S DSC$K_DTYPE_LU	—	—
PROCEDURE or FUNCTION	DSC$K_CLASS_S DSC$K_DTYPE_BPV	—	Bound procedure value by reference

[1]Descriptor's D_type depends on size of type.
[3]Only if PACKED ARRAY OF CHAR.
[2]Descriptor’s D_type depends on component type
[4]Component type can be a conformant VARYING OF CHAR.

Parameter Type	CLASS_NCA	CLASS_S
Ordinal	—	DSC$K_CLASS_S[1]
SINGLE	—	DSC$K_CLASS_S, DSC $K_DTYPE_F, DSC $K_DTYPE_FS
DOUBLE	—	DSC$K_CLASS_S, DSC $K_DTYPE_D, DSC $K_DTYPE_G, DSC $K_DTYPE_FT
QUADRUPLE	—	DSC$K_CLASS_S DSC $K_DTYPE_FX
RECORD	—	—
ARRAY	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC $K_DTYPE_T[3]
ARRAY OF VARYING OF CHAR	—	—
Conformant ARRAY	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC $K_DTYPE_T[3]
Conformant ARRAY OF VARYING OF CHAR[4]	—	—

Parameter Type	CLASS_NCA	CLASS_S
VARYING OF CHAR	—	—
Conformant VARYING OF CHAR	—	—
STRING	—	—
Schema name	—	—
Discriminated schema	—	—
SET	—	DSC$K_CLASS_S DSC $K_DTYPE_Z
FILE	—	DSC$K_CLASS_S DSC $K_DTYPE_Z
Pointer	—	DSC$K_CLASS_S DSC $K_DTYPE_LU
PROCEDURE or FUNCTION	—	—

5.3.3.1. CLASS_S Attribute

When the CLASS_S attribute is used on a formal parameter, the compiler generates a fixed-length scalar descriptor of a variable and passes its address to the called routine. Only ordinal, real, set, pointer, and one-dimensional packed arrays (fixed or conformant) that are of type OF CHAR can have the CLASS_S attribute on the formal parameter.

With the CLASS_S attribute, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

5.3.3.2. CLASS_A and CLASS_NCA Attributes

When you use the CLASS_A or CLASS_NCA attribute on a formal parameter, the compiler generates an array descriptor and passes its address to the called routine. The type of the CLASS_A and CLASS_NCA parameter must be an array (packed or unpacked, fixed or conformant) of an ordinal or real type.

With the CLASS_A and CLASS_NCA attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

5.3.3.3. %STDESCR Mechanism Specifier

When you use the %STDESCR mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. Only items of the following types can have the %STDESCR specifier on the actual parameter: character-string constants, string expressions, packed character arrays with lower bounds of 1, and packed conformant arrays with indexes of an integer or integer subrange type. The passing semantics depend on the variable represented by the actual parameter as follows:

- If the actual parameter is a variable of type PACKED ARRAY OF CHAR, %STDESCR implies variable semantics within the called routine.
- If the actual parameter is either a variable enclosed in parentheses, an expression, or a VARYING OF CHAR variable, %STDESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires one fixed-length string descriptor as a parameter:

```
[ASYNCHRONOUS,EXTERNAL(SYS$SETDDIR)] FUNCTION $SETDDIR
(%STDESCR New_Dir : PACKED ARRAY [$L1..$U1: INTEGER] OF CHAR;
VAR Old_Dir_Len : $UWORD := %IMMED 0;
VAR Old_Dir : [CLASS_S]PACKED ARRAY [$L2..$U2 : INTEGER] OF CHAR
:= %IMMED 0) : INTEGER; EXTERN;
.
.
.
Status := $SETDDIR('[VSI_Pascal]');
```

The actual parameter '[VSI_Pascal] ' is passed by string descriptor with foreign semantics to the formal parameter New_Dir.

5.3.3.4. %DESCR Mechanism Specifier

When you use the %DESCR mechanism specifier, the parameter generates a descriptor for an ordinal, real, or array variable and passes its address to the called routine. The type of %DESCR parameter can be any ordinal or real type, a VARYING OF CHAR string, or an array (packed or unpacked, fixed or conformant) of an ordinal or real type.

The passing semantics depend on the actual parameter's data type as follows:

- If the actual parameter is a variable, the %DESCR formal parameter implies variable semantics within the called routine.
- If the actual parameter is an expression or a variable enclosed in parentheses, %DESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires a varying-length string descriptor as its parameter:

```
TYPE
   Vary = VARYING [30] OF CHAR;

VAR
   Obj_String  : Vary;
   Times_Found : INTEGER;

[EXTERNAL] FUNCTION Search_String( %DESCR String_Val : Vary )
   : BOOLEAN; EXTERNAL;

.
.
.
IF Search_String( Obj_String )
   THEN
   Times_Found := Times_Found + 1;
```

The actual parameter Obj_String is passed by varying string descriptor with variable semantics to the formal parameter String_Val.

For More Information:

- On descriptor classes and types (*VSI OpenVMS Calling Standard*)

5.3.4. Summary of Passing Mechanisms and Passing Semantics

Table 5.4 summarizes the passing semantics used when various mechanisms are specified on either the formal or the actual parameter. For example, if a variable is passed to a formal parameter that was declared without the keyword VAR and either %REF or [REFERENCE] was specified, then variable passing semantics will be used. Likewise, if a variable is passed to a formal parameter which was declared with the keyword VAR and either %REF or [REFERENCE] was specified, then an error will occur.

If an actual parameter is preceded by an %IMMED specifier, regardless of what passing mechanism is used to declare the formal parameter, foreign semantics would be used, because a specifier appearing on the actual parameter always overrides the semantics specified on the formal parameter.

Table 5.4. Summary of Passing Mechanisms and Passing Semantics

	Actual Parameter			
	Variable		(Variable) or Expression	
Passing Mechanism	**No VAR on Formal**	**VAR on Formal**	**No VAR on Formal**	**VAR on Formal**
By immediate value				
%IMMED or [IMMEDIATE]	Foreign	Error	Foreign	Error
By reference				
Default for nonconformant or nonschema	Value	Variable	Value	Value[1]
%REF or [REFERENCE]	Variable	Error	Foreign	Error
By descriptor				
Default for conformant and schema	Value	Variable	Value	Value[1]
[CLASS_S]	Value	Variable	Value	Value[1]
[CLASS_A]	Value	Variable	Value	Value[1]
[CLASS_NCA]	Value	Variable	Value	Value[1]
%STDESCR	Variable	Error	Foreign	Error
%DESCR	Variable	Error	Foreign	Error

[1]If the formal parameter is declared with the READONLY attribute, then value-passing semantics is used; otherwise, it is an error.

5.4. Passing Parameters between VSI Pascal for OpenVMS and Other Languages

Passing parameters between VSI Pascal and other languages on VSI OpenVMS systems requires some additional knowledge about the semantics and mechanisms used by the VSI Pascal and the other compilers involved.

5.4.1. Parameter Mechanisms Versus Parameter Semantics

The Pascal language provides three parameter semantics: “VAR parameters,” “value parameters,” and “routine parameters.” These models define what happens to the parameters, not how the compiler actually implements them. “VAR parameters” are parameters that represent the actual variable passed to the routine. Changes made to the VAR parameter are reflected back to the actual variable passed in to the routine. “Value parameters” are parameters that are local copies of the expression passed into the routine. Changes made to the value parameter are not reflected back to any actual parameter passed in to the routine. “Routine parameters” are parameters that represent entire routines that may be called from inside the called routine.

The VSI Pascal compiler provides three parameter mechanisms: “by immediate value,” “by reference,” and “by descriptor.” These forms represent the actual implementation used by the compiler for the parameter. These forms are denoted by the [IMMEDIATE], [CLASS_S], [CLASS_A], and [CLASS_NCA] attributes (note, the [REFERENCE] attribute doesn't just specify a parameter mechanism, but also specifies a parameter semantic model).

VSI Pascal also provides a fourth parameter model called “foreign parameters.” These parameters become either VAR or value parameters depending on the actual parameter. If the actual parameter is a variable, then the parameter is treated as a VAR parameter. If the actual parameter is an expression, then the parameter is treated as a value parameter. These parameters are denoted by the %REF, %DESCR, and %STDESCR foreign mechanism specifiers and the [REFERENCE] attribute (identical in behavior to the %REF foreign parameter specifier).

Be careful not to confuse the term “value parameter” with the “by immediate value” mechanism. The “value” in “value parameter” describes the semantics of the parameter where changes made to the parameter inside the called routine are not reflected back to the actual parameter. It is a common misconception that VSI Pascal uses the “by immediate value” mechanism for “value parameters.”

5.4.2. Passing Nonroutine Parameters between VSI Pascal for OpenVMS and Other Languages

By default, VSI Pascal will use the “by reference” mechanism for the following VAR and value parameter types: Ordinal (integer, unsigned, char, Boolean, pointers, subranges, and enumerated types), Real (real, double, quadruple), Record, Array, Set, Varying, and File.

If you want to pass a parameter with the “by immediate value” mechanism, you can place the [IMMEDIATE] attribute on the definition of the formal parameter's definition or use the %IMMED foreign mechanism specifier on the actual parameter to override the default mechanism of the formal parameter. Only ordinal and real types may be passed with the “by immediate value” mechanism. Only value parameters may use the “by immediate value” mechanism.

If you want to accept a value parameter with the “by immediate value,” you can place the [IMMEDIATE] attribute on the definition of the formal parameter. Only ordinal and real types may be accepted with the “by immediate value” mechanism.

For example, to pass an integer with the “by immediate value” mechanisn to another routine,

```
[external] procedure rtn( p : [immediate] integer ); external;

begin
rtn(3);
```

```
rtn(some-integer-expression);
end;
```

If you want to pass a parameter with the “by descriptor” mechanism, you can place the [CLASS_A], [CLASS_S], or [CLASS_NCA] attributes on the formal parameter's definition. You can also use the %DESCR and %STDESCR foreign mechanism specifiers, but be aware that these also imply parameter semantics as well as the parameter-passing mechanism.

When passing values to a subrange parameter in a Pascal routine, the argument must be large enough to hold a value of the subrange's base-type even if the formal parameter contained a size attribute.

When passing Boolean or enumerated-type values to a VAR parameter in a Pascal routine, the calling routine must ensure that the sizes of the Boolean or enumerated-type matches the setting of the **/ENUMERATION_SIZE** qualifier or [ENUMERATION_SIZE] attribute used in the Pascal routine. For value parameters, you can pass the address of a longword as that will work for either setting.

When passing arrays or records to a Pascal routine, the calling routine must ensure that the array and record has the same layout (including any alignment holes) as chosen by the VSI Pascal compiler. You may want to use the **/SHOW=STRUCTURE_LAYOUT** listing section to help you determine the layout chosen by the VSI Pascal compiler.

By default, VSI Pascal will use the “by descriptor” mechanism for VAR and value conformant parameters. For conformant-varying parameters, VSI Pascal uses a CLASS_VS desciptor. For conformant-array parameters, VSI Pascal uses a CLASS_NCA descriptor.

Using a conformant-varying parameter or STRING schema parameter with a routine not written in Pascal is not very common since the called routine does not know how to deal with these strings. If you just are passing a string expression to the non-Pascal routine, using a conformant PACKED ARRAY OF CHAR is the right solution.

Since VSI Pascal will use either a CLASS_A or CLASS_NCA descriptor for the conformant PACKED ARRAY OF CHAR, but other languages will expect either a CLASS_S descriptor or the string "by reference", you will need to use either the [CLASS_S] attribute or the %REF foreign mechanism specifier.

For example, to pass a string expression to Fortran (which expects a CLASS_S descriptor),

```
[external] procedure fortrtn(
            p : [class_s] packed array [l..u:integer] of char); external;

  begin
  fortrtn('string');
  fortrtn(some-string-expression);
  end;
```

To pass a string expression to C (which expects a "by-reference" parameter and a null-terminated string),

```
[external] procedure crtn(
                %ref p : packed array [l..u:integer] of char); external;

  begin
  crtn('string'(0));
  crtn(some-string-expression+'0');
  end;
```

VSI Pascal has additional support to deal with null-terminated strings.

For More Information

- *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/]

When passing strings to an VSI Pascal routine from another language, you must use a descriptor if the Pascal formal parameter is a conformant parameter. VSI Pascal cannot accept a conformant parameter with the “by reference” mechanism. For conformant array parameters, you must generate a CLASS_NCA descriptor unless you select another descriptor class using the [CLASS_S], [CLASS_A], or [CLASS_NCA] attributes. For conformant varying parameters, you must generate a CLASS_VS descriptor on both platforms.

If you wish to use the "by reference" mechanism to pass strings into a Pascal routine, you must define a Pascal datatype that represents a fixed-length string (or varying-string with a maximum size) and use that datatype in the formal parameter definition.

The VSI Pascal schema type STRING is passed by CLASS_VS descriptor. Other VSI Pascal schema types use private data structures when passed between routines and cannot be accessed from routines written in other languages.

5.4.3. Passing Routine Parameters between VSI Pascal and Other Languages

On VSI OpenVMS x86-64 Systems:

By default, VSI Pascal on VSI OpenVMS x86-64 systems passes the address of a BPV (Bound Procedure Value) routine PROCEDURE and FUNCTION parameters. This BPV routine is created at run-time and is allocated from special heap storage managed by the compiler. This routine is responsible for loading the static link into register %r10 and then calling the target routine. If you use the [UNBOUND] attribute on the PROCEDURE or FUNCTION parameter or the %IMMED foreign mechanism specifier, the compiler will pass the actual address of the target routine.

On VSI OpenVMS x86-64 systems, VSI Pascal expects the address of code for routine parameters. In all VSI OpenVMS x86-64 languages, asking for the address of a routine yields the address of executable code. The OpenVMS LINKER will create trampoline routines in 32-bit address space if the actual routine is loaded in 64-bit address space.

On VSI OpenVMS I64 Systems:

By default, VSI Pascal on VSI OpenVMS I64 systems passes the address of a function descriptor for PROCEDURE or FUNCTION parameters. The presence of the [UNBOUND] attribute or the %IMMED foreign mechanism specifier has no effect over the generated code since the function descriptors in the *OpenVMS Calling Standard* allow any combination of bound and unbound routines to be passed around and invoked.

On VSI OpenVMS I64 systems, VSI Pascal expects the address of a function descriptor for routine parameters. In all VSI OpenVMS I64 languages, asking for the address of a routine yields the address of its function descriptor, since the actual address of the instructions is not useful by itself.

On VSI OpenVMS Alpha Systems:

By default, VSI Pascal on VSI OpenVMS Alpha systems passes the address of a procedure descriptor for PROCEDURE or FUNCTION parameters. The presence of the [UNBOUND] attribute or the

%IMMED foreign mechanism specifier has no effect over the generated code since the procedure descriptors in the *OpenVMS Calling Standard* allow any combination of bound and unbound routines to be passed around and invoked.

On VSI OpenVMS Alpha systems, VSI Pascal expects the address of a procedure descriptor for routine parameters. In all VSI OpenVMS Alpha languages, asking for the address of a routine yields the address of its procedure descriptor, since the actual address of the instructions is not useful by itself.

Chapter 6. Programming on VSI OpenVMS Systems

To eliminate duplication of programming and debugging efforts, VSI OpenVMS systems provide many routines to perform common programming tasks. These routines are collectively known as system routines. They include routines in the run-time library to assist you in such areas as mathematics, screen management, and string manipulation. Also included are *VSI OpenVMS Record Management Services* (RMS), which are used to access files and their records. There are also system services that perform tasks such as resource allocation, information sharing, and input/output coordination.

Because all VSI OpenVMS system routines adhere to the *VSI OpenVMS Calling Standard*, you can declare any system routine as an external routine and then call the routine from an VSI Pascal program.

6.1. Using System Definitions Files

To access system entry points, data structures, symbol definitions, and messages, VSI Pascal provides files that you can inherit (.PEN) or include (.PAS) in your application. Table 6.1 summarizes the source and environment files that VSI Pascal makes available to you in the directory SYS$LIBRARY (for instance, SYS$LIBRARY:STARLET.PEN).

Table 6.1. VSI Pascal for OpenVMS Definitions Files

File	Description
System Services Definitions File:	
STARLET.PAS/.PEN	Contains VSI OpenVMS system service definitions, LIB$ messages, MTH$ messages, OTS$ messages, SMG$ data structures and termtable, STR$ messages, RMS routine declarations, system symbolic names, and RMS data structures.
Run-Time Library Definitions Files:	
PASCAL$ACLEDIT_ROUTINES.PAS/.PEN	Contains ACLEDIT$ routine entry points.
PASCAL$C_ROUTINES.PAS/.PEN	Contains C$ routine entry points.
PASCAL$CLI_ROUTINES.PAS/.PEN	Contains CLI$ routine entry points.
PASCAL$CMA_ROUTINES.PAS/.PEN	Contains routine entry points, data structures, and messages for DECthreads. For more information on DECthreads, see the *Guide to POSIX Threads Library*.
PASCAL$CONV_ROUTINES.PAS/.PEN	Contains CONV$ routine entry points.
PASCAL$CVT_ROUTINES.PAS/.PEN	Contains CVT$ routine entry points.
PASCAL$DCX_ROUTINES.PAS/.PEN	Contains DCX$ routine entry points.
PASCAL$DTK_ROUTINES.PAS/.PEN	Contains DTK$ routine entry points, data structures, and messages.
PASCAL$EDT_ROUTINES.PAS/.PEN	Contains EDT$ routine entry points.
PASCAL$FDL_ROUTINES.PAS/.PEN	Contains FDL$ routine entry points.
PASCAL$LBR_ROUTINES.PAS/.PEN	Contains LBR$ routine entry points.
PASCAL$LIB_ROUTINES.PAS/.PEN	Contains LIB$ routine entry points.

File	Description
PASCAL$MAIL_ROUTINES.PAS/.PEN	Contains MAIL$ routine entry points.
PASCAL$MTH_ROUTINES.PAS/.PEN	Contains MTH$ routine entry points.
PASCAL$NCS_ROUTINES.PAS/.PEN	Contains NCS$ routine entry points.
PASCAL$OTS_ROUTINES.PAS/.PEN	Contains OTS$ routine entry points.
PASCAL$PPL_ROUTINES.PAS/.PEN	Contains PPL$ routine entry points, data structures, and messages.
PASCAL$PSM_ROUTINES.PAS/.PEN	Contains PSM$ routine entry points.
PASCAL$SMB_ROUTINES.PAS/.PEN	Contains SMB$ routine entry points.
PASCAL$SMG_ROUTINES.PAS/.PEN	Contain SMG$ routine entry points and messages.
PASCAL$SOR_ROUTINES.PAS/.PEN	Contains SOR$ routine entry points and messages.
PASCAL$STR_ROUTINES.PAS/.PEN	Contains STR$ routine entry points.
PASCAL$TPU_ROUTINES.PAS/.PEN	Contains TPU$ routine entry points.
Symbol Definitions Files:[1]	
LIBDEF.PAS	Contains definitions for all condition symbols from the general utility run-time library routines.
MTHDEF.PAS	Contains definitions for all condition symbols from the mathematical routines library.
SIGDEF.PAS	Contains miscellaneous symbol definitions used in condition handlers. These definitions are also included in STARLET.PEN.
VSI Pascal for OpenVMS Run-Time Library Files:	
PASDEF.PAS	Contains definitions for all condition symbols from the VSI Pascal run-time library routines.
PASSTATUS.PAS	Contains definitions for all values returned by the STATUS and STATUSV routines.

[1]These files are retained for compatibility with older versions of this product and do not contain symbol definitions for subsequent releases of the product. (For definitions that are complete for the latest release of VSI OpenVMS, use the individual PASCAL$ files or STARLET). To access these files, use the %INCLUDE directive in the CONST declaration section of your program.

For instance, the external routine declarations in STARLET define new identifiers by which you can refer to the routines. Example 6.1 shows that you can refer to SYS$HIBER as $HIBER if you use STARLET.

Example 6.1. Inheriting STARLET.PEN to Call SYS$HIBER

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Suspend (INPUT,OUTPUT);
TYPE
   Sys_Time = RECORD
      I,J : INTEGER;
      END;
   Unsigned_Word = [WORD] 0..65535;
VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
   Length       : Unsigned_Word;
   Job_Name     : VARYING[15] OF CHAR;
   Ascii_Time   : VARYING[80] OF CHAR;
   Binary_Time  : Sys_Time;
```

```
BEGIN
{ Print current date and time }
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR(Current_Time, 1, Length);

{ Get name of process to suspend }
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);

{ Get time to wake process }
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);

{ Convert time to binary }
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
THEN
   BEGIN
      WRITELN ('Illegal format for time string');
      HALT;
   END;

{ Suspend process }
IF NOT ODD ($SUSPND (PRCNAM := Job_Name))
THEN
   BEGIN
      WRITELN ('Cannot suspend process');
      HALT;
   END;

{ Schedule wakeup request for self }
IF ODD ($SCHDWK (DAYTIME := Binary_Time))
THEN
   $HIBER { Put self to sleep }
ELSE
   BEGIN
      WRITELN ('Cannot schedule wakeup');
      WRITELN ('Process will resume immediately');
   END;

{ Resume process }
IF NOT ODD ($RESUME (PRCNAM := Job_Name))
THEN
   BEGIN
      WRITELN ('Cannot resume process');
      HALT;
   END;
END.
```

6.2. Declaring System Routines

Before calling a routine, you must declare it. System routine names conform to one of the two following conventions:

```
[[facility-code]]$procedure-name
```

For example, LIB$PUT_OUTPUT is the run-time library routine used to write a record to the current output device and $ASCTIM is a system service routine used to convert binary time to ASCII time.

Because system routines are often called from condition handlers or asynchronous trap (AST) routines, you should declare system routines with the ASYNCHRONOUS attribute.

Each system routine is documented with a structured format in the appropriate VSI OpenVMS reference manual. The documentation for each routine describes the routine's purpose, the declaration format, the return value, and any required or optional parameters. Detailed information about each parameter is listed in the description. The following format is used to describe each parameter:

```
parameter-name
OpenVMS Usage :     OpenVMS data type
type          :     parameter data type
access        :     parameter access
mechanism     :     parameter-passing mechanism
```

Using this information, you must determine the parameter's data type (type), the parameter's passing semantics (access), the mechanism used to pass the parameter (mechanism), and whether the parameter is required or optional from the call format.

The following sections describe the methods available in VSI Pascal to obtain the various data types, access methods, and passing mechanisms.

6.2.1. Methods Used to Obtain VSI OpenVMS Data Types

The data specified by a parameter has a data type. Several VSI OpenVMS standard data types exist. A system routine parameter must use one of these data types.

For More Information:

- On VSI OpenVMS data types and equivalent VSI Pascal declarations (*VSI OpenVMS Calling Standard*)

6.2.2. Methods Used to Obtain Access Methods

The access method describes the way in which the called routine accesses the data specified by the parameter. The following three methods of access are the most common:

- Read only—data must be read by the called routine. The called routine does not write the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

- Write only—data that the called routine returns to the calling routine must be written into a location where the calling routine can access it. Such data can be thought of as output data. The called routine does not read the contents either before or after it writes into the location.

- Modify—a parameter specifies data that is both read and written by the called routine. In this case, the called routine reads the input data, which it uses in its operations, and then overwrites the input data with the results. Thus, when the called routine completes execution, the input data specified by the argument is lost.

Table 6.2 lists all access methods that may appear under the access entry in a parameter description, as well as the VSI Pascal translation.

Table 6.2. Access Type Translations

Access Entry	Method Used in VSI Pascal for OpenVMS
Call after stack unwind	Procedure or function parameter passed by immediate value
Function call (before return)	Function parameter
Jump after unwind	Not available
Modify	Variable semantics[1]
Read only	Value or foreign semantics
Call without stack unwind	Procedure parameter
Write only	Variable semantics[1]

[1]It is possible to obtain variable semantics by either specifying the VAR keyword on the formal parameter or by passing a variable as an actual parameter using %REF, %DESCR, or %STDESCR.

6.2.3. Methods Used to Obtain Passing Mechanisms

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the parameter-passing mechanism.

Table 6.3 lists all passing mechanisms that may appear under the mechanism entry in an argument description and the method used in VSI Pascal to obtain the passing mechanism.

Table 6.3. Mechanism Type Translations

Mechanism Entry	Method Used in VSI Pascal for OpenVMS
By value	%IMMED or [IMMEDIATE]
By reference	VAR, %REF or [REFERENCE] or default
By descriptor	
Fixed-length	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]
Dynamic-string	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]
Array	[CLASS_A]
Procedure	N.A.
Decimal-string	N.A.
Noncontiguous-array	Array type, conformant array parameter, %DESCR, or [CLASS_NCA]
Varying-string	Value, VAR, or %DESCR conformant parameter of type VARYING OF CHAR, or %DESCR parameter of type VARYING OF CHAR
Varying-string-array	Value, VAR, or %DESCR conformant parameter of type array of VARYING OF CHAR, or %DESCR parameter of type array of VARYING OF CHAR
Unaligned-bit-string	N.A.
Unaligned-bit-array	N.A.
String-with-bounds	N.A.

Mechanism Entry	Method Used in VSI Pascal for OpenVMS
Unaligned-bit-string-with-bounds	N.A.

Parameters passed by reference and used solely as input to a system service should be declared with VSI Pascal value semantics; this allows actual parameters to be compile-time and run-time expressions. When a system service requires a formal parameter with a mechanism specifier, you should declare the formal parameter with the READONLY attribute to specify value semantics. Other parameters passed by reference should be declared with VSI Pascal variable semantics to ensure that the output data is interpreted correctly. In some cases, by reference parameters are used for both input and output and should also be declared with variable semantics.

The following example shows the declaration of the Convert ASCII String to Binary Time (SYS$BINTIM) system service and a corresponding function designator:

```
TYPE
   $QUAD = [QUAD,UNSAFE] RECORD
              L0 : UNSIGNED;
              L1 : INTEGER;
           END;

VAR
   Ascii_Time  : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;

[ASYNCHRONOUS,EXTERNAL(SYS$BINTIM)] FUNCTION $BINTIM
   (TIMBUF : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER; EXTERNAL;
{In the executable section:}
IF NOT ODD ($BINTIM(Ascii_Time, Binary_Time))
THEN
   BEGIN
      WRITELN ('Illegal format for time string');
      HALT;
   END;
```

The first formal parameter requires the address of a character-string descriptor with value semantics; the second requires an address and uses variable semantics to manipulate the parameter within the service. Because you can call $BINTIM from a condition handler or AST routine, you should declare it with the ASYNCHRONOUS attribute. Also, because you may want to pass a volatile variable to the TIMADR parameter, you should use the VOLATILE attribute to indicate that the argument is allowed to be volatile.

6.2.4. Data Structure Parameters

Some system services require a parameter to be the address of a data structure that indicates a function to be performed or that holds information to be returned. Such a structure can be described as a list, a control block, or a vector. The size and POS attributes provide an efficient method of laying out these data structures. The size attributes ensure that the fields of the data structure are of the size required by the system service, and the POS attribute allows you to position the fields correctly.

For example, the Get Job/Process Information (SYS$GETJPIW) system service requires an item list consisting of an array of records of 12 bytes, where all but the last array cell requests one piece of data and the last array cell represents the item list terminator. By packing the record, you can guarantee

that the fields of each record are allocated contiguously. Example 6.2 uses the system service routine $GETJPIW to retrieve the process's name as a 12-byte string.

Example 6.2. Using $GETJPIW to Retrieve a Process Name

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Userid( OUTPUT );

TYPE
   Uword       = [WORD] 0..65535;
   Itmlst_Cell = PACKED RECORD
                    CASE INTEGER OF
                       1 : (Buf_Len   : Uword;
                            Item_Code : Uword;
                            Buf_Addr  : INTEGER;
                            Len_Addr  : INTEGER);
                       2 : (Term      : INTEGER);
                 END;
VAR
   Username_String   : [VOLATILE] VARYING [12] OF CHAR;
   Itmlst            : ARRAY [1..2] OF Itmlst_Cell := ZERO;
BEGIN
Itmlst[1].Buf_Len := 12;                        { 12 bytes returned }
Itmlst[1].Item_Code := JPI$_USERNAME;           { return user name }
Itmlst[1].Buf_Addr :=                           { store returned name here }
   IADDRESS(Username_String.BODY);
Itmlst[1].Len_Addr :=                           { store returned length here }
   IADDRESS(Username_String.LENGTH);
Itmlst[2].Term := 0;                            { terminate item list }

IF NOT ODD( $GETJPIW(,,,Itmlst) )
THEN
   WRITELN('error')
ELSE
   WRITELN('user name is ',Username_String);
END.
```

For More Information:

- On size attributes (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

6.2.5. Default Parameters

In some cases, you do not have to supply actual parameters to correspond to all the formal parameters of a system service. In VSI Pascal, you can supply default values for such optional parameters when you declare the service. You can then omit the corresponding actual parameters from the routine call. If you choose not to supply an optional parameter, you should initialize the formal parameter with the appropriate value using the by immediate value (%IMMED) mechanism. The correct default value is usually 0.

For example, the Cancel Timer (SYS$CANTIM) system service has two optional parameters. If you do not specify values for them in the actual parameter list, you must initialize them with zeros when they are declared. The following example is the routine declaration for SYS$CANTIM:

```
[ASYNCHRONOUS,EXTERNAL(SYS$CANTIM)] FUNCTION $CANTIM (
        %IMMED REQIDT : UNSIGNED := %IMMED 0;
        %IMMED ACMODE : UNSIGNED := %IMMED 0) : INTEGER; EXTERNAL;
```

A call to $CANTIM must indicate the position of omitted parameters with a comma, unless they all occur at the end of the parameter list. For example, the following are legal calls to $CANTIM using the previous external declaration:

```
$CANTIM (, PSL$C_USER);
$CANTIM (I);
$CANTIM;
```

PSL$C_USER is a symbolic constant that represents the value of a user access mode, and I is an integer that identifies the timer request being canceled. If you call $CANTIM with both of its default parameters, you can omit the actual parameter list completely.

When it is possible for the parameter list to be truncated, you can also specify the TRUNCATE attribute on the formal parameter declaration of the optional parameter. The TRUNCATE attribute indicates that an actual parameter list for a routine can be truncated at the point that the attribute was specified. However, once one optional parameter is omitted in the actual parameter list, it is not possible to specify any optional parameter following that. For example:

```
[ASYNCHRONOUS] FUNCTION LIB$GET_FOREIGN (
   VAR Resultant_String : [CLASS_S,VOLATILE]
           PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
       Prompt_String    : [CLASS_S,TRUNCATE]
           PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR := %IMMED 0;
   VAR Resultant_Length : [VOLATILE,TRUNCATE] $UWORD := %IMMED 0;
   VAR Flags            : [VOLATILE,TRUNCATE] UNSIGNED := %IMMED 0)
           : INTEGER; EXTERNAL;
```

With this declaration, it is possible to specify values for Resultant_String and Prompt_String and truncate the call list at that point. In this case, two parameters would be passed in the CALL instruction.

You may want to use a combination of the %IMMED 0 and TRUNCATE methods. This combination would allow you to skip the specification of intermediate optional parameters, as well as allow you to truncate the call list once all desired parameters have been specified.

Note that VSI OpenVMS system services require a value to be passed by parameters, including optional parameters; therefore, you should not use the TRUNCATE attribute when defining optional parameters to a system service. Instead, you should specify default values on the formal parameter declaration.

The TRUNCATE attribute is useful when calling routines for which the optional parameter is truly optional, for example, when calling run-time library routines.

For More Information:

- On the TRUNCATE attribute (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

6.2.6. Arbitrary Length Parameter Lists

Some run-time library routines require a variable number of parameters. For example, there is no fixed limit on the number of values that can be passed to functions that return the minimum or maximum value from a list of input parameters. The LIST attribute supplied by VSI Pascal allows you to indicate the mechanism by which excess actual parameters are to be passed. For example:

```
[ASYNCHRONOUS] FUNCTION MTH$DMIN1 (
   D_FLOATING   : DOUBLE;
   EXTRA_PARAMS : [LIST] DOUBLE) : DOUBLE; EXTERNAL;
```

Because the function MTH$DMIN1 returns the D_floating minimum of an arbitrary number of D_floating parameters, the formal parameter EXTRA_PARAMS is declared with the LIST attribute. All actual parameters must be double-precision real numbers passed by reference with value semantics.

For More Information:

On the LIST attribute (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

6.3. Calling System Routines

All system routines are functions that return an integer condition value; this value indicates whether the function executed successfully. An odd-numbered condition value indicates successful completion; an even-numbered condition value indicates a warning message or failure. Your program can use the VSI Pascal predeclared function ODD to test the function return value for success or failure. For example:

```
IF NOT ODD ($BINTIM(Ascii_Time,Binary_Time))
THEN
   BEGIN
      WRITELN('Illegal format for time string');
      HALT;
   END;
```

In addition, run-time library routines return one or two values: the result of a computation or the routine's completion status, or both. When the routine returns a completion status, you should verify the return status before checking the result of a computation. You can use the function ODD to test for success or failure or you can check for a particular return status by comparing the return status to one of the status codes defined by the system. For example:

```
VAR
   Seed_Value  : INTEGER;
   Rand_Result : REAL;

[ASYNCHRONOUS] FUNCTION MTH$RANDOM (
   VAR seed : [VOLATILE] UNSIGNED) : SINGLE; EXTERNAL;
{In the executable section:}
Rand_Result := MTH$RANDOM (Seed_Value);
```

When the routine's completion status is irrelevant, your program can treat the function as though it were an external procedure and ignore the return value. For example, your program can declare the Hibernate (SYS$HIBER) system service as a function but call it as though it were a procedure:

```
[ASYNCHRONOUS,EXTERNAL(SYS$HIBER)] FUNCTION $HIBER
   : INTEGER; EXTERNAL;
{In the executable section:}
$HIBER; { Put process to sleep }
```

Because SYS$HIBER is expected to execute successfully, the program will ignore the integer condition value that is returned.

6.4. Using Attributes

When writing programs that use VSI OpenVMS System Services and run-time library routines, it is common to use several VSI Pascal attributes.

The VOLATILE attribute indicates that a variable is written or read indirectly without explicit program action. The most common occurrence for this is with item lists. In that case, the address of the variable is placed in the item list (most likely using the IADDRESS routine). This address is then used later when the entire item list is passed to a system service. Without the VOLATILE attribute, the compiler does not realize that the call to the system service or run-time library routine uses the variable.

The UNBOUND attribute designates a routine that does not have a static link available to it. Without a static link, a routine can only access local variables, parameters, or statically allocated variables. System services that require AST or action routines want the address of an UNBOUND routine. Routines at the outer level of a PROGRAM or MODULE are UNBOUND by default.

The ASYNCHRONOUS attribute designates a routine that might be called asynchronously of any program action. This allows the compiler to verify that the asynchronous routine only accesses local variables, parameters, and VOLATILE variables declared at outer levels. Without the assurance that only VOLATILE variables are used, the asynchronous routine might access incorrect data, or data written by the routine will not be available to the main program.

6.5. Using Item Lists

Many VSI OpenVMS system services use item lists. **Item lists** are sequences of control structures that provide input to the system service and that describe where the service should place its output. These item lists can have an arbitrary number of cells and are terminated with a longword of value 0.

Since different programs need a different number of item list cells, you can use a schema type to define a generic item list data type. This schema type can then be discriminated with the appropriate number of cells. Consider the following example:

```
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1: (    { Normal Cell }
              Buffer_Length : [WORD] 0..65535;
              Item_Code     : [WORD] 0..65535;
              Buffer_Addr   : UNSIGNED;
              Return_Addr   : UNSIGNED
            );
         2: (    { Terminator }
              Terminator    : UNSIGNED
            );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY [1..Count] OF Item_List_Cell;
```

The Item_List_Cell data type specifies what a single cell looks like. The Buffer_Addr and Return_Addr fields are declared as UNSIGNED since most applications use the IADDRESS predeclared routine to fill them in. The Item_List_Template schema type defines an array of item list cells with a upper bound to be filled in by an actual discriminant.

To use this schema type, first determine the number of item list cells required including one cell for the terminator. After the number of cells has been determined, declare a variable discriminating the schema. Consider the following example:

```
VAR
   Item_List : Item_List_Template( 2 );
```

Additionally, since actual discriminants to schema can be run-time expressions, you can write a routine that can have item lists with a number of cells that is determined at run time.

After the item list variable has been declared, each cell must be filled in according to the system service and operation requested.

Consider the following example using the SYS$TRNLNM system service:

```
VAR
   Item_List       : Item_List_Template( 2 );
   Translated_Name : [VOLATILE] VARYING [132] OF CHAR;

   {Specify the buffer to return the translation:}
   Item_List[1].Buffer_Length   := SIZE( Translated_Name.BODY );
   Item_List[1].Item_Code       := LNM$_String;
   Item_List[1].Buffer_Addr     := IADDRESS( Translated_Name.BODY );
   Item_List[1].Return_Addr     := IADDRESS( Translated_Name.LENGTH);

   { Terminate the item list:}
   Item_List[2].Terminator      := 0;
```

The VAR section declares an item list with two cells. It also declares an output buffer for the system service. The VOLATILE attribute is used since the call to SYS$TRNLNM indirectly writes into the variable. The first cell is filled in with the operation desired, the size of the output buffer, the location to write the result, and the location to write the size of the result.

Using the SIZE predeclared function prevents the code from having to be modified if the output buffer ever changes size. Using the BODY and LENGTH predeclared fields of the VARYING string allows the system service to construct a valid VARYING OF CHAR string. Finally, the second cell of the item list is initialized. Since the second cell is the last cell, the terminator field must be filled in with a value of 0.

6.6. Using Foreign Mechanism Specifiers on Actual Parameters

The definition files provided by VSI Pascal (SYS$LIBRARY:STARLET.PAS and so forth) are created from a generic description language used by the VSI OpenVMS operating system. Since this description language does not contain all the features found in VSI Pascal, some of the translations do not take advantage of VSI Pascal features. Also, since several of the system services are generic in nature, it is impossible to provide a definitive definition for every situation.

If a formal parameter definition does not reflect the current usage, you can use a foreign mechanism specifier to direct the compiler to use a different passing mechanism or different descriptor type than the default for that parameter.

Consider the following:

- ASTADR parameter

  Many system services define this parameter to be a procedure parameter with no formal parameters. This is because the format of the arguments passed to the AST routine vary with the system service. If you specify a routine with parameters as the actual parameter to an ASTADR parameter, you will receive a compile-time error saying that the formal parameter and actual parameter have different parameter lists. To solve this problem, you can specify the %IMMED foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the routine without verifying that the parameter lists are identical.

- ASTPRM parameter

  Many system services define this parameter to be an UNSIGNED parameter passed by immediate value. Since the parameter to an AST routine is dependent on the application, it is often desired to pass the address of a variable instead of its contents. To solve this problem, you can specify the %REF foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the variable instead of the contents of the variable.

- P1..Pn parameters

  The P1 through P6 parameters of the $QIO and $QIOW system services and the P1 through P20 parameters of the $FAO system services are also defined to be UNSIGNED parameters passed by immediate value. If the actual parameter is not UNSIGNED or requires a different passing mechanism, you can specify the %REF foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the variable instead of the contents of the variable.

- RESULTANT_FILESPEC parameter of the LIB$FIND_FILE run-time library routine

  This parameter is declared to be a VAR conformant PACKED ARRAY OF CHAR parameter and is passed by a CLASS_A descriptor. However, the LIB$FIND_FILE routine can also accept CLASS_VS descriptors of VARYING OF CHAR variables. To cause the compiler to build a CLASS_VS descriptor instead of the default CLASS_A descriptor, you can specify the %DESCR foreign mechanism specifier on the actual VARYING OF CHAR parameter.

6.7. Using 64-Bit Pointer Types

VSI Pascal includes limited support for 64-bit pointers.

64-bit pointers can be declared by using the [QUAD] attribute on a pointer variable declaration. When [QUAD] is used, the generated code will use all 64 bits of the pointer.

6.7.1. Pascal Language Features Not Supported with 64-Bit Pointers

Several Pascal features are not supported with 64-bit pointers. These features are:

- Base types of 64-bit pointers cannot contain file types or schema types.
- The READ built-in routine cannot read into variables accessed with 64-bit pointers. For example, the following code fragment will be rejected by the compiler:

  ```
  var quad_ptr : [quad] ^integer;

  begin
  quad_ptr := my_alloc_routine(size(integer));
  read(quad_ptr^);
  end
  ```

- Strings allocated in P2 address space cannot be used with the READV or WRITEV predeclared routines.
- VSI Pascal understands 32-bit descriptors as defined by the VSI OpenVMS calling standard. Therefore, any VSI Pascal construct that relies on descriptors is not supported for variables accessed with 64-bit pointers. The features rejected for 64-bit pointers are:

- The use of %DESCR or %STDESCR on actual parameter values accessed with 64-bit pointers. For example, you cannot do the following:

```
type
  s32 = packed array [1..32] of char;
var
  qp : [quad] ^s;

begin
  qp := my_alloc_routine(size(s32));
  some_routine( %stdescr qp^ );
end;
```

- Passing variables accessed with 64-bit pointers to formal parameters declared with %DESCR or %STDESCR foreign mechanism specifiers.
- Passing variables accessed with 64-bit pointers to conformant array or conformant varying parameters.
- Passing variables accessed with 64-bit pointers to STRING parameters.
- The **/USAGE=64BIT_TO_DESCR** command line option can be used to disable these checks and the compiler will build a 32-bit descriptor containing the bottom 32-bits of the 64-bit address expression.
- At run time, the compiler will generate incorrect code when passing a VAR parameter that is accessed with a 64-bit pointer to a parameter that requires a descriptor. The generated code will build the descriptor with the lower 32-bits of the 64-bit address. For example:

```
type
    s32 = packed array [1..32] of char;
var
    qp : [quad] ^s;

procedure a( p : packed array [l..u:integer] of char );
  begin
  writeln(a);
  end;

procedure b( var p : s32 );
  begin
  a(p);   { This will generate a bad descriptor }
  end;

begin
qp := my_alloc_routine(size(s32));
b(qp^);
end;
```

6.7.2. Using 64-Bit Pointers with System Definition Files

For routines that have parameters that are 64-bit pointers, the Pascal definition uses a 64-bit record type. The definition files do not support either the INTEGER64 datatype or 64-bit pointers.

You can override the formal definition inside of definition files by using a foreign mechanism specifier (that is, %IMMED, %REF, %STDESCR, and %DESCR) on an actual parameter.

For example, the following is an example of calling lib$get_vm_64 using %ref to override the definition from PASCAL$LIB_ROUTINES.PEN:

```
[inherit('sys$library:pascal$lib_routines')]
program p64(input,output);

const
     arr_size = (8192 * 10) div 4; ! Make each array be 10 pages

type
     arr = array [1..arr_size] of integer;
     arrptr = [quad] ^arr;

var
     ptr : arrptr;
     ptrarr : array [1..10] of arrptr;
     i,j,stat : integer;
     sum : integer64;

!  PASCAL$LIB_ROUTINES.PAS on a V7.1 system contains
!  the following definitions for LIB$GET_VM_64
!
!type
!      $QUAD = [QUAD,UNSAFE] RECORD
!                 L0:UNSIGNED; L1:INTEGER; END;
!      $UQUAD = [QUAD,UNSAFE] RECORD
!                 L0,L1:UNSIGNED; END;
!      lib$routines$$typ4 = ^$QUAD;
!
![ASYNCHRONOUS] FUNCTION lib$get_vm_64 (
!         number_of_bytes : $QUAD;
!         VAR base_address : [VOLATILE] lib$routines$$typ4;
!         zone_id : $UQUAD := %IMMED 0) : INTEGER; EXTERNAL;
!
! Note that the BASE_ADDRESS parameter is a 64-bit pointer
! that will be returned by LIB$GET_VM_64.  The definition
! incorrectly declared it as a pointer to a record that is
! quadword sized.
!

begin

! Allocate memory with lib$get_vm_64.  The definition of
! lib$get_vm_64 declares the return address parameter as
! a quadword-sized record since it doesn't have sufficient
! information to generate a INTEGER64 or other type.
!
! Use an explicit '%ref' foreign mechanism specifier to
! override the formal parameter's type definition and pass
! our pointer to lib$get_vm_64.
!

writeln('arr_size = ',arr_size:1);
for i := 1 to 10 do
  begin
  stat := lib$get_vm_64( size(arr), %ref ptrarr[i] );
  if not odd(stat)
  then
```

```
      begin
      writeln('Error from lib$get_vm_64: ',hex(stat));
      lib$signal(stat);
      end;
  writeln('ptrarr[',i:1,'] = ',hex(ptrarr[i]));
  end;

! Read/write all the memory locations to get some page faults
!
writeln('Initialize all memory');
for i := 1 to 10 do
  for j := 1 to arr_size do
    ptrarr[i]^[j] := i + j;

sum := 0;
writeln('Add up all memory in reverse direction');
for i := 10 downto 1 do
  for j := arr_size downto 1 do
    sum := sum + ptrarr[i]^[j];
writeln('Sum of array contents = ',sum:1);

end.
```

The compiler allows the LONG and QUAD attributes to be specified on pointer types, as shown in the following example:

```
 var long_ptr : ^integer;
      quad_ptr : [quad] ^integer;
```

Both pointers point to integers, but `long_ptr` is 32 bits while `quad_ptr` is 64 bits.

Chapter 7. Input and Output Processing

This chapter provides details on the input/output (I/O) support provided for VSI OpenVMS systems.

7.1. Environment I/O Support

VSI Pascal uses the Record Management Services (RMS) to perform I/O tasks at the system level. In this environment, all of the VSI Pascal I/O model is supported; the model is based on RMS concepts. If these sections contain no information on a concept or element in the VSI Pascal I/O model, then this environment supports the concept or element exactly as it is described in the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/].

You can use RMS features through VSI Pascal when you call the OPEN procedure. For instance, when you call this procedure, you can specify the file organization, the component format, and the access method.

If you choose to use additional features of RMS that are not available in the VSI Pascal I/O model, you can write a user-action function that manipulates the RMS control blocks: the file access block (FAB), the record access block (RAB), and the extended attribute block (XAB). Once you write the user-action function, you pass the function name as a parameter to the OPEN procedure.

For More Information:

- On user-action functions (Section 7.2)
- On OPEN defaults (Section 7.1.6.1)
- On OPEN and the VSI Pascal I/O model (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])
- On RMS concepts (*VSI OpenVMS Guide to OpenVMS File Applications*)
- On the user interface to RMS (*VSI OpenVMS Record Management Services Reference Manual*)

7.1.1. Indexed Files

The VSI Pascal I/O model allows you to use most of the features of RMS indexed files. However, if you wish to use segmented or null keys, you must write a user-action function.

When an existing indexed file is opened, the run-time library compares the keys in the file against the KEY attributes specified in the program. If no KEY attribute was specified for the corresponding key in the indexed file, then the comparison is bypassed and the open continues. The run-time library compares the position and the data type of the file's keys against the KEY attributes specified. If the KEY attribute explicitly specifies a collating sequence (ASCENDING or DESCENDING), then the specified sequence must match that of the key in the file. If no sequence is specified, either sequence is allowed. The CHANGES and DUPLICATES options are not checked.

For More Information:

- On user-action functions (Section 7.2)

- On the OPEN procedure (Section 7.1.6)
- On indexed file organization and the KEY attribute (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

7.1.2. VSI OpenVMS Components and RMS Records

In the VSI Pascal I/O model, data items in a file are called components. In RMS, these items are called records.

7.1.3. Count Fields for Variable-Length Components

Each variable-length component contains a count field as a prefix. This count field contains the number of bytes in the rest of the component. For files on tape, this count field is 4 bytes in length; for files on disk, this count field is 2 bytes in length.

7.1.4. Variable-Length with Fixed-Length Control Field (VFC) Component Format

The VSI Pascal I/O model does not provide a direct means to create files of variable-length components with fixed-length control fields (VFC). If you open a file of this component format, VSI Pascal treats the file like a file of variable-length components. If you want to create files of this component format, you must write a user-action function.

For More Information:

- On user-action functions (Section 7.2)
- On VFC components (*VSI OpenVMS Guide to OpenVMS File Applications*)

7.1.5. Random Access by Record File Address (RFA)

The VSI Pascal I/O model does not allow random access by record file address. If you want to use this type of access, you must write a user-action function.

RMS supports random access by Record File Address (RFA) for relative and indexed files, and for sequential files only on disk. The RFA is a unique number supplied for files on disk. The RFA remains constant as long as the record is in the file. RMS makes the RFA available to your program every time the record is stored or retrieved. Your program can either ignore the RFA or it can keep it as a random-access pointer to the record for subsequent accesses.

If your disk file is sequential with variable-length records, the RFA provides the only method for randomly accessing records of that file.

For More Information:

- On user-action functions (Section 7.2)
- On RFA (*VSI OpenVMS Guide to OpenVMS File Applications*)

7.1.6. OPEN Procedure

When you use the OPEN procedure, RMS applies default values for OpenVMS file specifications, and assigns values to FAB, RAB, XAB, and Name Block data structures.

7.1.6.1. OPEN Defaults

When you use OPEN to open a file, RMS applies certain defaults when attempting to locate the physical file. Table 7.1 presents these defaults.

Table 7.1. Default Values for VSI OpenVMS File Specifications

Element	Default
Node	Local computer
Device	Current user device
Directory	Current user directory
File name	VSI Pascal file variable name or its logical name translation
File type	.DAT
Version number (history)	OLD: highest current number NEW: highest current number + 1

The OPEN procedure includes a default file-name parameter. Using this parameter, you can access the RMS default file-name parameter to set file-specification defaults. Consider the following example:

```
VAR
    My_File : VARYING [20] OF CHAR;
    My_File_Var : TEXT;
BEGIN
My_File := 'foo.bar';
OPEN( FILE_NAME := My_File,
      FILE_VARIABLE := My_File_Var,
      DEFAULT := '[another.dir]' );
```

The OPEN statement in the previous example opens the file called [ANOTHER.DIR]FOO.BAR. RMS applies the defaults in Table 7.1 to determine the node, the device, and the version number of the file.

For More Information:

- On file specifications (*VSI OpenVMS User's Manual*)
- On the OPEN procedure (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

7.1.6.2. OPEN and RMS Data Structures

Table 7.2 presents the status of RMS FAB fields when you call the OPEN procedure. If a field is not included in the following tables, it is initialized to zero.

Table 7.2. Setting of RMS File Access Block Fields by a Call to the OPEN Procedure

Field	Name	OPEN Parameters and Value
FAB$L_CTX	Context	Reserved to VSI.
FAB$L_DEV	Device characteristics	Returned by RMS.
FAB$L_DNA	Default file specification string address	DEFAULT parameter value, if specified; else, .DAT.
FAB$L_DNS	Default file specification string size	Set to length of default file name string.
FAB$B_FAC	File access options	

Field	Name	OPEN Parameters and Value
FAB$V_DEL	Allow deletions	1, if not HISTORY:=READONLY.
FAB$V_GET	Allow reads	1
FAB$V_PUT	Allow writes	1, if not HISTORY:=READONLY.
FAB$V_TRN	Allow truncations	1, if not HISTORY:=READONLY.
FAB$V_UPD	Allow updates	1, if not HISTORY:=READONLY.
FAB$L_FNA	File specification string address	FILE_NAME if specified, name of file variable if external file, else 0.
FAB$B_FNS	File specification string size	Set to length of file name string.
FAB$L_FOP	File processing options	
FAB$V_CIF	Create if nonexistent	1, if HISTORY := UNKNOWN.
FAB$V_DFW	Deferred write	1
FAB$V_DLT	Delete on close service	Set when file is closed, depends on DISPOSITION.
FAB$V_NAM	Name block inputs	1, if terminal file reopened to enable prompting.
FAB$V_SFC	Submit command file (when closed)	Set when file is closed, depends on DISPOSITION.
FAB$V_SQO	Sequential only	1, if ACCESS_METHOD:=SEQUENTIAL (default).
FAB$V_TEF	Truncate at end of file	Initialized to 0, set to 1 after REWRITE or TRUNCATE of a sequential organization file.
FAB$V_TMD	Temporary (marked for deletion)	1, if nonexternal file with no FILE_NAME specified and DISPOSITION:=DELETE specified or implied.
FAB$B_FSZ	Fixed control area size	2, if terminal file enabled for prompting.
FAB$W_IFI	Internal file identifier	Returned by RMS.
FAB$W_MRS	Maximum record size	RECORD_LENGTH if specified; file component size if ORGANIZATION is not SEQUENTIAL or if RECORD_TYPE:=FIXED.
FAB$L_NAM[1]	Name block address	Set to address of name block (the expanded and resultant string areas are set up, but the related file name string is not).
FAB$B_ORG	File organization	FAB$C_REL if ORGANIZATION:=RELATIVE; FAB$C_IDX if ORGANIZATION:=INDEXED; FAB$C_SEQ in all other cases.
FAB$B_RAT	Record attributes	
FAB$V_FTN	FORTRAN carriage control	1, if CARRIAGE_CONTROL:=FORTRAN.
FAB$V_CR	Add LF and CR	1, if CARRIAGE_CONTROL:=LIST (default for TEXT and VARYING OF CHAR files).
FAB$V_PRN	Print file format	1, if terminal file enabled for prompting.

Field	Name	OPEN Parameters and Value
FAB$B_RFM	Record format	FAB$C_FIX if RECORD_TYPE:=FIXED or if file component is of fixed size; FAB$C_VAR if RECORD_TYPE:=VARIABLE or file is VARYING or TEXT; FAB$C_STM if RECORD_TYPE:=STREAM; FAB$C_STMCR if RECORD_TYPE:=STREAM_CR; FAB$C_STMLF if RECORD_TYPE:=STREAM_LF; FAB$C_VFC if a terminal file enabled for prompting.
FAB$L_SDC	Spooling device characteristics	Returned by RMS.
FAB$L_XAB[2]	Extended attribute block address	The XAB chain always has a File Header Characteristics (FHC) extended attribute block in order to get the longest record length (XAB$W_LRL). If ACCESS_METHOD:=KEYED, key index definition blocks are also present. VSI may add additional XABs in the future. Your user-action function may insert XABs anywhere in the chain. This field is only valid during execution of user-action functions; VSI Pascal places 0 in this field after the call to OPEN.
FAB$B_SHR	File sharing	
FAB$V_SHRPUT	Allow other PUTs	1, if SHARING:=READWRITE.
FAB$V_SHRGET	Allow other GETs	1, if SHARING is not NONE (default if HISTORY:=READONLY).
FAB$V_SHRDEL	Allow other DELETEs	1, if SHARING:=READWRITE.
FAB$V_SHRUPD	Allow other UPDATEs	1, if SHARING:=READWRITE.
FAB$V_NIL	Allow no other operations	1, if SHARING:=NONE (default if HISTORY is not READONLY).

[1]After the call to OPEN, FAB$L_NAM must contain the same value it had before the call.

[2]You cannot change XABs provided by VSI, but you can add and delete XABs that you insert using a user-action function.

Table 7.3 presents the status of RMS RAB fields when you call the OPEN procedure. If a field is not included in the following table, it is initialized to zero.

Table 7.3. Setting of RMS Record Access Block Fields by a Call to the OPEN Procedure

Field	Name	OPEN Parameters and Value
RAB$L_CTX	Context	Reserved to VSI.
RAB$L_FAB[1]	FAB address	Set to address of FAB (allocated by VSI Pascal RTL).
RAB$W_ISI	Internal stream identifier	Returned by RMS.
RAB$L_KBF	Key buffer address	May be modified for individual file operations after the file is opened.

Field	Name	OPEN Parameters and Value
RAB$B_KRF	Key of reference	May be modified for individual file operations after the file is opened.
RAB$B_KSZ	Key size	May be modified for individual file operations after the file is opened.
RAB$B_RAC	Record access mode	May be modified for individual file operations after the file is opened.
RAB$L_RBF	Record address	May be modified for individual file operations after the file is opened.
RAB$L_RHB	Record header buffer	Set to address of 2-byte carriage-control information for terminal files enabled for prompting.
RAB$L_ROP	Record options	
RAB$V_NLK	No lock	May be modified for individual file operations after the file is opened.
RAB$V_RAH	Read ahead	1
RAB$V_TPT	Truncate file often PUT	May be modified for individual file operations after the file is opened.
RAB$V_UIF	Update if record exists	1, if ACCESS:=DIRECT.
RAB$V_WBH	Write behind	1
RAB$W_RSZ	Record size	May be modified for individual file operations after the file is opened.
RAB$L_STS	Completion status code	Returned by RMS.
RAB$L_UBF[1]	User record area address	Set to buffer address after file is opened (VSI Pascal RTL allocates buffer).
RAB$W_USZ[1]	User record area size	Set to size of record area; for files other than TEXT, the size is equal to the size of the component type; for TEXT files, the size is equal to the value of RECORD_LENGTH; otherwise, 255.

[1]After the call to OPEN, this field must contain the same value it had before the call.

Table 7.4 presents the status of RMS XAB fields when you call the OPEN procedure. If a field is not included in the following table, it is initialized to zero.

Table 7.4. Setting of Extended Attribute Block Fields by a Call to the OPEN Procedure

Field	Name	PASCAL OPEN Keyword and Value
XAB$B_DTP	Data type of key	Set to data type of key
XAB$B_FLG	Key option flags	
XAB$V_CHG	Changes allowed	0 if key is 0, else 1
XAB$V_DUP	Duplicates allowed	0 if key is 0, else 1
XAB$W_POS0	Key position	Position of key in indexed file
XAB$B_REF	Key of reference	Primary key is 0, first alternate key is 1, second alternate key is 2, and so on

Field	Name	PASCAL OPEN Keyword and Value
XAB$B_SIZ0	Key size	Size of key

Table 7.5 presents the status of RMS Name Block fields when you call the OPEN procedure. If a field is not included in the following table, it is initialized to zero.

Table 7.5. Setting of Name Block Fields by a Call to the OPEN Procedure

Field	Name	OPEN Keyword and Value
NAM$L_ESA[1]	Expanded string area	Address of RTL buffer
NAM$B_ESS[1]	Expanded string area	NAM$C_MAXRSS
NAM$L_RSA	Expanded string area	Address of RTL buffer
NAM$B_RSS	Expanded string area	NAM$C_MAXRSS

[1]These fields are only valid during execution of user-action functions; VSI Pascal places 0 in these fields after the call to OPEN.

For More Information:

- On opening indexed files (Section 7.1.1)

7.1.7. Default Line Limits

VSI Pascal determines a default line limit for TEXT files by translating the logical name PAS$LINELIMIT as a string of decimal digits. If this logical name has not been defined, there is no default line limit. You can override the default by calling the LINELIMIT procedure.

For More Information:

- On LINELIMIT (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

7.2. User-Action Functions

The user-action parameter of the OPEN procedure allows you to access RMS facilities not explicitly available in the VSI Pascal language by writing a function that controls the opening of the file. Inclusion of the user-action parameter causes the run-time library to call your function to open the file instead of calling RMS to open it according to its normal defaults.

The user-action parameter of the CLOSE procedure is similar to that of the OPEN procedure. It allows you to access RMS facilities not directly available in VSI Pascal by writing a function that controls the closing of the file. Including the user-action parameter causes the run-time library to call your function to close the file instead of calling RMS to close it according to its normal defaults.

When an OPEN or CLOSE procedure is executed, the run-time library uses the procedure's parameters to establish the RMS file access block (FAB) and the record access block (RAB), as well as to establish its own internal data structures. These blocks are used to transmit requests for file and record operations to RMS; they are also used to return the data contents of files, information about file characteristics, and status codes.

In order, the three parameters passed to a user-action function by the run-time library are as follows:

- FAB address
- RAB address

- File variable

A **user-action function** is usually written in VSI Pascal and includes the following:

- Modifications to the FAB or RAB, or both (optional)
- $OPEN and $CONNECT for existing files or $CREATE and $CONNECT for new files (required)
- Status check of the values returned by $OPEN or $CREATE and $CONNECT (required)
- Storage of FAB and RAB values in program variables (optional)
- Return of success or failure status value for the user-action function (required)

Note

Modification of any of the RMS file access blocks provided by the run-time library may interfere with the normal operation of the library.

Example 7.1 shows an VSI Pascal program that copies one file into another. The program features two user-action functions, which allow the output file to be created with the same size as the input file and to be given contiguous allocation on the storage media.

Example 7.1. User-Action Function

```
[INHERIT( 'SYS$LIBRARY:STARLET' )]
PROGRAM Contiguous_Copy( F_In, F_Out );

{
The input file F_In is copied to the output file F_Out.
F_Out has the same size as F_In and has contiguous
allocation.
}

TYPE
   FType = FILE OF VARYING[133] OF CHAR;

VAR
   F_In, F_Out    : FType;
   Alloc_Quantity : UNSIGNED;

FUNCTION User_Open( VAR FAB : FAB$TYPE;
                    VAR RAB : RAB$TYPE;
                    VAR F   : FType) : INTEGER;
   VAR
     Status : INTEGER;
   BEGIN          { Function User_Open }
   {Open file and remember allocation quantity }
   Status := $OPEN( FAB );
   IF ODD( Status ) THEN
      Status := $CONNECT( RAB );
   Alloc_Quantity := FAB.FAB$L_ALQ;
   User_Open := Status;
   END;           { Function User_Open }

FUNCTION User_Create( VAR FAB : FAB$TYPE;
                      VAR RAB : RAB$TYPE;
                      VAR F   : FType ) : INTEGER;
```

```
   VAR
     Status : INTEGER;
   BEGIN          { Function User_Create }
   { Set up contiguous allocation }
   FAB.FAB$L_ALQ := Alloc_Quantity;
   FAB.FAB$V_CBT := FALSE;
   FAB.FAB$V_CTG := TRUE;
   Status := $CREATE( FAB );
   IF ODD( Status ) THEN
      Status := $CONNECT( RAB );
   User_Create := Status;
   END;           { Function User_Create }

BEGIN             { main program }
{ Open files }
OPEN( F_In, HISTORY := READONLY, USER_ACTION := User_Open );
RESET(F_In );
OPEN( F_Out, HISTORY := NEW, USER_ACTION := User_Create );
REWRITE( F_Out );

{ Copy F_In to F_Out }
WHILE NOT EOF( F_In ) DO
   BEGIN
   WRITE( F_Out, F_In_^ );
   GET( F_In );
   END;

{ Close files }
CLOSE( F_In );
CLOSE( F_Out );
END.              { main program }
```

In this example, the record types FAB$TYPE and RAB$TYPE are defined in SYS$LIBRARY:STARLET, which the program inherits. The function User_Open is called as a result of the OPEN procedure for the input file F_In. The function begins by opening the file with the RMS service $OPEN. If $OPEN succeeds, the value of Status is odd; in that case, $CONNECT is performed. The allocation quantity contained in the FAB.FAB$L_ALQ field of the FAB is assigned to a variable so that this value can be used in the second user-action function. User_Open is then assigned the value of Status (in this case, TRUE), which is returned to the main program.

The function User_Create is called as a result of the OPEN procedure for the output file F_Out. The function assigns the allocation quantity of the input file to the FAB.FAB$L_ALQ field of the FAB, which contains the allocation size for the output file. The FAB field FAB.FAB$V_CBT is set to FALSE to disable the request that file storage be allocated contiguously on a best try basis. Then, the FAB field FAB.FAB$V_CTG is set to TRUE so that contiguous storage allocation is mandatory. Finally, the RMS service $CREATE is performed. If $CREATE is successful, $CONNECT will be done and the function return value will be that of $CREATE.

Once the OPEN procedures have been performed successfully, the program can then accomplish its main task, copying the input file F_In to the output file F_Out, which is the same size as F_In and has contiguous allocation. The last step in the program is to close the files.

For More Information:

- On the OPEN and CLOSE procedures (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

- On RMS file access blocks (Section 7.1.6.2)

7.3. File Sharing

Through the RMS file sharing capability, a file can be accessed by more than one executingprogram at a time or by the same program through more than one file variable. There are two kinds of file sharing: read sharing and write sharing. Read sharing occurs when several 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.

The extent to which file sharing can take place is determined by the following factors:

- Device type

  Sharing is possible only on disk files, since other files must be accessed sequentially.

- File organization

  All three file organizations permit read and write sharing on disk files.

- Explicit user-supplied information

  Whether or not file sharing actually takes place depends on two items of information that you provide for each program accessing the file. In VSI Pascal programs, this information is supplied by the values of the SHARING and HISTORY parameters in the OPEN procedure.

The HISTORY parameter determines how the program will access the file. HISTORY := NEW, HISTORY := OLD, and HISTORY := UNKNOWN determine that the program will read from and write to the file. HISTORY := READONLY determines that the program will only read from the file. If you try to open an existing file with HISTORY := OLD or HISTORY := UNKNOWN, the run-time library retries the OPEN procedure with HISTORY := READONLY if the initial OPEN fails with a privilege violation.

The SHARING parameter determines what other programs are allowed to do with the file. Read sharing can occur when SHARING := READONLY is specified by all programs that access the file. Write sharing is accomplished when all programs specify SHARING := READWRITE. To prevent sharing, specify SHARING:= NONE with the first program to access the file.

Programs that specify SHARING := READONLY or SHARING := READWRITE can access a file simultaneously; however, file sharing can fail under certain circumstances. For example, a program without either of these parameters will fail when it attempts to open a file currently being accessed by some other program. Or, a program that specifies SHARING := READONLY or SHARING := READWRITE can fail to open a file because a second program with a different specification is currently accessing that file.

When two or more programs are write sharing a file, each program should include a condition handler. This error-processing mechanism prevents program failure due to a record-locking error.

For More Information:

- On record-locking errors (Section 7.4)
- On condition handling (Chapter 8)

- On the OPEN procedure (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

7.4. Record Locking

The RMS record locking facility, along with the logic of the program, prevents two processes from accessing the same component simultaneously. It ensures that a program can add, delete, or update a component without having to do a synchronization check to determine whether that component is currently being accessed by another process.

When a program opens a relative or indexed file and specifies SHARING := READWRITE, RMS locks each component as it is accessed. When a component is locked, any program that attempts to access it fails and a record-locked error results. A subsequent I/O operation on the file variable unlocks the previously accessed component. Thus, at most one component is locked for each file variable.

If you use the READ procedure, VSI Pascal will implicitly unlock the component by executing the UNLOCK procedure during the execution of the READ procedure.

An VSI Pascal program can explicitly unlock a component by executing the UNLOCK procedure. To minimize the time during which a component is locked against access by other programs, the UNLOCK procedure should be used in programs that retrieve components from a shared file but that do not attempt to update them. VSI Pascal requires that a component be locked before a DELETE or an UPDATE procedure can be executed.

For More Information:

- On the OPEN, UNLOCK, DELETE, and UPDATE procedures (*VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/])

Chapter 8. Error Processing and Condition Handling

This chapter discusses condition handling terms and provides examples of condition handlers.

An exception condition is an event, usually an error, that occurs during program execution and is detected by system hardware or software or the logic in a user application program. A **condition handler** is a routine that is used to resolve exception conditions.

By default, the Condition Handling Facility (CHF) provides condition handling sufficient for most VSI Pascal programs. The CHF also processes user-written condition handlers.

The use of condition handlers requires considerable programming experience. You should understand the discussions of condition handling in the following volumes before attempting to write your own condition handler:

- *VSI OpenVMS Programming Concepts Manual*
- *VSI OpenVMS System Services Reference Manual*
- *VSI OpenVMS Calling Standard*

8.1. Condition Handling Terms

The following terms are used in the discussion of condition handling:

- Condition value—An integer value that identifies a specific condition.
- Stack frame—A standard data structure built on the stack during a routine call, starting from the location addressed by the frame pointer (FP) and proceeding to both higher and lower addresses; it is popped off the stack during the return from a routine.
- Routine activation—The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new routine activation is created every time a routine is called and is deleted when control passes from the routine.
- Establish—The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In VSI Pascal, condition handlers are established by means of the predeclared procedure ESTABLISH. A routine that establishes a condition handler is known as an establisher.
- Program exit status—The status of the program at its completion.
- Signal—The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system-supplied library routines, and by user routines. All signals are initiated by a call to the signaling facility, for which there are two entry points:
    - LIB$SIGNAL—Used to signal a condition and, possibly, to continue program execution
    - LIB$STOP—Used to signal a severe error and discontinue program execution, unless a condition handler performs an unwind operation

- Resignal—The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the value SS$_RESIGNAL.
- Unwind—The return of control to a particular routine activation, bypassing any intermediate routine activations. For example, if X calls Y, and Y calls Z, and Z detects an error, then a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

8.2. Overview of Condition Handling

When the VSI OpenVMS system creates a user process, a system-defined condition handler is established in the absence of any user-written condition handler. The system-defined handler processes errors that occur during execution of the user image. Thus, by default, a run-time error causes the system-defined condition handler to print error messages and to terminate or continue execution of the image, depending on the severity of the error.

When a condition is signaled, the system searches for condition handlers to process the condition. The system conducts the search for condition handlers by proceeding down the stack, frame by frame, until a condition handler is found that does not resignal. The default handler calls the system's message output routine to send the appropriate message to the user. Messages are sent to the SYS$OUTPUT and SYS$ERROR files. If the condition is not a severe error, program execution continues. If the condition is a severe error, the default handler forces program termination, and the condition value becomes the program exit status.

You can create and establish your own condition handlers according to the needs of your application. For example, a condition handler could create and display messages that describe specific conditions encountered during the execution of your program, instead of relying on system error messages.

8.2.1. Condition Signals

A condition signal consists of a call to either LIB$SIGNAL or LIB$STOP, the two entry points to the signaling facility. These entry points can be inherited from SYS$LIBRARY:PASCAL$LIB_ROUTINES.PEN.

If a condition occurs in a routine that is not prepared to handle it, a signal is issued to notify other active routines. If the current routine can continue after the signal is propagated, you can call LIB$SIGNAL. A higher-level routine can then determine whether program execution should continue. If the nature of the condition does not allow the current routine to continue, you can call LIB$STOP.

8.2.2. Handler Responses

A condition handler responds to an exception condition by taking action in three major areas:

- Condition correction
- Condition reporting
- Execution control

The handler first determines whether the condition can be corrected. If so, it takes the appropriate action and execution continues. If the handler cannot correct the condition, the condition may be resignalled; that is, the handler requests that another condition handler be sought to process the condition.

A handler's condition reporting can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execution
- Resignalling the same condition to send the appropriate message to the output file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition, for example, the production of a message designed for a specific application

A handler can control execution in several ways:

- By continuing from the signal. If the signal was issued through a call to LIB$STOP, the program exits.
- By doing a nonlocal GOTO operation (see Section 8.4, Example 5).
- By unwinding to the establisher at the point of the call that resulted in the exception. The handler can then determine the function value returned by the called routine.
- By unwinding to the establisher's caller (the routine that called the routine which established the handler). The handler can then determine the function value returned by the called routine.

8.3. Writing Condition Handlers

The following sections describe how to write and establish condition handlers and provide some simple examples.

8.3.1. Establishing and Removing Handlers

To use a condition handler, you must first declare the handler as a routine in the declaration section of your program; then, within the executable section, you must call the predeclared procedure ESTABLISH. The ESTABLISH procedure sets up an VSI Pascal language-specific condition handler that in turn allows your handler to be called. User-written condition handlers set up by ESTABLISH must have the ASYNCHRONOUS attribute and two integer array formal parameters. Such routines can access only local, read-only, and volatile variables, and local, predeclared, and asynchronous routines.

Because condition handlers are asynchronous, any attempt to access a nonread-only or nonvolatile variable declared in an enclosing block will result in a warning message. The predeclared file variables INPUT and OUTPUT are such nonvolatile variables; therefore, simultaneous access to these files from both an ordinary program and from an asynchronous condition handler's activation may have undefined results. The following steps outline the recommended method for performing I/O operations from a condition handler:

1. Declare a file with the VOLATILE attribute at program level.
2. Open this file to refer to SYS$INPUT, SYS$OUTPUT, or another appropriate file.
3. Use this file in the condition handler.

External routines (including system services) that are called by a condition handler require the ASYNCHRONOUS attribute in their declaration.

You should set up a user-written condition handler with the predeclared procedure ESTABLISH rather than with the run-time library routine LIB$ESTABLISH. ESTABLISH follows the VSI Pascal procedure-calling rules and is able to handle VSI Pascal condition handlers more efficiently than LIB$ESTABLISH. A condition handler set up by LIB$ESTABLISH might interfere with the default error handling of the VSI Pascal run-time system, and cause unpredictable results.

The following example shows how to establish a condition handler using the VSI Pascal procedure ESTABLISH:

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr) : INTEGER;
   EXTERN;
   .
   .
   .
ESTABLISH (Handler);
```

To establish the handler, call the ESTABLISH procedure. To remove an established handler, call the predeclared procedure REVERT, as follows:

```
REVERT;
```

As a result of this call, the condition handler established in the current stack frame is removed. When control passes from a routine, any condition handler established during the routine's activation is automatically removed.

8.3.2. Declaring Parameters for Condition Handlers

A condition handler is an integer-valued function that is called when a condition is signaled. Two formal VAR parameters must be declared for a condition handler:

- An integer array to refer to the parameter list from the call to the signal routine (the signal array); that is, the list of parameters included in calls to LIB$SIGNAL or LIB$STOP (see Section 8.2.1)
- An integer array to refer to information concerning the routine activation that established the condition handler (the mechanism array). The size and contents of the mechanism array is different on VSI OpenVMS I64, VSI OpenVMS Alpha, and VSI OpenVMS x86-64 systems.

For example, a condition handler can be defined as follows:

```
TYPE
   Sigarr  = ARRAY[0..9] OF INTEGER;
   Mecharr = ARRAY[0..(SIZE(CHF2$TYPE)-4) DIV 4] OF INTEGER;

[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs : Sigarr;
    VAR Mechargs : Mecharr)   : INTEGER;
   EXTERN;
      .
      .
      .
ESTABLISH (Handler);
      .
      .
      .
```

The signal procedure passes the following values to the array Sigargs:

Value	Description
`Sigargs[0]`	The number of parameters being passed in this array (parameter count).
`Sigargs[1]`	The primary condition being signaled (condition value). See Section 8.3.4 for a discussion of condition values.

Value	Description
`Sigargs[2 to n]`	The optional parameters supplied in the call to LIB$SIGNAL or LIB$STOP; note that the index range of Sigargs should include as many entries as are needed to refer to the optional parameters.

The routine that established the condition handler passes the following values, which contain information about the establisher's routine activation, to the array Mechargs:

VSI OpenVMS I64 Value	VSI OpenVMS Alpha Value	VSI OpenVMS x86-64 Value	Description
`Mechargs[0]`	`Mechargs[0]`	`Mechargs[0]`	The number of parameters being passed in this array.
Not available	`Mechargs[2]`	`Mechargs[2]`	The address of the stack frame that established the handler.
`Mechargs[2]`	Not available	Not available	The previous stack pointer for the frame that established the handler.
`Mechargs[4]`	`Mechargs[4]`	`Mechargs[4]`	The number of calls that have been made (that is, the stack frame depth) from the routine activation up to the point at which the condition was signaled.
`Mechargs[12]`	`Mechargs[12]`	`Mechargs[12]`	The value of register R0 (R8 on OpenVMS I64, %rax on OpenVMS x86-64) at the time of the signal.
`Mechargs[14]`	`Mechargs[14]`	`Mechargs[14]`	The value of register R1 (R9 on OpenVMS I64, %rdx on OpenVMS x86-64) at the time of the signal.

For a complete description of the mechanism array, see the *VSI OpenVMS Calling Standard*.

8.3.3. Handler Function Return Values

Condition handlers are functions that return values to control subsequent execution. These values and their effects are listed as follows:

Value	Effect
SS$_CONTINUE	Continues execution from the signal. If the signal was issued by a call to LIB$STOP, the program does not continue, but exits.
SS$_RESIGNAL	Resignals to continue the search for a condition handler to process the condition.

In addition, a condition handler can request a stack unwind by calling the $UNWIND system service routine. You can inherit $UNWIND from SYS$LIBRARY:STARLET.PEN.

When $UNWIND is called, the function return value of the condition handler is ignored. The handler modifies the saved registers R0 and R1 in the mechanism parameters to specify the called function's return value.

A stack unwind is usually made to one of two places:

- The point in the establisher at which the call was made that resulted in the exception. Specify the following:

```
Status := $UNWIND (Mechargs[4],0);
```

- The routine that called the establisher. Specify the following:

```
Status := $UNWIND (Mechargs[4]+1,0);
```

8.3.4. Condition Values and Symbols

The VSI OpenVMS system uses condition values to indicate that a called routine has either executed successfully or failed, and to report exception conditions. Condition values are usually symbolic names that represent 32-bit packed records, consisting of fields (usually interpreted as integers) that indicate which system component generated the value, the reason the value was generated, and the severity of the condition.

A warning severity code (0) indicates that although output was produced, the results may be unpredictable. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results may not be correct. A severe error code (4) indicates that the error was of such severity that no output was produced.

A condition handler can alter the severity code of a condition value to allow execution to continue or to force an exit, depending on the circumstances.

Occasionally a condition handler may require a particular condition to be identified by an exact match; that is, each bit of the condition value bits (0..31) must match the specified condition. For example, you may want to process a floating overflow condition only if the severity code is still 4 (that is, if no previous condition handler has changed the severity code) and the control bits have not been modified. A typical condition handler response is to change the severity code and resignal.

In most cases, however, you want some response to a condition, regardless of the value of the severity code or control bits. To ignore the severity and control fields of a condition value, declare and call the LIB$MATCH_COND function.

8.3.5. Using Condition Handlers that Return SS$_CONTINUE

VSI Pascal condition handlers can do one of the following after appropriately responding to the error:

- Use a nonlocal GOTO to transfer control to a label in an enclosing block
- Return SS$_CONTINUE if the handler is conditioned to dismiss the error then signal to continue processing
- Return SS$_RESIGNAL if the handler is conditioned to continue searching for additional handlers to call
- Call the $UNWIND system service to establish a new point to resume execution when the handler returns to the system

When an exception occurs, the system calls a handler in the Pascal Run-Time Library that is established by the generated code. This handler in the RTL in turn calls the user condition handler that was established with the ESTABLISH built-in routine.

The RTL handler contains a check to prevent a user handler from returning SS$_CONTINUE for a certain class of Pascal Run-Time Errors that could cause an infinite loop if execution was to continue at the point of the error.

There is a situation in which this check may cause unexpected behavior:

The user handler called $UNWIND and then returned with SS$_CONTINUE. Because the $UNWIND service was called, execution will not resume at the point of the error even if SS$_CONTINUE is returned to the system. However, the RTL handler is not aware that $UNWIND has been called, and will report that program operation cannot continue for this type of error. The solution is to return SS$_RESIGNAL instead of SS$_CONTINUE after calling $UNWIND in the user handler.

However, this solution is not possible if you establish the LIB$SIG_TO_RET routine with the ESTABLISH built-in routine. LIB$SIG_TO_RET is a routine that can be used as a condition handler to convert a signal into a return to the caller of the routine that established LIB$SIG_TO_RET. Because LIB$SIG_TO_RET returns SS$_NORMAL, which in turn is the same value as SS$_CONTINUE, the handler in the Pascal RTL will report that program operation cannot continue for this type of error. The solution for this case is to establish your own handler with the ESTABLISH built-in routine that calls LIB$SIG_TO_RET and then returns SS$_RESIGNAL. You cannot establish LIB$SIG_TO_RET directly as a handler with the ESTABLISH built-in routine.

For More Information:

- On the format of a condition value (*VSI OpenVMS Calling Standard*)
- On calling the LIB$MATCH_COND function (Section 8.4)

8.4. Examples of Condition Handlers

The examples in this section inherit the $UNWIND system service routine from SYS$LIBRARY:STARLET.PEN. They also assume the following declaration has been made:

```
[INHERIT( 'SYS$LIBRARY:STARLET', 'SYS$LIBRARY:PASCAL$LIB_ROUTINES' )]
PROGRAM Error_Handling( INPUT, OUTPUT);

TYPE
   Sig_Args  = ARRAY[0..100] OF INTEGER;        { Signal parameters }
   Mech_Args = ARRAY[0..(SIZE(CHF2$TYPE)-4)DIV 4] OF [UNSAFE] INTEGER;
                                                { Mechanism parameters }
```

Example 1

```
[ASYNCHRONOUS] FUNCTION Handler_0
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                        { do something appropriate }
          .
      Handler_0 := SS$_CONTINUE;  { condition handled,
                                    propagate no further }
      END
   ELSE
      Handler_0 := SS$_RESIGNAL;  { propagate condition
                                    status to other handlers }
   END;
```

This example shows a simple condition handler. The handler identifies the condition being signaled as one that it is prepared to handle and then takes appropriate action. Note that for all unidentified condition statuses, the handler resignals. A handler must always follow this behavior.

Example 2

```
[ASYNCHRONOUS] FUNCTION Handler_1
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)   : [UNSAFE] INTEGER;

   BEGIN
   IF SA[1] = SS$_UNWIND
   THEN
      BEGIN
          .
          .                          { cleanup }
          .
      END;
   Handler_1 := SS$_RESIGNAL;
   END;
```

When writing a handler, remember that it can be activated with a condition of SS$_UNWIND, signifying that the establisher's stack frame is about to be unwound. If the establisher has special cleanup to perform, such as freeing dynamic memory, closing files, or releasing locks, the handler should check for the SS$_UNWIND condition status. If there is no cleanup, the required action of resignalling all unidentified conditions results in the correct behavior. On return from a handler activated with SS$_UNWIND, the stack frame of the routine that established the handler is deleted (unwound).

Example 3

```
[ASYNCHRONOUS] FUNCTION Handler_2
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                   { cleanup }
          .

      MA[12] := expression;       { establish function result }

     $UNWIND;                     { unwind to caller of establisher }

     END;
   Handler_2 := SS$_RESIGNAL;
   END;
```

A handler can perform a default unwind to force return to the caller of its establisher. If the establisher is a function whose result is expected in R0 or R0 and R1, the handler must establish the return value by modifying the appropriate positions of the mechanism array (the locations of the return R0 and R1 values). If the establisher is a function whose result is returned by the extra-parameter method, the handler must establish the condition value by assignment to the function identifier. In this case, you must observe two additional restrictions:

- The handler must be nested within the function
- The function result must be declared VOLATILE

Example 4

```
[ASYNCHRONOUS] FUNCTION Handler_3
    (VAR SA : Sig_Args;
     VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                  { cleanup }
          .
      MA[12] := expression;   { establish function result seen by caller }

      $UNWIND (MA[4]);        { unwind to establisher }
      END;
   Handler_3 := SS$_RESIGNAL;
   END;
```

A handler can also force return to its establisher immediately following the point of call. In this case, you should make sure that the handler understands whether the currently uncompleted call was a function call (in which case a returned value is expected) or a procedure call. If the uncompleted call is a function call that will return a value in R0 or R0 and R1, then the handler can modify the mechanism array to supply a value. If, however, the uncompleted call is a function call that will return a value using the extra-parameter mechanism, then there is no way for the handler to supply a value.

Example 5

```
[ASYNCHRONOUS] FUNCTION Handler_4
    (VAR SA : Sig_Args;
     VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      GOTO 99;
   Handler_4 := SS$_RESIGNAL;
   END;
```

A handler can force control to resume at an arbitrary label in its scope. Note that this reference is to a label in an enclosing block, because a GOTO to a local label will remain within the handler. In accordance with the *VSI OpenVMS Calling Standard*, VSI Pascal implements references to labels in enclosing blocks by signaling SS$_UNWIND in all stack frames that must be deleted.

Example 6

```
FUNCTION EXP_With_Status
   (X : REAL;
    VAR Status : INTEGER )    : REAL;

   FUNCTION MTH$EXP
```

```
      (A : REAL) : REAL;
      EXTERNAL;

   [ASYNCHRONOUS] FUNCTION Math_Error
      (VAR SA : Sig_Args;
       VAR MA : Mech_Args)       : [UNSAFE] INTEGER;

      BEGIN   { Math_Error }
      IF LIB$MATCH_COND (SA[1], MTH$_FLOOVEMAT, MTH$_FLOUNDMAT) <> 0
      THEN
         BEGIN
            IF ODD( Status )                { record condition status
            THEN                              if no previous error }

            Status := SA[1]::Cond_Status;   { condition handled,
            Math_Error := SS$_CONTINUE;       propagate no further }
         END
       ELSE
          Math_Error := SS$_RESIGNAL;       { propagate condition status
                                              to other handlers }
       END;

   BEGIN   { EXP_With_Status }
   STATUS := SS$_SUCCESS;
   ESTABLISH (Math_Error);
   EXP_With_Status := MTH$EXP (X);
   END;
```

This example shows a handler that records the condition status if a floating overflow or underflow error is detected during the execution of the mathematical function MTH$EXP.

Example 7

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Use_A_Handler(INPUT,OUTPUT);

TYPE
   Sigarr  = ARRAY [0..9] OF INTEGER;
   Mecharr = ARRAY [0..(Size(CHF2$TYPE)-4)DIV 4)] OF INTEGER;
VAR
   F1,F2 : REAL;
[ASYNCHRONOUS] FUNCTION My_Handler
    (VAR Sigargs  : Sigarr;
     VAR Mechargs : Mecharr) : INTEGER;

   VAR
      Outfile : TEXT;

    [ASYNCHRONOUS] FUNCTION LIB$FIXUP_FLT
       (VAR Sigargs  : Sigarr;
        VAR Mechargs : Mecharr;
           New_Opnd : REAL := %IMMED 0) : INTEGER;
       EXTERNAL;

    [ASYNCHRONOUS] FUNCTION LIB$SIM_TRAP
       (VAR Sigargs  : Sigarr;
        VAR Mechargs : Mecharr) : INTEGER;
       EXTERNAL;
       BEGIN
       OPEN(Outfile,'TT:');
       REWRITE(Outfile);
```

```
      { Handle various conditions }
      CASE Sigargs[1] OF

      { Convert floating faults to traps }
      SS$_FLTDIV_F, SS$_FLTOVF_F :
         LIB$SIM_TRAP(Sigargs,Mechargs);

      { Handle the floating divide by zero trap }
      SS$_FLTDIV :
         BEGIN
            WRITELN(Outfile,'Floating divide by zero');
            My_Handler := SS$_CONTINUE;
         END;

      { Handle the floating overflow trap }
      SS$_FLTOVF :
         BEGIN
            WRITELN(Outfile,'Floating overflow');
            My_Handler := SS$_CONTINUE;
         END;

      { Handle taking the square root }
      MTH$_SQUROONEG :
          BEGIN
             WRITELN(Outfile,'Square root of a negative number');
             My_Handler := SS$_CONTINUE;
          END;

      { Handle the reserved operand left by SQRT }
      SS$_ROPRAND :
          BEGIN
             WRITELN(Outfile,'Reserved floating operand');
             LIB$FIXUP_FLT(Sigargs,Mechargs);
             My_Handler := SS$_CONTINUE;
          END;

      OTHERWISE
         BEGIN
            WRITELN(Outfile,'Condition occurred, ',HEX(Sigargs[1]));
            My_Handler := SS$_RESIGNAL;
         END;

      END;

      CLOSE(Outfile);

      END;
BEGIN
ESTABLISH(My_Handler);
F1 := 0.0;
F2 := 1E38;

{ Generate exception conditions }
F1 := F2 / 0.0;
F1 := F2 * f2;
F1 := SQRT(-1.0);
END.
```

Chapter 9. Migrating Between Different Architectures

This chapter provides information on issues that affect programs being moved between different OpenVMS platforms.

9.1. Sharing Environment Files Across Platforms

VSI Pascal inherits environment files created from a compiler for the same target platform. For example, you cannot inherit environment files generated by VSI Pascal for OpenVMS Alpha with the VSI Pascal compiler for VSI OpenVMS I64.

9.2. Default Size for Enumerated Types and Booleans

The default size for enumerations and Booleans in unpacked structures is longword on all current VSI OpenVMS systems. On legacy OpenVMS VAX systems, the default size was byte for Booleans and small enumerations or words for larger enumerations.

If you need the OpenVMS VAX behavior on current VSI OpenVMS systems, you can use one of the following:

- **`/ENUMERATION_SIZE=BYTE`** qualifier
- [ENUMERATION_SIZE(BYTE)] attribute
- Individual [BYTE] or [WORD] attributes on the affected fields or components

The default for OpenVMS VAX compilers was **`/ENUMERATION_SIZE=BYTE`**, for compatibility.

9.3. Default Data Layout for Unpacked Arrays and Records

On current VSI OpenVMS systems, the default data layout is “natural” alignment, where record fields and array components are aligned on boundaries based on their size (for example, INTEGERs on longword boundaries, INTEGER64s on quadword boundaries).

On OpenVMS VAX systems, the default alignment rule was to allocate such fields on the next byte boundary. If you need the OpenVMS VAX behavior, you can use the **`/ALIGN=VAX`** qualifier or the [ALIGN(VAX)] attribute.

9.4. Overflow Checking

When overflow checking is enabled, the INT built-in signals a run-time error if its actual parameter cannot be represented as an INTEGER32 value.

If you have a large unsigned value that you wish to convert to a negative integer, you must use a typecast to perform the operation.

9.5. Bound Procedure Values

On OpenVMS VAX systems, a Bound Procedure Value was a 2-longword data structure holding the address of the entry point and a frame-pointer to define the nested environment. VAX Pascal expected one of these 2-longword structures for PROCEDURE or FUNCTION parameters.

A routine not written in Pascal needed different code depending on whether it would receive a Bound Procedure Value or a simple routine address. When passing routines to %IMMED formal routine parameters, VAX Pascal passed the address of the entry point; otherwise, it passed the address of a Bound Procedure Value.

On VSI OpenVMS I64 and VSI OpenVMS Alpha systems, a Bound Procedure Value is a special type of procedure descriptor that invokes a hidden jacket routine that in turn initializes the static-link-pointer and calls the real routine. VSI Pascal expects a function or a procedure descriptor for PROCEDURE or FUNCTION parameters.

On VSI OpenVMS x86-64 systems, a Bound Procedure Value is a compiler-generated routine that initializes the special %r10 static link register and calls the real routine. VSI Pascal expects a code address for PROCEDURE or FUNCTION parameters.

A routine not written in Pascal does not require difference code for Bound Procedure Values. When passing routines to %IMMED formal routine parameters, (or asking for the IADDRESS of a routine) VSI Pascal passes the address of a procedure descriptor as if the %IMMED was not present.

9.6. Different Descriptor Classes for Conformant Array Parameters

VSI Pascal uses the “by descriptor” mechanism to pass conformant parameters from one routine to another. For conformant array parameters, VSI Pascal uses a CLASS_NCA descriptor on VSI OpenVMS systems. The CLASS_NCA descriptors generate more efficient code when accessing array components and are able to describe arrays with alignment holes or padding.

If you have a foreign routine that constructs CLASS_A descriptors for Pascal, you need to examine the code to see if changes are necessary:

- For certain actual parameters, the CLASS_A and CLASS_NCA descriptors are identical except for the DSC$B_CLASS field (which VSI Pascal does not examine).
- For other parameters, you will either have to generate a CLASS_NCA descriptor or you can add an explicit CLASS_A attribute to the formal conformant parameter in the Pascal routine.

9.7. Data Layout and Conversion

On VSI OpenVMS I64 and VSI OpenVMS Alpha systems, the layout of data can severely impact performance. The Itanium and Alpha architecture and the VSI OpenVMS I64 and VSI OpenVMS Alpha systems have strong preferences about data alignment and size.

OpenVMS x86-64 systems are barely impacted by unaligned data but aligning data can result in shorter code sequences which can execute faster.

The VSI Pascal compiler has several features to enable you to write Pascal code that will get the best performance on the target system.

The remainder of this section describes the different types of record layouts, VSI Pascal features that support them, how to get the best performance with your data structures, and how to convert existing code for better performance.

This section focuses on records, but arrays also have similar properties. In almost all cases, where record fields are discussed, you can substitute array components.

9.7.1. Natural Alignment, VAX Alignment, and Enumeration Sizes

The compiler has the ability to lay out records in two ways:

- OpenVMS VAX alignment

  Fields and components less than or equal to 32 bits are allocated on the next available bit; otherwise they are allocated on the next available byte.

- Natural alignment where an object is aligned based on its size

  Essentially fields and components are allocated on the next naturally aligned address for their data type. For example:

  - 8-bit character strings should start on byte boundaries
  - 16-bit integers should start at addresses that are a multiple of 2 bytes (word alignment)
  - 32-bit integers and single-precision real numbers should start at addresses that are a multiple of 4 bytes (longword alignment)
  - 64-bit integers and double-precision real numbers should start at addresses that are a multiple of 8 bytes (quadword alignment)

For aggregates such as arrays and records, the data type to be considered for purposes of alignment is not the aggregate itself, but rather the elements of which the aggregate is composed. Varying 8-bit character strings must, for example, start at addresses that are a multiple of 2 bytes (word alignment) because of the 16-bit count at the beginning of the string. For records, the size is rounded up to a multiple of their natural alignment (a record with natural alignment of longword has a size that is a multiple of longwords, for example).

The OpenVMS VAX and naturally aligned record formats are fully documented in the *VSI OpenVMS Calling Standard*.

The size as well as the alignment of record fields and array components can affect performance. VSI Pascal uses larger allocation for unpacked Booleans and enumeration types to help performance, as shown in Table 9.1.

Table 9.1. Unpacked Sizes of Fields and Components

Datatype	Unpacked Size with VAX Alignment	Unpacked Size with Natural Alignment
Boolean	1 byte	4 bytes

Datatype	Unpacked Size with VAX Alignment	Unpacked Size with Natural Alignment
Enumerated types	1 or 2 bytes	4 bytes

For compatibility reasons, the size of all data types in PACKED records and arrays are the same for both VAX and natural alignment formats.

9.7.2. VSI Pascal for OpenVMS Features Affecting Data Alignment and Size

VSI Pascal has the following DCL qualifiers:

- **/ALIGN=*option***, where ***option*** is either NATURAL or VAX
- **/ENUMERATION_SIZE=*option***, where ***option*** is either BYTE or LONG

The **/ALIGN** qualifier option controls the default record format used by the compiler. The **/ENUMERATION_SIZE** qualifier option controls whether the compiler allocates Boolean and enumeration types as longwords or as 1 or 2 bytes.

The default alignment format is **NATURAL** and the default enumeration size is **LONG**. The other settings exist to aid in porting legacy code from OpenVMS VAX systems.

A corresponding pair of attributes can be used at the PROGRAM/MODULE level and on VAR and TYPE sections to specify the desired alignment format and enumeration size:

- ALIGN (*option*), where *option* is either NATURAL or VAX
- ENUMERATION_SIZE (*option*), where *option* is either BYTE or LONG

By using these attributes at the MODULE level, you can extract the records into a separate module and create an environment file with the desired alignment format. By using these attributes on VAR or TYPE sections, you can isolate the records in the same source file.

9.7.3. Optimal Record Layout

The optimal record layout is one where all the record's fields are naturally sized on naturally aligned boundaries and the overall record is as small as possible (for example, the fewest number of padding bytes required for proper alignment).

The compiler automatically places all fields of unpacked records on naturally aligned boundaries.

To allow the compiler to do this placement, you should refrain from using explicit positioning and alignment attributes on record fields unless required by your application. The keyword PACKED should be avoided in all cases except:

- PACKED ARRAY OF CHARs require the PACKED keyword to be manipulated as strings. Since chars are each 1 byte, using the PACKED keyword does not hurt their performance in any way.
- PACKED SETs may perform better than unpacked SETs. For PACKED SETs, the compiler can sometimes allocate fewer bits for the set field or variable. These smaller sets can often be manipulated directly with longword or quadword instructions, instead of using a generic run-time library routine for larger sets.

Inside unpacked records, PACKED SET fields are no slower than unpacked SET fields. The same holds true for variables of PACKED SETs. PACKED SETs of size 32 or 64 bits are the best performing set types; otherwise a multiple of 8 bits improves performance to a lesser degree.

You may still need to use PACKED if you rely on the record for compatability with binary data files or when assuming that types like PACKED ARRAY OF BOOLEAN are implemented as bit strings.

While the compiler can position record fields at natural boundaries, it cannot minimize the alignment bytes that are required between fields. The calling standard requires the compiler to allocate record fields in the same lexical order that they appear in the source file. For example:

```
type t1 = record
          f1 : char;
          f2 : integer;
          f3 : char;
          f4 : integer;
          end;
```

The size of this record is 16 bytes:

- F1 is a byte field, followed by 3 padding bytes to position F2 at a longword boundary
- F2 is 4 bytes
- F3 is a single byte, followed by 3 more padding bytes to position F4 at a longword boundary
- F4 is 4 bytes

The optimal layout would be:

```
type t2 = record
          f1,f2 : integer;
          f3,f4 : char;
          end;
```

The size of this record is only 12 bytes:

- F1 and F2 are placed on adjacent longword boundaries
- F3 and F4 can immediately follow, since they can appear on any byte boundary, they in turn are followed by 2 padding bytes to round the size of the record up to a multiple of its natural alignment of longword.

To achieve the fewest alignment bytes, you should place larger fields at the beginning of the record and smaller fields at the end. If you have record fields of schema types that have run-time size, you should place those at the very end of the record, since their offset requires run-time computation.

You can get the optimal record layout by:

- Avoiding the PACKED keyword except for PACKED ARRAY OF CHARs and possibly PACKED SETs
- Avoiding explicit POS or ALIGNED attributes
- Placing larger fields before smaller fields
- Placing fixed-size fields before run-time sized fields

9.7.4. Optimal Data Size

On VSI OpenVMS Alpha systems, data items that are smaller than 32 bits might impose a performance penalty, due to the additional instructions required to access them. The compiler will attempt to reorder loads and stores that manipulate adjacent items smaller than 32 bits to minimize the number of memory references required.

For performance reasons, VSI Pascal will allocate Boolean and enumerated types as longwords in unpacked records or arrays.

You should avoid any explicit size attributes on subrange types. While it is true that [BYTE] 0..255 is smaller than 0..255 (which would allocate 4 bytes, since it is a subrange of INTEGER), the additional overhead of accessing the byte-sized subrange might be than the extra 3 bytes of storage. Using the BIT attribute on subranges is even less effective in terms of the extra instructions required to manipulate a 13-bit integer subrange inside a record. Use these attributes only where needed.

9.7.5. Converting Existing Records

When moving code from a legacy OpenVMS VAX system to a current VSI OpenVMS system, you probably want to make sure that you are getting the best performance on the new system. To do that, you must use natural alignment on your record types.

9.7.6. Applications with No External Data Dependencies

If your application has no external data dependencies (such as no stored binary data files, no binary data transmitted to some external device), then the conversion is as simple as:

- Using the default natural alignment.
- Using the default enumeration size.
- Removing any uses of PACKED that are not needed.
- Removing any explicit positioning or size attributes that are not needed.
- Optionally reordering fields to place larger fields before smaller fields. This does not make the record faster, but does make it smaller.

Depending on your data types, the removal of any PACKED keywords or attributes may make little improvement in performance. For example, a PACKED ARRAY OF REAL is identical in size and performance to an unpacked ARRAY OF REAL.

VSI Pascal has two features to help you identify poorly aligned records and how often they are used:

- The **`/USAGE=PERFORMANCE`** command-line option

  This option causes the compiler to generate messages for declarations and uses of record fields that are poorly aligned or poorly sized. For example:

  ```
  program a;

  type r = packed record
           f1 : boolean;
           f2 : integer;
           end;
  ```

```
begin
end.
```

In this program the compiler can highlight the following:

```
$ pascal/usage=performance test.pas

         f1 : boolean;
.........^
%PASCAL-I-COMNOTSIZ, Component is not optimally sized
at line number 4 in file DISK$:[DIR]TEST.PAS;32
         f2 : integer;
.........^
%PASCAL-I-COMNOTALN, Component is not optimally aligned
at line number 5 in file DISK$:[DIR]TEST.PAS;32
%PASCAL-S-ENDDIAGS, PASCAL completed with 2 diagnostics
```

In this example, the size of the Boolean field in the PACKED ARRAY is only 1 bit. Single bit fields require additional instructions to process. The integer field is not aligned on a well-aligned boundary for the target system. The **/USAGE=PERFORMANCE** qualifier gives performance information customized to the target system. For example, INTEGERs should be on a longword boundary for "good" performance.

- The **/SHOW=STRUCTURE_LAYOUT** command-line option.

  This option causes the compiler to generate a structure layout summary in the listing file.

  This summary gives size and offset information about variables, types, and fields. It also flags the same information as the **/USAGE=PERFORMANCE** command-line option.

  For example, compiling the above program with the following command produces the following in the listing file:

```
$ pascal/list/show=structure_layout test.pas

Comments      Offset         Size
-----------  -----------  -----------
                            5 Bytes     R {In PROGRAM A} = PACKED
 RECORD
Size             0 Bytes      1 Bit           F1 : BOOLEAN
Align            1 Bit        4 Bytes         F2 : INTEGER
                                              END
```

  This output shows the size of the record R as well as the sizes and offsets of the records fields. It also highlights any components that were poorly sized or poorly aligned.

9.7.7. Applications with External Data Dependencies

If your application has external data dependencies, the process is more involved, since you have to isolate and understand the dependencies.

Possible steps when porting legacy VAX Pascal code include:

- Using the **/ALIGN=VAX** qualifier
- Using the **/ENUMERATION_SIZE=BYTE** qualifier

- Using the **/FLOAT=D_FLOAT** qualifier (if you have any DOUBLE binary data)
- Leaving the code exactly as is

This should produce the same behavior on a VSI OpenVMS system as you had on your OpenVMS VAX system with the following exception: Using D_Floating data on VSI OpenVMS systems only provides 53 bits of mantissa instead of 56 bits as on VAX systems; using D_floating data on VSI OpenVMS systems causes the compiler to convert to/from D_Floating data and IEEE T_Floating data to actually perform any needed operations.

You then have to identify which records in your program have external data dependencies. These include binary files (for example, FILE OF xxx), shared memory sections with other programs, and binary information passed to a library routine (such as an VSI OpenVMS item list).

You can immediately begin to convert records without external data dependencies into optimal format (for example, remove any unneeded PACKED keywords and attributes as described earlier).

You need to classify records with external dependencies further into:

- Records that cannot be naturally aligned due to a hard dependency that cannot be changed (like a record that maps onto an external piece of hardware, or a record that is passed to some software you cannot change).
- Records that can be changed after conversion of binary data or cooperating software.

Isolate records that you cannot change into their own environment file by using **/ALIGN=VAX**, **/ENUM=BYTE**, and **/FLOAT=D_FLOAT**. You can also attach the ALIGN and ENUMERATION_SIZE attributes to the TYPE or VAR sections that define these records. In this case, you need to also change any uses of the DOUBLE datatype to the D_FLOAT datatype, to ensure that the proper floating format is used.

You do not need to isolate the record if it uses the PACKED keyword, since PACKED records are identical regardless of the **/ALIGN** or **/ENUM** qualifiers. Nevertheless, isolating the records with dependencies might be useful in the future if you eventually intend to change the format.

For records that you might change, you need to decide whether it is worthwhile to convert the record and any external binary data. If the record is of low use and you have a large quantity of external data, the cost of conversion is probably too high. If a record is of high use but is mostly aligned, then the conversion also may not be worthwhile. However, a high-use record that is poorly aligned suggests conversion of external data regardless of the amount of effort required.

There are two types of poorly aligned records:

- Records that use the PACKED keyword

  PACKED records lay out the same with either setting of the **/ALIGN** or **/ENUMERATION_SIZE** qualifiers. To get natural alignment, you must remove the PACKED keyword. However, the keyword PACKED by itself does not guarantee poor alignment. For example:

  ```
  type t = packed record
           f1,f2 : integer;
           end;
  ```

  This record is well aligned with or without the PACKED keyword. It is also well aligned with **/ALIGN=NATURAL** and **/ALIGN=VAX**. You can remove the PACKED keyword for completeness, but nothing else needs to be done.

- Unpacked records that lay out differently with **/ALIGN=NATURAL** and **/ALIGN=VAX**

  These records automatically are well-aligned by the compiler when recompiled with **/ALIGN=NATURAL**. However, there are some unpacked records are already well-aligned with both alignment formats. For example:

  ```
  type t = record
           f1,f2 : integer;
           end;
  ```

  This unpacked record is well aligned with **/ALIGN=NATURAL** and **/ALIGN=VAX**. Nothing else needs to be done to this record.

The **/USAGE=PERFORMANCE** and **/SHOW=STRUCTURE_LAYOUT** DCL qualifiers can be helpful for identifying poorly aligned records.

For PACKED keywords, you can compile with and without the PACKED keyword to see if the fields are positioned at the same offsets or not.

You have classified the records with external data dependencies into:

- Records that are well-aligned with both alignment/enumeration formats
- Records that are poorly aligned, where conversion is not worthwhile
- Records that are poorly aligned, where conversion is worthwhile

For the well-aligned records, no additional work is needed now, but be aware that you still have an external data dependency that might cause problems if you add fields to the record in the future.

Isolate records that are not being converted into the same environment file or into the TYPE or VAR sections where you placed the records that you could not convert.

For records that are worth converting, you need to plan how to convert the external binary data or cooperating software. For cooperating software, you need to ensure that it gets modified so it views the record with the “natural” layout. You can determine the layout by using the **/SHOW=STRUCTURE_LAYOUT** command-line option described above. For binary data, you need to write a conversion program.

Converting existing binary data involves writing a program that reads the existing data into a poorly aligned record, copies the data into a well aligned record, and then writes out the new record.

A simple conversion program would look like:

```
program convert_it(oldfile,newfile);

[align(vax),enumeration_size(byte)]
type oldtype = packed record
               { Existing record fields }
               end;

type newtype = record
               { Record fields reorganized for optimal alignment }
               end;

var oldfile = file of oldtype;
    newfile = file of newtype;
```

```
    oldvar : oldtype;
newvar : newtype;

begin
reset(oldfile);
rewrite(newfile);
while not eof(oldfile) do
   begin
   read(oldfile,oldvar);

   { For each field, sub-field, etc. move the data }
   newvar.field1 := oldvar.field1;
   newvar.field2 := oldvar.field2;

   write(newfile,newtype);
   end;
close(oldfile);
close(newfile);
end.
```

Notice the “type” keyword before the definition of the “newtype” type. Without this keyword, “newtype” would be in the same type definition part as “oldtype” and would be processed with the same ALIGN and ENUMERATION_SIZE settings.

If you have embedded DOUBLE data, you must use the D_FLOAT predefined type in the “oldtype” definition, since the default on VSI OpenVMS I64 and VSI OpenVMS x86-64 systems is for T_floating format and the default on VSI OpenVMS Alpha systems is for G_Floating format. The compiler does not allow a simple assignment of a D_FLOAT value to a T_FLOAT or G_FLOAT variable. You need to use the CNV$CONVERT_FLOAT routine provided with VSI OpenVMS to convert the floating data.

Appendix A. Errors Returned by STATUS and STATUSV Functions

This appendix lists the error conditions detected by the STATUS and STATUSV functions, their symbolic names, and the corresponding values. The symbolic names and their values are defined in the file SYS$LIBRARY:PASSTATUS.PAS, which you can include with a %INCLUDE directive in a CONST section of your program. To test for a specific condition, you compare the STATUS or STATUSV return values against the value of a symbolic name.

The symbolic names correspond to some of the run-time errors listed in Appendix C; however, not all run-time errors can be detected by STATUS.

There is a one-to-one correspondence between the symbolic constants returned by STATUS or STATUSV documented in PASSTATUS.PAS and the VSI OpenVMS condition code values in SYS$LIBRARY:PASDEF.PAS. The following routine shows how to map the return value of STATUS to its corresponding condition code located in PASDEF.PAS:

```
FUNCTION CONVERT_STATUS_TO_CONDITION(STAT:INTEGER):INTEGER;
   BEGIN
      CONVERT_STATUS_TO_CONDITION := 16#218644 + STAT * 8;
   END;
```

Table A.1 lists the symbolic names and the values returned by the STATUS and STATUSV functions and explains the error condition that corresponds to each value.

Table A.1. STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ACCMETINC	5	Specified access method is not compatible with this file.
PAS$K_AMBVALENU	30	“String” is an ambiguous value for the enumerated type “type”.
PAS$K_CURCOMUND	73	DELETE or UPDATE was attempted while the current component was undefined.
PAS$K_DELNOTALL	100	DELETE is not allowed for a file with sequential organization.
PAS$K_EOF	–1	File is at end-of-file.
PAS$K_ERRDURCLO	16	Error occurred while the file was being closed.
PAS$K_ERRDURDEL	101	Error occurred during execution of DELETE.
PAS$K_ERRDUREXT	127	Error occurred during execution of EXTEND.
PAS$K_ERRDURFIN	102	Error occurred during execution of FIND or FINDK.
PAS$K_ERRDURGET	103	Error occurred during execution of GET.
PAS$K_ERRDUROPE	2	Error occurred during execution of OPEN.
PAS$K_ERRDURPRO	36	Error occurred during prompting.
PAS$K_ERRDURPUT	104	Error occurred during execution of PUT.
PAS$K_ERRDURRES	105	Error occurred during execution of RESET or RESETK.
PAS$K_ERRDURREW	106	Error occurred during execution of REWRITE.
PAS$K_ERRDURTRU	107	Error occurred during execution of TRUNCATE.

Name	Value	Meaning
PAS$K_ERRDURUNL	108	Error occurred during execution of UNLOCK.
PAS$K_ERRDURUPD	109	Error occurred during execution of UPDATE.
PAS$K_ERRDURWRI	50	Error occurred during execution of WRITELN.
PAS$K_EXTNOTALL	128	EXTEND is not allowed for a shared file.
PAS$K_FAIGETLOC	74	GET failed to retrieve a locked component.
PAS$K_FILALRCLO	15	File is already closed.
PAS$K_FILALROPE	1	File is already open.
PAS$K_FILNAMREQ	14	File name must be specified in order to save, print, or submit an internal file.
PAS$K_FILNOTDIR	110	File is not open for direct access.
PAS$K_FILNOTFOU	3	File was not found.
PAS$K_FILNOTGEN	111	File is not in generation mode.
PAS$K_FILNOTINS	112	File is not in inspection mode.
PAS$K_FILNOTKEY	113	File is not open for keyed access.
PAS$K_FILNOTOPE	114	File is not open.
PAS$K_FILNOTSEQ	115	File does not have sequential organization.
PAS$K_FILNOTTEX	116	File is not a text file.
PAS$K_GENNOTALL	117	Generation mode is not allowed for a read-only file.
PAS$K_GETAFTEOF	118	GET attempted after end-of-file has been reached.
PAS$K_INSNOTALL	119	Inspection mode is not allowed for a write-only file.
PAS$K_INSVIRMEM	120	Insufficient virtual memory.
PAS$K_INVARGPAS	121	Invalid argument passed to an VSI Pascal Run-Time Library procedure.
PAS$K_INVFILSYN	4	Invalid syntax for file name.
PAS$K_INVKEYDEF	9	Key definition is invalid.
PAS$K_INVRECLEN	12	Record length nnn is invalid.
PAS$K_INVSYNBIN	37	“String” is invalid syntax for a binary value.
PAS$K_INVSYNENU	31	“String” is invalid syntax for a value of an enumerated type.
PAS$K_INVSYNHEX	38	“String” is invalid syntax for a hexadecimal value.
PAS$K_INVSYNINT	32	“String” is invalid syntax for an integer.
PAS$K_INVSYNOCT	39	“String” is invalid syntax for an octal value.
PAS$K_INVSYNREA	33	“String” is invalid syntax for a real number.
PAS$K_INVSYNUNS	34	“String” is invalid syntax for an unsigned integer.
PAS$K_KEYCHANOT	72	Changing the key field is not allowed.
PAS$K_KEYDEFINC	10	KEY(nnn) definition is inconsistent with this file.
PAS$K_KEYDUPNOT	71	Duplication of key field is not allowed.
PAS$K_KEYNOTDEF	11	KEY(nnn) is not defined in this file.
PAS$K_KEYVALINC	70	Key value is incompatible with file's key nnn.

Name	Value	Meaning
PAS$K_LINTOOLON	52	Line is too long; exceeds record length by nnn characters.
PAS$K_LINVALEXC	122	LINELIMIT value exceeded.
PAS$K_NEGWIDDIG	53	Negative value in width or digits (of a field width specification) is invalid.
PAS$K_NOTVALTYP	35	“String” is not a value of type “type”.
PAS$K_ORGSPEINC	8	Specified organization is inconsistent with this file.
PAS$K_RECLENINC	6	Specified record length is inconsistent with this file.
PAS$K_RECTYPINC	7	Specified record type is inconsistent with this file.
PAS$K_RESNOTALL	124	RESET is not allowed for an internal file that has not been opened.
PAS$K_REWNOTALL	123	REWRITE is not allowed for a file opened for sharing.
PAS$K_SUCCESS	0	Last file operation completed successfully.
PAS$K_TEXREQSEQ	13	Text files must have sequential organization and sequential access.
PAS$K_TRUNOTALL	125	TRUNCATE is not allowed for a file opened for sharing.
PAS$K_UPDNOTALL	126	UPDATE is not allowed for a file that has sequential organization.
PAS$K_WRIINVENU	54	WRITE operation attempted on an invalid enumerated value.

Appendix B. Entry Points to VSI Pascal for OpenVMS Utilities

This appendix describes the entry points to utilities in the VSI OpenVMS Run-Time Library that can be called as external routines by an VSI Pascal program. These utilities allow you to access VSI Pascal extensions that are not directly provided by the language.

B.1. PAS$FAB (f)

The PAS$FAB function returns a pointer to the RMS file access block (FAB) of file f. After this function has been called, the FAB can be used to get information about the file and to access RMS facilities not explicitly available in the VSI Pascal language.

The component type of file f can be any type; the file must be open.

PAS$FAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_FAB  = ^FAB$TYPE;

FUNCTION PAS$FAB
   (VAR F : Unsafe_File) : Ptr_to_FAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$FAB. The type FAB$TYPE is defined in the VSI Pascal environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS FAB does not interfere with the normal operations of the VSI OpenVMS Run-Time Library (RTL). Future changes to the RTL may change the way in which the FAB is used, which may in turn require you to change your program.

For More Information:

- On the VSI OpenVMS Run-Time Library (*VSI OpenVMS Programming Concepts Manual*)

B.2. PAS$RAB (f)

The PAS$RAB function returns a pointer to the RMS record access block (RAB) of file f. After this function has been called, the RAB can be used to get information about the file and to access RMS facilities not explicitly available in the VSI Pascal language.

The component type of file f can be any type; the file must be open.

PAS$RAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_RAB  = ^RAB$TYPE;

FUNCTION PAS$RAB
   (VAR F : Unsafe_File) : Ptr_to_RAB;
```

```
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$RAB. The type RAB$TYPE is defined in the VSI Pascal environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS RAB does not interfere with the normal operations of the VSI OpenVMS Run-Time Library. Future changes to the RTL may change the way in which the RAB is used, which may in turn require you to change your program.

For More Information:

- On the VSI OpenVMS Run-Time Library (*VSI OpenVMS Programming Concepts Manual*)

B.3. PAS$MARK2 (s)

The PAS$MARK2 function returns a pointer to a heap-allocated object of the size specified by s. If this pointer value is then passed to the PAS$RELEASE2 function, all objects allocated with NEW or PAS$MARK2 since the object was allocated are deallocated. PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VSI Pascal. Their use is not recommended in a modular programming environment. The PAS$MARK2 and PAS$RELEASE2 routines do not work with 64-bit pointers.

While a mark is in effect, any DISPOSE operation will not actually delete the storage, but merely mark the storage for deletion. To free the memory, you must use PAS$RELEASE2.

PAS$MARK2 is an external function that must be explicitly declared. Because the parameter to PAS$MARK2 is the size of the object (unlike the parameter to the predeclared procedure NEW), the best method for using this function is to declare a separate function name for each object you wish to mark. The following example shows how PAS$MARK2 could be declared and used as a function named Mark_Integer to allocate and mark an integer variable:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer: Ptr_to_Integer;

[EXTERNAL(PAS$MARK2)] FUNCTION Mark_Integer
   (%IMMED S : Integer := SIZE(Integer))
   : Ptr_to_Integer;
   EXTERN;
   .
   .
   .
Marked_Integer := Mark_Integer;
```

The parameter to PAS$MARK2 can be 0, in which case the function value is only a pointer to a marker, and cannot be used to store data.

B.4. PAS$RELEASE2 (p)

The PAS$RELEASE2 function deallocates all storage allocated by NEW or PAS$MARK2 since the call to PAS$MARK2 allocates the parameter p.

PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VSI Pascal. Their use is not recommended in a modular programming environment. PAS$RELEASE2 disables AST delivery during its execution, so it should not be used in a real-time environment. The PAS$MARK2 and PAS$RELEASE2 routines do not work with 64-bit pointers.

PAS$RELEASE2 is an external function that must be explicitly declared. An example of its declaration and use is as follows:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer : Ptr_to_Integer;

[EXTERNAL(PAS$RELEASE2)] PROCEDURE Release
   (P :[UNSAFE] Ptr_to_Integer);
   EXTERN;
   .
   .
   .
Release (Marked_Integer);
```

In this example, Marked_Integer is assumed to contain the pointer value returned by a previous call to PAS$MARK2.

For More Information:

- On PAS$MARK2 (Section B.3)

Appendix C. Diagnostic Messages

This appendix summarizes the error messages that can be generated by an VSI Pascal program at compile time and at run time.

C.1. Compiler Diagnostics

The VSI Pascal compiler reports compile-time diagnostics in the source listing (if one is being generated) and summarizes them on the terminal (in interactive mode) or in the batch log file (in batch mode). Compile-time diagnostics are preceded by the following:

```
           I-
%PASCAL-   W-
           E-
           F-
```

Table C.1 shows the severity level indicated by each letter.

Table C.1. Compiler Message Warning Levels

Letter	Meaning
I	An informational message that flags extensions to the Pascal standard, identifies unused or possibly uninitialized variables, or provides additional information about a more severe error.
W	A warning that flags an error or construct that may cause unexpected results, but that does not prevent the program from linking and executing.
E	An error that prevents generation of machine code; instead, the compiler produces an empty object module indicating that E-level messages were detected in the source program.
F	A fatal error.

If the source program contains either E- or F-level messages, the errors must be corrected before the program can be linked and executed.

All diagnostic messages contain a brief explanation of the event that caused the error. This section lists compile-time diagnostic messages in alphabetical order, including their severity codes and explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

64BITBASTYP, 64-bit pointer base types cannot contain file variables

Error: File types may not be allocated in 64-bit P2 address space, because their implementation currently assumes 32-bit pointers in internal data structures.

64BITNOTALL, 64-bit pointers are not allowed in this context

Error: File types may not be allocated in 64-bit P2 address space, because their implementation currently assumes 32-bit pointers in internal data structures.

ABSALIGNCON, Absolute address / alignment conflict

Error: The address specified by the AT attribute does not have the number of low-order bits implied by the specified alignment attribute.

ACCMETHCON, Specified ACCESS_METHOD conflicts with file's record organization

Warning: You cannot specify ACCESS_METHOD:=DIRECT for a file that has indexed organization or sequential organization and variable-length records. You cannot specify ACCESS_METHOD:=KEYED for a file with sequential or relative organization.

ACTHASNOFRML, Actual parameter has no corresponding formal parameter

Error: The number of actual parameters specified in a routine call exceeds the number of formal parameters in the routine's declaration, and the last formal parameter does not have the LIST attribute.

ACTMULTPL, Actual parameter specified more than once

Error: Each formal parameter (except one with the LIST attribute) can have only one corresponding actual parameter.

ACTPASCNVTMP, Conversion: actual passed is resulting temporary

ACTPASRDTMP, Formal requires read access: actual parameter is resulting temporary

ACTPASSIZTMP, Size mismatch: actual passed is resulting temporary

ACTPASWRTMP, Formal requires write access: actual parameter is resulting temporary

Warning: A temporary variable is created if an actual parameter does not have the size, type, and accessibility properties required by the corresponding foreign formal parameter.

ACTPRMORD, Actual parameter must be ordinal

Error: The actual parameter that specifies the starting index of an array for the PACK or UNPACK procedure must have an ordinal type.

ADDIWRDALIGN, ADD_INTERLOCKED requires variable with at least word alignment

ADDIWRDSIZE, ADD_INTERLOCKED requires 16-bit variable

Error: These restrictions are imposed by the instruction sequence that is used on the target architecture.

ADDRESSVAR, “parameter name” is a VAR parameter, ADDRESS is illegal

Warning: You should not use the ADDRESS function on a nonvolatile variable or component or on a formal VAR parameter.

ADISCABSENT, Formal discriminant “discriminant name” has no corresponding actual discriminant

Error: An actual discriminant must be specified for every formal discriminant in a schema type definition.

ADISCHASNOFRML, Actual discriminant has no corresponding formal discriminant

Error: The number of actual discriminants specified is greater than the number of formal discriminants defined in the schema type definition.

AGGNOTALL, Aggregate variable access of this type not allowed, must be indexed

Error.

ALIATRTYPCON, Alignment attribute / type conflict

ALIGNAUTO, Alignment greater than *n* conflicts with automatic allocation

Error: The value *n* has the value 4 on VSI OpenVMS I64, VSI OpenVMS x86-64; and 3 on VSI OpenVMS Alpha. VSI OpenVMS I64 and VSI OpenVMS x86-64 systems align the stack on an octaword boundary. VSI OpenVMS Alpha systems align the stack on a quadword boundary. You cannot specify a greater alignment for automatically allocated variables.

ALIDOWN, Alignment down-graded from default of ALIGNED(*n*)

Information: The value of *n* is based on the size of the object that is being downgraded.

ALIGNFNCRES, Alignment greater than *n* not allowed on function result

Error: The value *n* has the value 4 on VSI OpenVMS I64 and OpenVMS x86-64 systems; and 3 on VSI OpenVMS Alpha systems. The use of an attribute on a routine conflicts with the requirements of the object's type.

ALIGNINT, ALIGNED expression must be integer value in range 0..n, defaulting to m

Error: The value n has the value of the largest argument to the ALIGNED attribute allowed on the platform.

ALIGNVALPRM, Alignment greater than *n* not allowed on value parameter

Error: The value *n* has the value 4 on VSI OpenVMS I64 and OpenVMS x86-64 systems; and 3 on VSI OpenVMS Alpha systems. The use of an attribute on a parameter conflicts with the requirements of the object's type.

ALLPRMSAM, All parameters to 'MIN ' or 'MAX ' must have the same type

Error.

APARMACTDEF, Anonymous parameter "parameter number" has neither actual nor default

Error: If the declaration of a routine failed to specify a name for a formal parameter, a call to the routine will result in this error message. The routine declaration will also cause an error to be reported.

ARITHOPNDREQ, Arithmetic operand(s) required

Error.

ARRCNTPCK, Array cannot be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be unpacked.

ARRHAVSIZ, "routine name" requires that ARRAY component have compile-time known size

Error: You cannot use the PACK and UNPACK procedures to pack or unpack one multidimensional conformant array into another. The component type of the dimension being copied must have a compile-time known size; that is, it must have some type other than a conformant schema.

ARRMSTPCK, Array must be PACKED

Error: At least one parameter to the PACK or UNPACK procedure must be of type PACKED.

ARRNOTSTR, Array type is not a string type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. The only legal array, therefore, is PACKED ARRAY [1..n] OF CHAR.

ASYREQASY, ASYNCHRONOUS "calling routine" requires that "called routine" also be ASYNCHRONOUS

Warning.

ASYREQVOL, ASYNCHRONOUS “routine name” requires that “variable name” be VOLATILE

Warning: A variable referred to in a nested asynchronous routine must have the VOLATILE attribute.

ATINTUNS, AT address must be an integer value

Error.

ATREXTERN, “attribute name” attribute allowed only on external routines

Error: The LIST and CLASS_S attributes can be specified only with the declarations of external routines.

ATTRCONCMDLNE, Attribute contradicts command line qualifier

Error: The double-precision attribute specified contradicts the `/FLOAT`, `/G_FLOATING`, or `/NOG_FLOATING` qualifier specified on the compile command line.

ATTRCONFLICT, Attribute conflict: “attribute name”

Information: This message can appear as additional information on other error messages.

ATTRONTYP, Descriptor class attribute not allowed on this type

Error: The use of the descriptor class attribute on the variable, parameter, or routine conflicts with the requirements of the object's type.

AUTOGTRMAXINT, Allocation of “variable name” causes automatic storage to exceed MAXINT bits

Error: The VSI Pascal implementation restricts automatic storage to a size of 2,147,483,647 bits.

AUTOMAX, Unable to quadword align automatic variables, using long alignment

Information.

BADANAORG, Analysis data file “file name” is not on a random access device

Fatal.

BADENVORG, Environment file “file name” is not on a random access device

Fatal.

BADSETCMP, < and > not permitted in set comparisons

Error.

BINOCTHEX, Expecting BIN, OCT, or HEX

Error: You must supply BIN, OCT, or HEX as a variable modifier when reading the variable on a nondecimal basis.

BLKNOTFND, “routine” block “routine name” declared FORWARD in “block name” is missing

Error.

BLKTOODEEP, Routine blocks nested too deeply

Error: You cannot nest more than 31 routine blocks.

BNDACTDIFF, Actual's array bounds differ from those of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible.

BNDCNFRUN, Bounds of conformant ARRAY “array name” not known until run-time

Error: You cannot use the UPPER and LOWER functions on a dynamic array parameter in a compile-time constant expression.

BNDSUBORD, Bound expressions in a subrange type must be ordinal

Error: The expressions that designate the upper and lower limits of a subrange must be of an ordinal type.

BOOLOPREQ, BOOLEAN operand(s) required

Error: The operation being performed requires operands of type BOOLEAN. Such operations include the AND, OR, and NOT operators and the SET_INTERLOCKED and CLEAR_INTERLOCKED functions.

BOOSETREQ, BOOLEAN or SET operand(s) required

Error.

BYTEALIGN, Type larger than 32 bits can be positioned only on a byte boundary

Error: See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for information on the types that are allocated more than 32 bits.

CALLFUNC, Function “function name” called as procedure, function value discarded

Warning.

CARCONMNGLS, CARRIAGE_CONTROL parameter is meaningless given file's type

Warning: The carriage-control parameter is usually meaningful only for files of type TEXT and VARYING OF CHAR.

CASLABEXPR, Case label and case selector expressions are not compatible

Error: All case labels in a CASE statement must be compatible with the expression specified as the case selector.

CASORDRELPTR, Compile-time cast allowed only between ordinal, real, and pointer types

CASSELORD, Case selector expression must be an ordinal type

Error:

CASSRCSIZ, Source type of a cast must have a size known at compile-time

CASTARSIZ, Target type of a cast must have a size known at compile-time

Error: A variable being cast by the type cast operator cannot be a conformant array or a conformant VARYING parameter. An expression being cast cannot be a conformant array parameter, a conformant VARYING parameter, or a VARYING OF CHAR expression. The target type of the cast cannot be VARYING OF CHAR.

CDDABORT, %DICTIONARY processing of CDD record definition aborted

Error: The VSI Pascal compiler is unable to process the CDD record description. See the accompanying CDD messages for more information.

CDDBADDIR, %DICTIONARY directive not allowed in deepest %INCLUDE, ignored

Error: A program cannot use the %DICTIONARY directive in the fifth nested %INCLUDE level. The compiler ignores all %DICTIONARY directives in the fifth nested %INCLUDE level.

CDDBADPTR, invalid pointer was specified in CDD record description

Warning: The CDD pointer data type refers to a CDD path name that cannot be extracted, and is replaced by ^INTEGER.

CDDBIT, Ignoring bit field in CDD record description

Information: The VSI Pascal compiler cannot translate a CDD bit data type that is not aligned on a byte boundary and whose size is greater than 32 bits.

CDDBLNKZERO, Ignoring blank when zero attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD BLANK WHEN ZERO clause.

CDDCOLMAJOR, CDD description specifies a column-major array

Error: The VSI Pascal compiler supports only row-major arrays. Change the CDD description to specify a row-major array.

CDDDEPITEM, Ignoring depends item attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD DEPENDING ON ITEM attribute.

CDDDFLOAT, D_floating CDD datatype was specified when compiling with G_FLOATING

Warning: The CDD record description contains a D_floating data type while compiling with G_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDFLDVAR, CDD record description contains field(s) after CDD variant clause

Error: The CDD record description contains fields after the CDD variant clause. Because VSI Pascal translates a CDD variant clause into a Pascal variant clause, and a Pascal variant clause must be the last field in a record type definition, the fields following the CDD variant clause are illegal.

CDDGFLOAT, G_floating CDD datatype was specified when compiling with NOG_FLOATING

Warning: The CDD record description contains a G_floating data type while compiling with D_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDILLARR, Aligned array elements can not be represented, replacing with [BIT(n)] RECORD END

Information: The VSI Pascal compiler does not support CDD record descriptions that specify an array whose array elements are aligned on a boundary greater than the size needed to represent the data type. It is replaced with [BIT(n)] RECORD END, where n is the appropriate length in bits.

CDDINITVAL, Ignoring specified initial value specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD INITIAL VALUE clause.

CDDMINOCC, Ignoring minimum occurs attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD MINIMUM OCCURS attribute.

CDDONLYTYP, %DICTIONARY may only appear in a TYPE definition part

Error: The %DICTIONARY directive is allowed only in the TYPE section of a program.

CDDRGHTJUST, Ignoring right justified attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD JUSTIFIED RIGHT clause.

CDDSCALE, Ignoring scaled attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD scaled data types.

CDDSRCTYPE, Ignoring source type attribute specified in CDD record description

Information: The VSI Pascal compiler does not support the CDD source type attribute.

CDDTAGDEEP, CDD description nested variants too deep

Error: A CDD record description may not include more than 15 levels of CDD variants. The compiler ignores variants beyond the fifteenth level.

CDDTAGVAR, Ignoring tag variable and any tag values specified in CDD record description

Information: The VSI Pascal compiler does not fully support the CDD VARIANTS OF field description statement. The specified tag variable and any tag values are ignored.

CDDTOODEEP, CDD description nested too deep

Error: Attributes for the CDD record description exceed the implementation's limit for record complexity. Modify the CDD description to reduce the level of nesting in the record description.

CDDTRUNCREF, Reference string which exceeds 255 characters has been truncated

Information: The VSI Pascal compiler does not support reference strings greater than 255 characters.

CDDUNSTYP, Unsupported CDD datatype “standard data type name”

Information: The CDD record description for an item has attempted to use a data type that is not supported by VSI Pascal. The VSI Pascal compiler makes the data type accessible by declaring it as [BYTE(n)] RECORD END where n is the appropriate length in bytes. Change the data type to one that is supported by VSI Pascal or manipulate the contents of the field by passing it to external routines as variables or by using the VSI Pascal type casting capabilities to perform an assignment.

CLSCNFVAL, CLASS_S is only valid with conformant strings

Error: When the CLASS_S attribute is used in the declaration of an internal routine, it can be used only on a conformant PACKED ARRAY OF CHAR. The conformant variable must also be passed by value semantics.

CLSNOTALLW, “descriptor class name” not allowed on a parameter of this type

Error: Descriptor class attributes are not allowed on formal parameters defined with either an immediate or a reference passing mechanism.

CMTBEFEOF, Comment not terminated before end of input

Error.

CNFCANTCNF, Component of PACKED conformant parameter cannot be conformant

Error.

CNFREQNCA, Conformants of this parameter type require CLASS_NCA

Error: The conformant parameter cannot be described with the default CLASS_A descriptor. Add the CLASS_NCA attribute to the parameter declaration.

CNSTRNOTALL, Nonstandard constructors are not allowed on nonstatic types

Error: You can write constructors for nonstatic types using the standard style of constructor.

CNSTRONZERO, Record constructors only allow OTHERWISE ZERO

Error.

CNTBEARRCMP, Not allowed on an array component

CNTBEARRIDX, Not allowed on an array index

CNTBECAST, Not allowed on a cast

CNTBECNFCMP, Not allowed on a conformant array component

CNTBECNFIDX, Not allowed on a conformant array index

CNTBECNFVRY, Not allowed on a conformant varying component

CNTBECOMP, Not allowed on a compilation unit

CNTBECONST, Not allowed on a CONST definition part

CNTBEDEFDECL, Not allowed on any definition or declaration part

CNTBEDESPARM, Not allowed on a %DESCR foreign mechanism parameter

CNTBEEXESEC, Not allowed on an executable section

CNTBEFILCMP, Not allowed on a file component

CNTBEFORMAL, Not allowed on a formal discriminant

CNTBEFUNC, Not allowed on a function result

CNTBEIMMPARM, Not allowed on a parameter passed by an immediate passing mechanism

CNTBELABEL, Not allowed on a LABEL declaration part

CNTBEPCKCNF, Not allowed on a PACKED conformant array component

CNTBEPTRBAS, Not allowed on a pointer base

CNTBERECFLD, Not allowed on a record field

CNTBEREFPARM, Not allowed on a parameter passed by a reference passing mechanism

CNTBERTNDECL, Not allowed on a routine declaration

CNTBERTNPARM, Not allowed on a routine parameter

CNTBESCHEMA, Not allowed on a nonstatic type

CNTBESETRNG, Not allowed on a set range

CNTBESTDPARM, Not allowed on a %STDESCR foreign mechanism parameter

CNTBETAGFLD, Not allowed on a variant tag field

CNTBETAGTYP, Not allowed on a variant tag type

CNTBETO, Not allowed on TO BEGIN/END DO

CNTBETYPDEF, Not allowed on a type definition

CNTBETYPE, Not allowed on a TYPE definition part

CNTBEVALPARM, Not allowed on a value parameter

CNTBEVALUE, Not allowed on a VALUE initialization part

CNTBEVALVAR, Not allowed on a VALUE variable

CNTBEVAR, Not allowed on a VAR declaration part

CNTBEVARBLE, Not allowed on a variable

CNTBEVARPARM, Not allowed on a VAR parameter

CNTBEVRYCMP, Not allowed on a varying component

Information: These messages can appear as additional information on other error messages.

COMCONFLICT, COMMON “block name” conflicts with another COMMON or PSECT of same name

Error: You can allocate only one variable in a particular common block, and the name of the common block cannot be the same as the names of other common blocks or program sections used by your program.

COMNOTALN, Component is not optimally aligned

Information: The component indicated is not well aligned and accesses to it will be inefficient.

COMNOTSIZ, Component is not optimally sized

Information: The component indicated is not well sized and accesses to it will be inefficient.

COMNOTALNSIZE, Component is not optimally aligned and sized

Information: The component indicated is not naturally aligned and sized, accesses to it will be inefficient.

COMNOTPOS, Fixed size field positioned after a run-time sized field is not optimal

Information: Much better code can be generated for indicated component if it precedes all of the run-time sized fields.

CONTXTIGN, Text following constant definition ignored

Warning: When defining constants with the `/CONSTANT` DCL qualifier, any text appearing after a valid constant definition is ignored.

CPPFILERR, Unable to open included file

Error.

CRETIMMOD, Creation time for module "module name" in environment "environment file name" differs from creation time in previous environments

Warning: Two or more PEN files referred to a module, but the PEN files did not agree on the creation date/time for the module. This can occur if you recompile a module but do not recompile all the modules that inherited its PEN file.

CSTRBADTYP, Constructor: only ARRAY, RECORD, or SET type

CSTRCOMISS, Constructor: component(s) missing

CSTRNOVRNT, Constructor: no matching variant

CSTRREFAARR, Repetition factor allowed only in ARRAY constructors

CSTRREFAINT, Repetition factor must be integer

CSTRREFALRG, Repetition factor too large

CSTRREFANEG, Repetition factor cannot be negative

CSTRTOOMANY, Constructor: too many components

Error: You can write constructors only for data items of an ARRAY type. You must specify one and only one value in the constructor for each component of the type. In an array constructor, you cannot use a negative integer value as a repetition factor to specify values for consecutive components.

CSTRREFAINT, Repetition factor must be an integer

Error.

CTESTRSIZ, Compile-time strings must be less than 8192 characters

Error.

CTGARRDESC, Contiguous array descriptor cannot describe size/alignment properties

Information: Conformant array parameters, dynamic array parameters, and %DESCR array parameters all use the contiguous array descriptor mechanism in the *VSI OpenVMS Calling Standard*. Size and alignment attributes are prohibited on such arrays, as these attributes can create noncontiguous allocation. This message can appear as additional information in other error messages.

DEBUGOPT, /NOOPTIMIZE is recommended with /DEBUG

Information: Unexpected results may be seen when debugging an optimized program. To prevent conflicts between optimization and debugging, you should compile your program with /NOOPTIMIZE until it is thoroughly debugged. Then you can recompile the program with optimization enabled to produce more efficient code.

DECLORDER, Declarations are out of order

Error: The TO BEGIN DO and TO END DO declarations in a module must appear at the end of the module and may not be reordered.

DEFRTNPARM, Default parameter syntax not allowed on routine parameters

DEFVARPARM, Default parameter syntax not allowed on VAR parameters

Error.

DESCOMABORT, Further processing of /DESIGN=COMMENTS has been aborted

Error: An error has occurred that prohibits further comment processing.

DESCOMERR, An error has occurred while processing design information

Error.

DESCOMSEVERR, An internal error has occurred while processing /DESIGN=COMMENTS - please submit an SPR

Error: A fatal error has occurred during comment processing. Please submit a problem report including sufficient information to reproduce the program, including the version numbers of the Language-Sensitive Editor/Source Code Analyzer and the VSI Pascal compiler.

DESCTYPCON, Descriptor class / type conflict

Error: The descriptor class for parameter passing conflicts with the parameter's type. Refer to Section 5.3.3 of the *VSI Pascal User Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-user-manual/] for legal descriptor class/type combinations.

DESIGNTOOOLD, The comment processing routines are too old for the compiler

Error: The support routines for the `/DESIGN=COMMENT` qualifier are obsolete. Contact your system manager.

DIRCONVISIB, Directive contradicts visibility attribute

Error: The EXTERN, EXTERNAL, and FORTRAN directives conflict directly with the LOCAL and GLOBAL attributes.

DIREXPECT, No matching directive for the %IF directive

Error: A %IF directive must contain a %THEN clause and be terminated by %ENDIF.

DIRUNEXP

Error: Conditional compilation directives other than %IF are only valid after the parts of a %IF directive.

DISCLIMIT, Limit of 255 discriminants exceeded

Error.

DISNOTORD, Discriminant type must be an ordinal type

Error: The formal discriminant in a schema type definition must be an ordinal type.

DONTPACKVAR, "routine name" is illegal, variable can never appear in a packed context

Error: You cannot call the BITSIZE and BITNEXT functions for conformant parameters.

DUPLALIGN, Alignment already specified

DUPLALLOC, Allocation already specified

DUPLATTR, Attribute already specified

DUPLCLASS, Descriptor class already specified

DUPLDOUBLE, Double precision already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPLFIN, TO END DO already specified, DUPLINIT, TO BEGIN DO already specified

Error: Only one TO BEGIN DO and one TO END DO section can appear in the same module.

DUPLGBLNAM, Duplicated global name

Warning: The GLOBAL attribute cannot appear on more than one variable or routine with the same name.

DUPLMECH, Passing mechanism already specified

DUPLOPT, Optimization already specified

DUPLSIZE, Size already specified

DUPLVISIB, Visibility already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

DUPTYPALI, Alignment already specified by type identifier "type name"

DUPTYPALL, Allocation already specified by type identifier "type name"

DUPTYPATR, Attribute already specified by type identifier "type name"

DUPTYPDES, Descriptor class already specified by type identifier "type name"

DUPTYPSIZ, Size already specified by type identifier "type name"

DUPTYPVIS, Visibility already specified by the type identifier "type name"

Error: An attribute specified for an object was already specified in the definition of the object's type.

ELEOUTRNG, Element out of range

Error: A value specified in a set constructor used as a compile-time constant expression does not fall within the subrange defined as the set's base type.

EMPTYCASE, Empty case body

Error: You failed to specify any case labels and corresponding statements in the body of a CASE statement.

ENVERROR, Environment resulted from a compilation with Errors

Error: When a program inherits an environment file that compiled with errors, unexpected results may occur during the program's compilation. The environment file inherited by the program compiled with errors. Unexpected results may occur in the program now being compiled.

ENVFATAL, Environment resulted from a compilation with Fatal Errors

Error: The environment file inherited by the program compiled with fatal errors. Unexpected results may occur in the program now being compiled.

ENVOLDVER, Environment was created by a VAX Pascal V2 compiler, please recompile

Warning: The environment file inherited by the program was created by a VAX Pascal Version 2.0 compiler. You should regenerate the environment file with a newer version of the compiler.

ENVWARN, Environment resulted from a compilation with Warnings

Warning: The environment file inherited by the program compiled with warnings. Unexpected results may occur in the program now being compiled.

ENVWRGCMP, Environment identifier was compiled by an VSI Pascal for platform compiler

Fatal.

ERREALCNST, Error in real constant: digit expected

Error.

ERRNONPOS, ERROR parameter can be specified only with nonpositional syntax

Error.

ERROR, %ERROR

Error: This message is generated by the %ERROR directive.

ERRORLIMIT, Error Limit = “current error limit”, source analysis terminated

Fatal: The error limit specified for the program's compilation was exceeded; the compiler was unable to continue processing the program. By default, the error limit is set at 30, but you can use the error limit switch at compile time to change it.

ESTBASYNCH, ESTABLISH requires that “routine name” be ASYNCHRONOUS

Warning.

EXPLCONVREQ, Explicit conversion to lower type required

Error: An expression of a higher-ranked type cannot be assigned to a variable of a lower-ranked type; you must first convert the higher-ranked expression by using DBLE, SNGL, TRUNC, ROUND, UTRUNC, or UROUND, as appropriate.

EXPNOTRES, Expression does not contribute to result

Information: The optimizer has determined that part of the expression does not affect the result of the expression and it will not evaluate that part of the expression.

EXPR2ONVAL, Expression is allowed only on real, integer, or unsigned values

Error: The second expression (and preceding colon) are allowed only if the value being written is of a real, integer, or unsigned type.

EXPRARITH, Expression must be arithmetic

Error: An expression whose type is not arithmetic cannot be assigned to a variable of a real type.

EXPRARRIDX, Expression is incompatible with unpacked array's index type

Error: The index type of the unpacked array is not compatible with the index type of either the PACK or UNPACK procedure it was passed to.

EXPRCOMTAG, Expression is not compatible with tag type

Error: A case label specified for a NEW, DISPOSE, or SIZE routine must be assignment compatible with the tag type of the variant.

EXPRINFUNC, Expression allowed only in FUNCTION

Error.

EXPRNOTSET, Expression is not a SET type

Error: The compiler encountered an expression of some type other than SET in a call to the CARD function.

EXTRNALLOC, Allocation attribute conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

EXTRNAMDIFF, External names are different

Information: This message can appear as additional information on other error messages.

EXTRNCFLCT, "PSECT or FORWARD" conflicts with EXTERNAL visibility

Error: The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

FILEVALASS, FILE evaluation / assignment is not allowed

Error: You cannot attempt to evaluate a file variable or assign values to it.

FILHASSCH, FILE component may not contain nonstatic types or discriminant identifiers

Error: VSI Pascal restricts components of files to those with compile-time size.

FILOPNDREQ, FILE operand required

Error: The EOF, EOLN, and UFB functions require parameters of file types.

FILVARFIL, FILE_VARIABLE parameter must be of a FILE type

Error: The file variable parameter to the OPEN and CLOSE procedures must denote a file variable.

FLDIVPOS, Field "field name" is illegally positioned

Error: A POS attribute attempted to position a record field before the end of the previous field in the declaration.

FLDNOTKNOWN, Unknown record field

Error.

FLDONLYTXT, Field width allowed only when writing to a TEXT file

Error.

FLDRADINT, Field width or radix expression must be of type INTEGER

Error: The field-width or radix expression in a WRITE, WRITELN, or WRITEV routine must be of type INTEGER.

FORACTORD, FOR loop control variable must be of an ordinal type

FORACTVAR, FOR loop control must be a true variable

Error: The control variable of a FOR statement must be a simple variable of an ordinal type and must be declared in a VAR section. For example, it cannot be a field in a record that was specified by a WITH statement, or a function identifier.

FLDWDTHINT, Field-width expression must be of type integer

Error.

FORCTLVAR, "variable name" is a FOR control variable

Warning: The control variable of a FOR statement cannot be assigned a value; used as a parameter to the ADDRESS function; passed as a writable VAR, %REF, %DESCR, or %STDESCR parameter; used as the control variable of a nested FOR statement; or written into by a READ, READLN, or READV procedure.

FORINEXPR, Expression is incompatible with FOR loop control variable

Error: The type of the initial or final value specified in a FOR statement is variable.

FRMLPRMDESC, Formal parameters use different descriptor formats

FRMLPRMINCMP, Formal routine parameters are not compatible

FRMLPRMNAM, Formal parameters have different names

FRMLPRMSIZ, Formal parameters have different size attributes

FRMLPRMTYP, Formal parameters have different types

Information: These messages can appear as additional information on other error messages.

FRSTPRMSTR, READV requires first parameter to be a string expression

Error: You must specify at least two parameters for the READV procedure—a character-string expression and a variable into which new values will be read.

FRSTPRMVARY, WRITEV requires first parameter to be a variable of type VARYING

Error.

FTRNOTHER, Feature not supported io this context

Error.

FTRNOTPOR, Feature not supported on platform(s)

Information.

FTRNOTSUP, Feature not supported on this platform

Error.

FUNCTRESTYP, Routine must be declared as FUNCTION to specify a result type

Error: You cannot specify a result type on a PROCEDURE declaration.

FUNRESTYP, Function result types are different

Information: This message can appear as additional information on other error messages.

FWDREPATRLST, Declared FORWARD; repetition of attribute list not allowed

FWDREPPRMLST, Declared FORWARD; repetition of formal parameter list not allowed

FWDREPRESTYP, Declared FORWARD; repetition of result type not allowed

Error: If the heading of a routine has the FORWARD directive, the declaration of the routine body cannot repeat the formal parameter list, the result type (applies only if the routine is a function), or any attribute lists that appeared in the heading.

FWDWASFUNC, FORWARD declaration was FUNCTION

FWDWASPROC, FORWARD declaration was PROCEDURE

Error.

GOTONOTALL, GOTO not allowed to jump into a structured statement

Warning: Jumping into a structured statement may yield incorrect behavior and/or additional compile-time errors.

GOTSZOVFL, GOT table overflow for module "name"

Error: The GOT (Global Offset Table) for the module is too large. Break up the module into multiple modules.

GTR32BITS, “routine name” cannot accept parameters larger than 32 bits

Error: DEC and UDEC cannot translate objects larger than 32 bits into their textual equivalent.

HIDATOUTER, HIDDEN legal only on definitions and declarations at outermost level

Error: When an environment file is being generated, it is possible to prevent information concerning a declaration from being included in the environment file by using the HIDDEN attribute. However, because an environment file consists only of declarations and definitions at the outermost level of a compilation unit, the HIDDEN attribute is legal only on these definitions and declarations.

IDENTGTR31, Identifier longer than 31 characters exceeds capacity of compiler

Warning.

IDNOTLAB, Identifier “symbol name” not declared as a label

Error.

IDXNOTCOMPAT, Index type is not compatible with declaration

Error: The type of an index expression is not assignment compatible with the index type specified in the array's type definition.

IDXREQDKEY, Creating INDEXED organization requires dense keys

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's alternate keys must be dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified for the file's component type.

IDXREQKEY0, Creating INDEXED organization requires FILE OF RECORD with at least KEY(0)

Warning: When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's component type must be a record for which a primary key, designated by the [KEY(0)] attribute, is defined.

ILLINISCH, Illegal initialization of variable of nonstatic type

Error: Nonstatic variables, such as those created from schema types, cannot be initialized in the VALUE declaration part. To initialize these variables, you must use the initial state feature.

IMMEDBNDROU, Immediate passing mechanism may not be used on bound routine “routine name”

Warning: You cannot prefix a formal or an actual routine parameter with the immediate passing mechanism unless the routine was declared with the UNBOUND attribute.

IMMEDUNBND, Routines passed by immediate passing mechanism must be UNBOUND

Warning: A formal routine parameter that has the immediate passing mechanism must also have the UNBOUND attribute.

IMMGTR32, Immediate passing mechanism not allowed on values larger than 32 or 64 bits

Error: See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information on the types that are allocated more than 32 or 64 bits.

IMMHAVSIZ, Type passed by immediate passing mechanism must have compile-time known size

Error: You cannot specify an immediate passing mechanism for a conformant parameter or a formal parameter of type VARYING OF CHAR.

INCMPBASE, Incompatible with SET base type

Error: If no type identifier denotes the base type of a set constructor, the first element of the constructor determines the base type. The type of all subsequent elements specified in the constructor must be compatible with the type of the first.

INCMPFLDS, Record fields are not the same type

Error.

INCMPOPND, Incompatible operand(s)

Error: The types of one or more operands in an expression are not compatible with the operation being performed.

INCMPPARM, Incompatible “routine” parameter

Error: An actual routine parameter is incompatible with the corresponding formal parameter.

INCMPTAGTYP, Incompatible variant tag types

Error: This message can appear as additional information on other error messages.

INCTOODEEP, %INCLUDE directives nested too deeply, ignored

Error: A program cannot include more than five levels of files with the %INCLUDE directive. The compiler ignores %INCLUDE files beyond the fifth level.

INDNOTORD, Index type must be an ordinal type

Error: The index type of an array must be an ordinal type.

INFO, %INFO

Information: This message is generated by the %INFO directive.

INITNOEXT, INITIALIZE routine may not be EXTERNAL

INITNOFRML, INITIALIZE routine must have no formal parameter list

Error.

INITSYNVAR, Illegal initialization syntax – Use VALUE

Error.

INPNOTDECL, INPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

INSTNEWLSE, Please install a new version of LSE

Error: The version of the Language-Sensitive Editor/Source Code Analyzer on your system is too old for the compiler. Contact your system manager.

INVCASERNG, Invalid range in case label list

Error.

INVEVAL, Array or Record evaluation not allowed

Error.

INVQUAVAL, Value for optimizer level is out of range. Default value will be used.

Error.

IVATTOPT, Unrecognized option for attribute

Explanation: You attempted to specify an invalid option for one of the following attributes:

- CHECK (Warning)
- FLOAT (Warning)
- KEY (Error)
- OPTIMIZE (Warning)

IVATTR, Unrecognized attribute

Error.

IVAUTOMOD, AUTOMATIC variable is illegal at the outermost level of a MODULE

Error: You cannot specify the AUTOMATIC attribute for a variable declared at module level.

IVCHKOPT, Unrecognized CHECK option

Warning.

IVCOMBFLOAT, Illegal combination of D_floating and G_floating

Error: You cannot combine D_floating and G_floating numbers in a binary operation.

IVDIRECTIVE, Unrecognized directive

Error: The directive following a procedure or function heading is not one of those recognized by the VSI Pascal compiler.

IVENVIRON, Environment “environment name” has illegal format, source analysis terminated

Fatal: The environment file inherited by the program has an illegal format; compilation is immediately aborted. However, a listing will still be produced if one was being generated.

IVFUNC, Invalid use of function "function name"

IVFUNCALL, Invalid use of function call

IVFUNCID, Invalid use of function identifier

Error: These messages result from illegal attempts to assign values or otherwise refer to the components of the function result (if its type is structured), use the type cast operator on a function identifier or its result, or deallocate the storage reserved for the function result (if its type is a pointer).

IVKEYOPT, Unrecognized KEY option

Error.

IVKEYVAL, FINDK KEY_VALUE cannot be an array (other than PACKED ARRAY [1..n] OF CHAR)

Error.

IVKEYWORD, Missing or unrecognized keyword

Error: The compiler failed to find an identifier where it expected one in a call to the OPEN or CLOSE procedure, or it found an identifier that was not legal in this position in the parameter list.

IVMATCHTYP, Invalid MATCH_TYPE parameter to FINDK

Error.

IVOPTMOPT, Unrecognized OPTIMIZE option

Warning.

IVOTHVRNT, Illegal use of OTHERWISE within CASE variant

Error: The VSI Pascal extension of using OTHERWISE in a record constructor is only defined at the outer level of a record.

IVQUALFILE, Illegal switch “switch name” on file specification

Warning: Only the `/LIST` and `/NOLIST` qualifiers are allowed on the file specification of a %INCLUDE directive.

IVQUOCHAR, Illegal nonprinting character (ASCII “nnn”) within quotes

Warning: The only nonprinting characters allowed in a quoted string are the space and tab; the use of other nonprinting characters in a string causes this warning. To include nonprinting characters in a string, you should use the extended string syntax described in the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/].

IVRADIX, Invalid radix was specified in the extended number

Error.

IVRADIXDGIT, Illegal digit in binary, octal, or hexadecimal constant

Error.

IVREDECL, Illegal redeclaration gives "symbol name" multiple meanings in "scope name"

IVREDECLREC, Illegal redeclaration gives "symbol name" multiple meanings in this record

IVREDEF, Illegal redefinition gives "symbol name" multiple meanings in "scope name"

Warning: When an identifier is used in any given block, it must have the same meaning wherever it appears in the block.

IVUSEALIGN, Invalid use of alignment attribute

IVUSEALLOC, Invalid use of allocation attribute

Error.

IVUSEATTR, Invalid use of “attribute name” attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEATTRLST, Invalid use of an attribute list

Error.

IVUSEBNDID, Illegal use of bound identifier “identifier name”

Error: An identifier that represents one bound of a conformant schema was used where a variable was expected, such as in an assignment statement or in a formal VAR parameter section. The restrictions on the use of a bound identifier are identical to those on a constant identifier.

IVUSEDES, Invalid use of descriptor class attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEFNID, Illegal use of function identifier “identifier name”

Error: Two examples of illegal uses are the assignment of values to the components of the function result (if its type is structured) and the passing of the function identifier as a VAR parameter.

IVUSEPOI, Illegal use of type POINTER or UNIV_PTR

Error: Values of type POINTER and UNIV_PTR can not be dereferenced with the ^ operator or used with the built-in routines NEW and DISPOSE.

IVUSESIZ, Invalid use of size attribute

Error: The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEVISIB, invalid use of visibility attribute

Error: The use of a visibility attribute conflicts with the requirements of the object's type.

KEYINTRNG, KEY number must be an integer value in range 0..254

Error: The key number specified by a KEY attribute must fall in the integer subrange 0..254.

KEYNOTALIGN, KEY “key number” field “field name” at bit position “bit position” is unaligned

KEYORDSTR, KEY allowed only on ordinal and fixed-length string fields

KEYPCKREC, KEY field in PACKED RECORD must have an alignment attribute

KEYREDECL, Key number “key number” is multiply defined

KEYSIZ1_2_4, Size of an ordinal key must be 1, 2 or 4 bytes

KEYSIZ2_4, Size of a signed integer key must be 2 or 4 bytes

KEYSIZSTR, Size of a string key cannot exceed 255 bytes

KEYUNALIGN, KEY field cannot be UNALIGNED

Error.

LABDECIMAL, Label number must be expressed in decimal radix

Error.

LABINCTAG, Variant case label's type is incompatible with tag type

Error: The type of a constant specified as a case label of a variant record is not assignment compatible with the type of the tag field.

LABNOTFND, No definition of label “label name” in statement part of “block name”

Error: A label that you declared in a LABEL section does not prefix a statement in the executable section.

LABREDECL, Redefinition of label “label name” in “block name”

Error: A label cannot prefix more than one statement in the same block.

LABRNGTAG, Variant case label does not fall within range of tag type

Error: A constant specified as a case label of a variant record is not within the range defined for the type of the tag field.

LABTOOBIG, Label “label number” is greater than MAXINT

Error.

LABUNDECL, Undeclared label “label name”

Error: VSI Pascal requires that you declare all labels in a LABEL declaration section before you use them in the executable section.

LABUNSATDECL, Unsatisfied declaration of label “label name” is not local to “block name”

Error: A label that prefixes a statement in a nested block was declared in an enclosing block.

LIBESTAB, LIB$ESTABLISH is incompatible with VSI Pascal; use predeclared procedure ESTABLISH

Warning: VSI Pascal establishes its own condition handler for processing Pascal-specific run-time signals. Calling LIB$ESTABLISH directly replaces the handler supplied by the compiler with a user-written handler; the probable result is improper handling of run-time signals. You should use Pascal's predeclared ESTABLISH procedure to establish user-written condition handlers.

LISTONEND, LIST attribute allowed only on final formal parameter

Error.

LISTUSEARG, Formal parameter has LIST attribute, use predeclared function ARGUMENT

Error: A formal parameter with the LIST attribute cannot be directly referenced. You should use the predeclared function ARGUMENT to reference the actual parameters corresponding to the formal parameter.

LNETOOLNG, Line too long, is truncated to 255 characters

Error: A source line cannot exceed 255 characters. If it does, the compiler disregards the remainder of the line.

LOWGTRHIGH, Low-bound exceeds high-bound

Error: The definition of the flagged subrange type is illegal because the value specified for the lower limit exceeds that for the upper limit.

MAXLENINT, Max-length must be a value of type integer

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXLENRNG, Max-length must be in range 1..65535

Error: The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXNUMENV, Maximum number of environments exceeded

Fatal: More than 512 environment files were used in the compilation.

MECHEXTERN, Foreign mechanism specifier allowed only on external routines

Error.

MISSINGEND, No matching END, expected near line "line number"

Information: The compiler expected an END statement at a location where none was found. Compilation proceeds as though the END statement were correctly located.

MODINIT26, Module name limited to 26 characters when initialization required

Error: When a module contains schema types, discriminated schema types, variables of discriminated schema types, or a TO BEGIN DO statement clause, the module name is limited to 26 characters.

MODOFNEGNUM, MOD of a negative modulus has no mathematical definition

Error: In the MOD operation A MOD B, the operand B must have a positive integer value. This message is issued only when the MOD operation occurs in a compile-time constant expression.

MSTBEARRAY, Type must be ARRAY

Error.

MSTBEARRVRY, Type must be ARRAY or VARYING

Error: You cannot use the syntax [index] to refer to an object that is not of type ARRAY or VARYING OF CHAR.

MSTBEBOOL, Control expression must be of type BOOLEAN

Error: The IF, REPEAT, and WHILE statements require a Boolean control expression.

MSTBEDEREF, Must be dereferenced

Information.

MSTBEDISCR, Schema type must be discriminated

Error: An undiscriminated schema type is not allowed everywhere that a regular type name is allowed.

MSTBEORDSETARR, Type must be ordinal, SET, or ARRAY

Error.

MSTBEREC, Type must be RECORD

Error.

MSTBERECVRY, Type must be RECORD or VARYING

Error: You cannot use the syntax “Variable.Identifier” to refer to an object that is not of type RECORD or VARYING OF CHAR.

MSTBESTAT, Cannot initialize non-STATIC variables

Error: You cannot initialize variables declared without the STATIC attribute in nested blocks, nor can you initialize program-level variables whose attributes give them some allocation other than static.

MSTBETEXT, “I/O routine” requires FILE_VARIABLE of type TEXT

Error: The READLN and WRITELN procedures operate only on text files.

MULTDECL, “symbol name” has multiple conflicting declarations, reason(s):

Error.

NCATOA, Cannot reformat content of actual's CLASS_NCA descriptor as CLASS_A

Error: This message can appear as additional information on other error messages.

NEWQUADAGN, “type name”'s base type is ALIGNED(“nnn”); NEW handles at most ALIGNED(3)

Error: You cannot call the NEW procedure to allocate pointer variables whose base types specify alignment greater than a quadword. To allocate such variables, you must use external routines.

NOACTCOM, No actuals are compatible with schema formal parameter

Information: Undiscriminated schema formal parameters denoting subranges or sets cannot be used as value parameters. In these cases, no actual parameter can ever be compatible with the formal parameter.

NOASSTOFNC, Block does not contain an assignment to function result “function name”

Warning: The block of a function must include a statement that assigns a return value to the function identifier.

NOCONVAL, A constant value was not specified for field “field name”

Error.

NODECLVAR, “symbol name” is not declared in a VAR section of “block name”

Error: You cannot initialize a variable using the VALUE section if the variable was not declared in the same block in which the VALUE section appears.

NODSCREC, No descriptor class for RECORD type

Error: The *VSI OpenVMS Calling Standard* does not define a descriptor format for records; therefore, you cannot specify %DESCR for a parameter of type RECORD.

NODSCRSCH, No descriptor class for schematic types

Error.

NOFLDREC, No field “field name” in RECORD type “type name”

Error: The field specified does not exist in the specified record.

NOFRMINDECL, Declaration of “routine” parameter “routine name” supplied no formal parameter list

Information: You specified actual parameters in a call on a formal routine parameter that was declared with no formal parameters. Although such a call was legal in VAX Pascal Version 1.0, it does not follow the rules of the Pascal standard. You should edit your program to reflect this change.

NOINITEXT, Initialization not allowed on EXTERNAL variables

NOINITINH, Initialization not allowed on inherited variables

Error: You can initialize only those variables whose storage is allocated in this compilation.

NOINITVAR, Cannot initialize "symbol name"—it is not declared as a variable

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

NOLISTATTR, Parameter to this predeclared function must have LIST attribute

Error: ARGUMENT and ARGUMENT_LIST_LENGTH require their first parameter to be a formal parameter with the LIST attribute.

NONATOMIC, Unable to generate code for atomic access

Warning: Due to poor alignment, the code generator is unable to generate an atomic code sequence to read or write the volatile object.

NONGRNACC, Unable to generate code for requested granularity

Warning: Due to poor alignment, the code generator is unable to generate a code sequence for the granularity requested.

NOREPRE, No textual representation for values of this type

Error: You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. Such types are RECORD, ARRAY (other than PACKED ARRAY [1..n] OF CHAR), SET, and pointer.

NOTAFUNC, "symbol name" is not declared as a "routine."

Error: An identifier followed by a left parenthesis, a semicolon, or one of the reserved words END, UNTIL, and ELSE is interpreted as a call to a routine with no parameters. This message is issued if the identifier was not declared as a procedure or function identifier. Note that in the current version, functions can be called with the procedure call statement.

NOTASYNCH, "routine name" is not ASYNCHRONOUS

Information: This message can appear as additional information on other error messages.

NOTATAG, "identifier" is not a tag-identifier

Error: The identifier used with the CASE OF construct in a record constructor must be a tag identifier.

NOTATYPE, “symbol name” is not a type identifier

Error: An identifier that does not represent a type was used in a context where the compiler expected a type identifier.

NOTAVAR, “symbol name” is not declared as a variable

Error: You cannot assign a value to any object other than a variable.

NOTAVARFNID, “symbol name” is not declared as a variable or a function identifier

Error: You cannot assign a value to any object other than a variable or a function identifier.

NOTAVARPARM, “symbol name” is not declared as a variable or parameter

Error.

NOTBEADDR, May not be parameter to ADDRESS

NOTBEARGV, May not be used as a parameter to ARGV\NOTBEASSIGN, May not be assigned

NOTBECALL, May not be called as a FUNCTION

NOTBECAST, May not be type cast

NOTBEDEREF, May not be dereferenced

NOTBEDES, May not be passed by untyped %DESCR

NOTBEEVAL, May not be evaluated

NOTBEFILOP, May not be used in a file operation

NOTBEFLD, May not be field selected

NOTBEFNCPRM, May not be passed as a FUNCTION parameter

NOTBEFORCTL, May not be used as FOR loop variable

NOTBEFORDES, May not be passed as a descriptor foreign parameter

NOTBEFOREF, May not be passed as a reference foreign parameter

NOTBEIADDR, May not be parameter to IADDRESS

NOTBEIDX, May not be indexed

NOTBEIMMED, May not be passed by untyped immediate passing mechanism

NOTBENEW, May not be written into by NEW

NOTBENSTCTL, May not be control variable for an inner FOR loop

NOTBEREAD, May not be written into by READ

NOTBEREF, May not be passed by untyped reference passing mechanism

NOTBERODES, May not be passed as a READONLY descriptor foreign parameter

NOTBEROFOR, May not be passed as a READONLY reference foreign parameter

NOTBEROVAR, May not be passed as a READONLY VAR parameter

NOTBETOUCH, May not be read/modified/written

NOTBEVAR, May not be passed as a VAR parameter

NOTBEWODES, May not be passed as a WRITEONLY descriptor foreign parameter

NOTBEWOFOR, May not be passed as a WRITEONLY reference foreign parameter

NOTBEWOVAR, May not be passed as a WRITEONLY VAR parameter

NOTBEWRTV, May not be parameter to WRITEV

Information: These messages can appear as additional information on other error messages.

NOTBYTOFF, Field “field name” is not aligned on a byte boundary

Error.

NOTDECLROU, “symbol name” is not declared as a “routine”.

NOTINITIAL, “routine name” is not INITIALIZE

Information: These messages can appear as additional information on other error messages.

NOTINRNG, Value does not fall within range of the tag type

Error: The value specified as the case label of a variant record is not a legal value of the tag field's type. This message is also issued if a case label in a call to NEW, DISPOSE, or SIZE falls outside the range of the tag type.

NOTNEWTYP, Schema must define a new type

Error: The type-denoter of a schema definition must define a new type; for example, a subrange, an array, or a record.

NOTXTLIB, No text library was specified at compile time

Error: The specified %INCLUDE module could not be accessed because a text library was not specified on the command line or in the PASCAL$LIBRARY logical name.

NOTSAMTYP, Not the same type

NOTUNBOUND, “routine name” is not UNBOUND

Information: These messages can appear as additional information on other error messages.

NOTSCHEMA, “symbol name” is not a schema type

Error.

NOTVARNAM, Parameter to this predeclared function must be simple variable name

Error: The parameter cannot be indexed, be dereferenced, have a field selected, or be an expression. It must be the name of the entire variable.

NOTVOLATILE, “variable name” is non-VOLATILE

Warning: You should not use the ADDRESS function on a nonvolatile variable or component or on a formal VAR parameter.

NOUNSATDECL, No unsatisfied declaration of label “label name” in “block name”

Error.

NUMFRMLPARM, Different numbers of formal parameters

Information: This message can appear as additional information on other error messages.

NXTACTDIFF, NEXT of actual's component differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size and alignment of the array's inner dimensions.

OLDDECLSYN, Obsolete “routine” parameter declaration syntax

Information: The declaration of a formal routine parameter uses the obsolete VAX Pascal Version 1.0 syntax. You should edit your program to incorporate the current version syntax, which is mandated by the Pascal standard.

OPNDASSCOM, Operands are not assignment compatible

OPNOTINT, Operand(s) must be of type integer

Error.

OPNDNAMCOM, Operands are not name compatible

Error.

ORDOPNDREQ, Ordinal operand(s) required

Error or Warning: This message is at warning level if you try to use INT, ORD, or UINT on a pointer expression. It is at error level if you use PRED or SUCC on an expression whose type is not ordinal.

OUTNOTDECL, OUTPUT not declared in heading

Error: A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

OVRDIVZERO, Overflow or division by zero in compile-time expression

Error.

PACKSTRUCT, “component name” of a PACKED structured type

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

PARMACTDEF, Formal parameter “parameter name” has neither actual nor default

Error: If a formal parameter is not declared with a default, you must pass an actual parameter to it when calling its routine.

PARMCLAMAT, Parameter section classes do not match

Information: This message can appear as additional information on other error messages.

PARMLIMIT, VSI OpenVMS architectural limit of 255 parameters exceeded

Error: You cannot declare a procedure with more than 255 formal parameters. A function whose result type requires that the result be stored in more than 64 bits or whose result type is a character string cannot have more than 254 formal parameters. In a call to a routine declared with the LIST attribute, you also cannot pass more than 255 (or 254) actual parameters.

PARMSECTMAT, Division into parameter sections does not match

Information: This message can appear as additional information on other error messages.

PARSEFAIL, error parsing command line; use `PASCAL` command

Fatal: The VSI Pascal compiler was invoked without using the **`PASCAL`** DCL command.

PARSEFAIL, error parsing command line; using an invalid CLD table

Fatal: The VSI Pascal compiler was invoked with an incorrect or obsolete command-line definition in SYS$LIBRARY:DCLTABLES. Contact your system manager to reinstall SYS$LIBRARY: DCLTABLES.

PASPREILL, Passing predeclared “routine name” is illegal

Error: You cannot use the IADDRESS function on a predeclared routine for which there is no corresponding routine in the run-time library (such as the interlocked functions). In addition, you cannot pass a predeclared routine as a parameter if there is no way to write the predeclared routine's formal parameter list in VSI Pascal. Examples of the latter case are the PRED and SUCC functions and many of the I/O routines.

PASSEXTERN, Passing mechanism allowed only on external routines

Error.

PASSNOTLEG, Passing mechanism not legal for this type

Error.

PCKARRBOO, PACKED ARRAY OF BOOLEAN parameter expected

Error.

PCKUNPCKCON, Packed/unpacked conflict

Information: This message can appear as additional information on other error messages.

PLACEBEFEOLN, Placeholder not terminated before end of line

Error.

PLACEIVCHAR, Illegal nonprinting character (ASCII “decimal representation of character”) within placeholder

Warning.

PLACENODOT, Repetition of pseudocode placeholders not allowed

Error.

PLACESEEN, Placeholder encountered

Error.

PLACEUNMAT, Unmatched placeholder delimiter

Error.

POSAFTNONPOS, Positional parameter cannot follow a nonpositional parameter

Error.

POSALIGNCON, Position / alignment conflict

Error: The bit position specified by the POS attribute does not have the number of low-order bits implied by the specified alignment attribute.

POSINT, POS expression must be a positive integer value

Error.

PRENAMRED, Predeclared name cannot be redefined

Error: A predeclared name may not be redefined when defining constants with the `/CONSTANT` DCL qualifier.

PREREQPRMLST, Passing predeclared "routine name" requires formal to include parameter list

Error: To pass one of the predeclared routines EXPO, ROUND, TRUNC, UNDEFINED, UTRUNC, UROUND, DBLE, SNGL, QUAD, INT, ORD, and UINT as an actual parameter to a routine, you must specify a formal parameter list in the corresponding formal routine parameter.

PRMKWNSIZ, Parameter must have a size known at compile-time

Error: The BIN, HEX, OCT, DEC, and UDEC functions cannot be used on conformant parameters. The SIZE and NEXT functions cannot be used on conformant parameters in compile-time constant expressions.

PROCESSFILE, Compiling file "file name"

Information.

PROCESSRTN, Generating code for routine "routine name"

Information.

PROGSCHENV, PROGRAM with schema may not create environment

Error: A program that declares a schema type cannot have the [ENVIRONMENT] attribute. Schema declarations should be placed in a separate module and inherited by the program.

PROPRMEXT, Declaration of "program parameter name" is EXTERNAL-program parameter files must be locally allocated

PROPRMFIL, A program parameter must be a variable of type FILE

PROPRMINH, Declaration of "program parameter name" is inherited-program parameter files must be locally allocated

PROPRMLEV, Program parameter "program parameter name" is not declared as a variable at the outermost level

Error: Any external file variable (other than INPUT and OUTPUT) that is listed in the program heading must also be declared as a file variable in a VAR section in the program block.

PSECTMAXINT, Allocation of "symbol name" causes PSECT "PSECT name" to exceed MAXINT bits

Error: The VSI Pascal implementation restricts the size of a program section to 2,147,483,647 bits.

PTRCMPEQL, Pointer values may only be compared for equality

Error: The equality (=) and inequality (<>) operators are the only operators allowed for values of a pointer type; all other operators are illegal.

PTREXPRCOM, Pointer expressions are not compatible

Error: The base types of two pointer expressions being compared for equality (=) or inequality (<>) are not structurally compatible.

QUOBEFEOL, Quoted string not terminated before end of line

Error.

QUOSTRING, Quoted string expected

Error: The compiler expects the %DICTIONARY and %INCLUDE directives, and the radix notations for binary (%B), hexadecimal (%X), and octal constants (%O), to be followed by a quoted string of characters.

RADIXTEXT, Radix input requires FILE_VARIABLE of type TEXT

Error: The input radix specifiers (BIN, OCT, and HEX) operate only on text files.

READONLY, "variable name" is READONLY

Warning: You cannot use a read-only variable in any context that would store a new value in the variable. For example, a read-only variable cannot be used in a file operation.

REALCNSTRNG, Real constant out of range

Error: See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for details on the range of real numbers.

REALOPNDREQ, Real (SINGLE, DOUBLE or QUADRUPLE) operand(s) required

Error.

RECHASFILE, Record contains one or more FILE components, POS is illegal

Error.

RECHASTMSTMP, Record contains one or more TIMESTAMP components, POS is illegal

Error.

RECLENINT, RECORD_LENGTH expression must be of type integer

Error: The value of the record length parameter to the OPEN procedure must be an integer.

RECLENMNGLS, RECORD_LENGTH parameter is meaningless given file's type

Warning: The record length parameter is usually relevant only for files of type TEXT and VARYING OF CHAR.

RECMATCHTYP, MATCH_TYPE identifier “NXT or NXTEQL” is recommended instead of “GTR or GEQ”

Information.

REDECL, A declaration of “symbol name” already exists in “block name”

Error: You cannot redeclare an identifier or a label in the same block in which it was declared. Inheriting an environment is equivalent to including all of its declarations at program or module level.

REDECLATTR, “attribute name” already specified

Error: Only one member of a particular attribute class can appear in the same attribute list.

REDECLFLD, Record already contains a field “field name”

Error: The names of the fields in a record must be unique; they cannot be duplicated between variants.

REINITVAR, “variable name” has already been initialized

Error: Variables are the only data items that can be initialized, and they can be initialized only once.

REPCASLAB, Value has already appeared as a label in this CASE statement

Error: You cannot specify the same value more than once as a case label in a CASE statement.

REPFACZERO, Repetition factor cannot be the function ZERO

REQCLAORNCA, Arrays and conformants of this parameter type require either CLASS_A or CLASS_NCA

REQCLS, Scalars and strings of this parameter type require CLASS_S

Error.

REGNATAGN, Operand must be naturally aligned

Error.

REQNOCH, Primary key requires NOCHANGES option

Error.

REQPKDARR, The combination of CLASS_S and %STDESCR requires a PACKED ARRAY OF CHAR structure

Error.

REQREADVAR, READ or READV requires at least one variable to read into

Error: The READ and READV procedures require that you specify at least one variable to be read from a file.

REQWRITELEM, WRITE requires at least one write-list-element

Error: The WRITE procedure requires that you specify at least one item to be written to a file.

RESPTRTYP, Result must be a pointer type

Information.

REVRNTLAB, Value has already appeared as a label in this variant part

Error: You cannot specify the same value more than once as a case label in a variant part of a record.

RTNSTDESCR, Routines cannot be passed using %STDESCR

Error.

SCHCONST, Nonstatic constants are not allowed

Error: Constants cannot be made for nonstatic types since that would yield constants without compile-time size and value.

SCHFLDALN, Field in nonstatic type may not have greater than byte alignment

Error.

SCHOVERLAID, Use of schema types conflicts with OVERLAID attribute

Error: The OVERLAID attribute cannot be used on programs or modules that discriminate schema at the outermost level.

SENDSPR, Internal Compiler Error

Fatal: An error has occurred in the execution of the VSI Pascal compiler. Along with this message, you will receive information that helps you find the location in the source program and the name of the compilation phase at which the error occurred. You may be able to rewrite the section of your program that caused the error and thus successfully compile the program. However, even if you are able to remedy the problem, please submit a report to VSI and provide a machine-readable copy of the program.

SEQ11FORT, PDP-11 specific directive SEQ11 treated as equivalent to FORTRAN directive

Information.

SETBASCOM, SET base types are not compatible

Error: The base type of two sets used in a set operation are not compatible.

SETELEORD, SET element expression must be of an ordinal type

Error: The expressions used to denote the elements of a set constructor or the bounds of a set type definition must have an ordinal type.

SETNOTRNG, SET element is not in range 0..255

Error: In a set whose base type is a subrange of integers or unsigned integers, all set elements in the set's type definition or in a constructor for the set must be in the range 0..255.

SIZACTDIFF, SIZE of actual differs from that of other parameters in same section

Error: All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size of the array's outermost dimension.

SIZARRNCA, Explicit size on ARRAY dimension makes CLASS_NCA mandatory

Error.

SIZATRTYPCON, Size attribute / type conflict

Error: For an ordinal type, the size specified must be at least as large as the packed size but no larger than 64 bits. Pointer types may be either 32 or 64 bits. Type SINGLE exactly 32 bits, type DOUBLE exactly 64 bits, and type QUADRUPLE exactly 128 bits. For types ARRAY, RECORD, SET, and VARYING OF CHAR, the size specified must be at least as large as their packed sizes. For the details of allocation sizes in VSI Pascal, see the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/].

SIZCASTYP, Variable's size conflicts with cast's target type

Error: In a type cast operation, the size of the variable and the size of the type to which it is cast must be identical.

SIZEDIFF, Sizes are different

Information: This message can appear as additional information on other error messages.

SIZEINT, Size expression must be a positive integer value

Error.

SIZGTRMAX, Size exceeds MAXINT bits

Error: The size of a record or an array type or the size specified by a size attribute exceeds 2,147,483,647 bits. The VSI Pascal implementation imposes this size restriction.

SIZMULTBYT, Size of component of array passed by descriptor is not a multiple of bytes

Error: When an array or a conformant parameter is passed using the %DESCR mechanism specifier, the descriptor built by the compiler must follow the *VSI OpenVMS Calling Standard*. Such a descriptor can describe only an array whose components fall on byte boundaries.

SPEOVRDECL, Foreign mechanism specifier required to override parameter declaration

Error: When you specify a default value for a formal VAR or routine parameter, you must also use a mechanism specifier to override the characteristics of the parameter section.

SPURIOUS, “error message” at “line number”--- “column number”

Information: The compiler did not correctly note the location of this error in your program and later could not position and print the correct error message. You may be able to correct the section of your program that caused the error and thus avoid this error. Please submit a report (SPR) and provide a machine-readable copy of the program if you receive this error.

SRCERRORS, Source errors inhibit continued compilation—correct and recompile

Fatal: A serious error previously detected in the source program has corrupted the compiler's symbol tables and inhibits further compilation. Correct the serious error and recompile the program.

SRCTXTIGNRD, Source text following end of compilation unit ignored

Warning: The compiler ignores any text following the END statement that terminates a compilation unit. This error probably resulted from an unmatched END statement in your program.

STDACTINCMP, Nonstandard: actual is not name compatible with other parameters in same section

Information: According to the Pascal standard, all actual parameters passed to a parameter section must have the same type identifier or the same type definition. This message is issued only if you have specified the standard switch on the compile command line.

STDATTRLST, Nonstandard: attribute list

STDBIGLABEL, Nonstandard: label number greater than 9999

STDBLANKPAD, Nonstandard: blank-padding used during string operation

STDBNDRMUSE, Nonstandard: usage of formal parameter for routine "routine name"

STDCALLFUNC, Nonstandard: function "function name" called as a procedure

STDCASLBLRNG, Nonstandard: label range in case selector

STDCAST, Nonstandard: type cast operator

STDCMPCOMPAT, Nonstandard: cannot "PACK or UNPACK", array component types are incompatible

STDCMPDIR, Nonstandard: compiler directive

STDCOMFUNACC, Nonstandard: component function access

STDCNFARR, Nonstandard: conformant array syntax

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDCNSTR, Nonstandard: array or record constructor

Information: A VAX Pascal Version 1.0 style constructor was used. You should convert this constructor to the new constructor syntax provided in the current version of VSI Pascal to be compatible with the Extended Pascal standard.

STDCONCAT, Nonstandard: concatenation operator

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDCONST, Nonstandard: "type name" constant

Information: Binary, hexadecimal, and octal constants and constants of type DOUBLE, QUADRUPLE, UNSIGNED, INTEGER64, and UNSIGNED64 are extensions to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDCONSTACC, Nonstandard: structured constant access

Information: This message is issued if you have specified a standard option other than extended on the compile command line.

STDCTLDECL, Nonstandard: control variable "variable name" not declared in VAR section of "block name"

Information: The Pascal standard requires that the control variable of a FOR statement be declared in the same block in which the FOR statement appears.

STDDECLSEC, Nonstandard: declaration sections either out of order or duplicated in "block name"

Information: In the Pascal standard, the declaration sections must appear in the order LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION. The ability to specify the sections in any order is an extension. This message occurs only if you have specified the standard switch on the compile command line.

STDDEFPARM, Nonstandard: default parameter declaration

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDDIRECT, Nonstandard: "directive name" directive

Information: The EXTERN, EXTERNAL, FORTRAN, and SEQ11 directives are extensions to Pascal. (FORWARD is the only directive specified by the Pascal standard.) This message is issued only if you have specified the standard switch on the compile command line.

STDDISCREF, Nonstandard: schema discriminant reference

STDDISCSCHEMA, Nonstandard: discriminated schema

Information: These messages are issued if you have specified a standard argument other than extended on the compile command line.

STDEMPCASLST, Nonstandard: empty case-list element

Information: This message is issued if you do not specify any case labels and executable statements between two semicolons or between OF and a semicolon in the CASE statement. You must also have specified the standard switch on the compile command line.

STDEMPPARM, Nonstandard: empty actual parameter position

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDEMPREC, Nonstandard: empty record section

Information: The Pascal standard does not allow record type definitions of the form RECORD END. This message appears only if you have specified the standard switch on the compile command line.

STDEMPSTR, Nonstandard: empty string

Information: This message refers to extensions to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDEMPVRNT, Nonstandard: empty variant

Information: This message occurs if you do not specify a variant between two semicolons or between OF and a semicolon. You must also have specified the standard switch on the compile command line.

STDEOLCOM, Nonstandard: End of line comment

Information: The message is issued if you use the exclamation point character to treat the remainder of the line as a comment. You must also have specified the standard switch on the compile command line.

STDERRPARM, Nonstandard: error-recovery parameter

STDEXPON, Nonstandard: exponentiation operator

STDEXTSTR, Nonstandard: extended string syntax

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDFLDHIDPTR, Nonstandard: record field identifier “field identifier name” hides type identifier “field identifier name”

Information.

STDFORIN, Nonstandard: SET-iteration in FOR statement

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDFORMECH, Nonstandard: foreign mechanism specifier

Information: This message refers to an extension to Pascal and is issued only if you have specified the standard switch on the compile command line.

STDFORWARD, Nonstandard: PROCEDURE/FUNCTION block “routine name” and its FORWARD heading are not in the same section

Information: The Extended Pascal standard requires that FORWARD declared routines must specify their corresponding blocks without intervening LABEL, CONST, TYPE, or VAR sections. This message is issued only if you have specified the extended argument to the standard switch on the compile command line.

STDFUNIDEVAR, Nonstandard: function identified variable

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDFUNCTRES, Nonstandard: FUNCTION returning a value of a “type name” type

Information: The ability of functions to have structured result types is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDINCLUDE, Nonstandard: %INCLUDE directive

STDINITVAR, Nonstandard: initialization syntax in VAR section

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDKEYWRD, Nonstandard: “keyword name”

Information: This message is issued if you have specified a standard option other than extended on the compile command line.

STDMATCHVRNT, Nonstandard: no matching variant label

Information: This message is issued if you call the NEW or DISPOSE procedure, and one of the case labels specified in the call does not correspond to a case label in the record variable. You must also have specified the standard switch on the compile command line.

STDMODCTL, Nonstandard: potential uplevel modification of "variable name" prohibits use as control variable

Information: You cannot use as the control variable of a FOR statement any variable that might be modified in a nested block. This message is issued only if you have specified the standard switch on the compile command line.

STDMODULE, Nonstandard: MODULE declaration

Information: The item listed in this message is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDNILCON, Nonstandard: use of reserved word NIL as a constant

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the standard switch on the compile command line.

STDNOFRML, Nonstandard: FUNCTION or PROCEDURE parameter declaration lacks formal parameter list

Information: This message is issued if you try to pass actual parameters to a formal routine parameter for which you declared no formal parameter list. You must also have specified the standard switch on the compile command line.

STDNONPOS, Nonstandard: nonpositional parameter syntax

STDOTHER, Nonstandard: OTHERWISE clause

STDPASSPRE, Nonstandard: passing predeclared "routine name"

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDNOTIN, Nonstandard: NOT IN operator

Information: This message refers to an extension in Pascal and is issued only if you have specified the standard switch on the compile command line.

STDPCKSET, Nonstandard: combination of packed and unpacked sets

Information: The Pascal standard does not allow packed and unpacked sets to be combined in set operations. This message is issued only if you have specified the standard switch on the compile command line.

STDPRECONST, Nonstandard: predeclared constant "constant name"

Information: The constants MAXCHAR, MAXINT64, MAXUNSIGNED, MAXUNSIGNED64, MAXREAL, MINREAL, EPSREAL, MAXDOUBLE, MINDOUBLE, EPSDOUBLE, MAXQUADRUPLE, MINQUADRUPLE, and EPSQUADRUPLE are extensions to Pascal. MAXCHAR, MAXREAL, MINREAL, and EPSREAL are contained in Extended Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDPREDECL, Nonstandard: predeclared "routine"

Information: Many predeclared procedures and functions are extensions to Pascal. The use of these routines causes this message to be issued if you have specified the standard switch on the compile command line.

STDPRESCH, Nonstandard: predefined schema "type name"

Information: This message is issued if you have specified a standard switch other than extended on the compile command line.

STDPRETYP, Nonstandard: predefined type "type name"

Information: The types SINGLE, DOUBLE, INTEGER64, QUADRUPLE, UNSIGNED, UNSIGNED64, and VARYING OF CHAR are extensions to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDQUOSTR, Nonstandard: quotes enclosing radix constant

Information: This message is issued if you have specified the extended option on the compile command line.

STDRADFORMAT, Nonstandard: use format "radix"#nnn for radix constant

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified the extended argument to the standard switch on the compile command line.

STDRADIX, Nonstandard: radix constant

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified a standard switch other than extended on the compile command line.

STDRDBIN, Nonstandard: binary input from a TEXT file

STDRDENUM, Nonstandard: enumerated type input from a TEXT file

STDRDHEX, Nonstandard: hexadecimal input from a TEXT file

STDRDOCT, Nonstandard: octal input from a TEXT file

STDRDSTR, Nonstandard: string input from a TEXT file

Information: The Pascal standard allows only INTEGER, CHAR, and REAL values to be read from a text file. The ability to read values of other types is an extension to Pascal. These messages are issued only if you have specified the standard switch on the compile command line.

STDREDECLNIL, Nonstandard: redeclaration of reserved word NIL

Information: The Pascal standard considers NIL a reserved word, while VSI Pascal considers it to be a predeclared identifier. Thus, if you have specified the standard switch on the compile command line, this message will be issued if you attempt to redefine NIL.

STDREM, Nonstandard: REM operator

Information: The item listed in this message is an extension to Pascal. This message is issued only if you have specified the standard switch on the compile command line.

STDSCHEMA, Nonstandard: schema type definition

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDSCHEMAUSE, Nonstandard: use of schema type

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDSIMCON, Nonstandard: only simple constant (optional sign) or quoted string

Information: Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, constants of type BOOLEAN, and enumerated types. This message is issued only if you have specified the standard switch on the compile command line.

STDSPECHAR, Nonstandard: "$" or "_" in identifier

STDSTRCOMPAT, Nonstandard: string compatibility

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDSTRUCT, Nonstandard: types do not have same name

Information: Because the Pascal standard does not recognize structural compatibility, two types must have the same type identifier or type definition to be compatible. This message is issued only if you have specified the standard switch on the compile command line.

STDSYMLABEL, Nonstandard: symbolic label

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDTAGFLD, Nonstandard: invalid use of tag field

Information: The tag field of a variant record cannot be a parameter to the ADDRESS function, nor can you pass it as a writable VAR, %REF, %DESCR, or %STDESCR formal parameter. This message is issued only if you have specified the standard switch on the compile command line.

STDTODECL, Nonstandard: TO BEGIN/END DO declaration

Information: This message is issued if you have specified a standard argument other than extended on the compile command line.

STDUNSAFE, Nonstandard: UNSAFE compatibility

Information: If you have used the UNSAFE attribute on an object that is later tested for compatibility, you will receive this message. You must also have specified the standard switch on the compile command line.

STDUSEDCNF, Nonstandard: conformant array used as a string

STDUSEDPCK, Nonstandard: PACKED ARRAY [1..1] OF CHAR used as a string

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDVALCNFPRM, Nonstandard: conformant array may not be passed to value conformant parameter

Information.

STDVALUE, Nonstandard: VALUE initialization section, STDVAXCDD, Nonstandard: %DICTIONARY directive

Information: These messages refer to extensions to Pascal and are issued only if you have specified the standard switch on the compile command line.

STDVRNTCNSTR, Nonstandard: variant field outside constructor variant part

Information: This message refers to the use of an extension to Pascal. This message is issued only if you have specified the extended argument to the standard switch the compile command line.

STDVRNTPART, Nonstandard: empty variant part

Information: According to the Pascal standard, a variant part that declares no case labels and field lists between the words OF and END is illegal. This message occurs only if you have specified the standard switch on the compile command line.

STDVRNTRNG, Nonstandard: variant labels do not cover the range of the tag type

Information: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record. This message is issued only if you have specified the standard switch on the compile command line.

STDWRTBIN, Nonstandard: binary output to a TEXT file

STDWRTENUM, Nonstandard: user defined enumerated type output to a TEXT file

STDWRTHEX, Nonstandard: hexadecimal output to a TEXT file

STDWRTOCT, Nonstandard: octal output to a TEXT file

Information: The Pascal standard allows only INTEGER, BOOLEAN, CHAR, REAL, and PACKED ARRAY [1..n] OF CHAR values to be written to a text file. The ability to write values of other types is an extension to Pascal. These messages are issued only if you have specified the standard switch on the compile command line.

STDSUBSTRING, Nonstandard: Substring notation

Information.

STDZERO, Nonstandard: ZERO function used in constructor

Information: This message refers to an extension in Pascal and is issued only if you have specified the standard switch on the compile command line.

STOREQEXC, Allocations to Psect "name" exceeded growth bounds

Error: Too much data is allocated to the Psect. Either place variables into different Psects or break up the program into multiple modules

STREQLLEN, String values must be of equal length

Error: You cannot perform string comparisons on character strings that have different lengths.

STROPNDREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) operand required

STRPARMREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) parameter required

STRTYPREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) type required

Error: The file-name parameter to the OPEN procedure and the parameter to the LENGTH function must be character strings of the types listed.

SYNASCII, Illegal ASCII character

SYNASSERP, Syntax: ":=", ";" or ")" expected

SYNASSIGN, Syntax: ":=" expected

SYNASSIN, Syntax: ":=" or IN expected

SYNASSSEMI, Syntax: ":=" or ";" expected

SYNATRCAST, Syntax: attribute list not allowed on a type cast

SYNATTTYPE, Syntax: attribute-list or type specification

SYNBEGDECL, Syntax: BEGIN or declaration expected

SYNBEGEND, Syntax: BEGIN or END expected

SYNBEGIN, Syntax: BEGIN expected

SYNCOASSERP, Syntax: ",", ":=", ";" or ")" expected

SYNCOELRB, Syntax: ",", ".." or "]" expected

SYNCOLCOMRP, Syntax: ":>", "," or ")" expected

SYNCOLON, Syntax: ":>" expected

SYNCOMCOL, Syntax: "," or ":>" expected

SYNCOMDO, Syntax: "," or DO expected

SYNCOMEQL, Syntax: "," or "=" expected

SYNCOMMA, Syntax: "," expected

SYNCOMRB, Syntax: "," or "]" expected

SYNCOMRP, Syntax: "," or ")" expected

SYNCOMSEM, Syntax: "," or ";" expected

SYNCONTMESS, Syntax: CONTINUE or MESSAGE expected\SYNCOSERP, Syntax: ",", ";" or ")" expected

SYNDIRBLK, Syntax: directive or block expected

Error: The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNDIRMIS, Syntax: directive missing, EXTERNAL assumed

Error: In the absence of a directive where one is expected, the compiler assumes that EXTERNAL is the intended directive and proceeds with compilation based on that assumption.

SYNDO, Syntax: DO expected

SYNELIPSIS, Syntax: ".." expected

SYNELSESTMT, Syntax: ELSE or start of new statement expected

SYNEND, Syntax: END expected

SYNEQL, Syntax: “=” expected

SYNEQLLP, Syntax: “=” or “)” expected

SYNERRCTE, Error in compile-time expression

SYNEXPR, Syntax: expression expected

SYNEXSEOTEN, Syntax: expression, “;”, OTHERWISE or END expected

SYNFUNPRO, Syntax: FUNCTION or PROCEDURE expected

SYNHEADTYP, Syntax: routine heading or type identifier expected

SYNIDCAEND, Syntax: identifier, CASE or END expected

SYNIDCARP, Syntax: identifier, CASE or “)” expected

SYNIDCASE, Syntax: identifier or CASE expected

SYNIDENT, Syntax: identifier expected

SYNILLEXPR, Syntax: ill-formed expression

SYNINT, Syntax: integer expected

SYNINTBOO, Syntax: integer, boolean, or string literal expected

SYNINVSEP, Syntax: invalid token separator

SYNIVATRLST, Syntax: illegal attribute list

SYNIVPARM, Syntax: illegal actual parameter

SYNIVPRMLST, Syntax: illegal actual parameter list

SYNIVSYM, Syntax: illegal symbol

SYNIVVAR, Syntax: illegal variable

SYNLABEL, Syntax: label expected

SYNLBRAC, Syntax: “[” expected

SYNLPAREN, Syntax: “(” expected

SYNLPASEM, Syntax: “(” or “;” expected

SYNLPCORB, Syntax: “(”, “,” or “]” expected

SYNLPSECO, Syntax: “(”, “;” or “:” expected

SYNMECHEXPR, Syntax: mechanism specifier or expression expected

SYNNEWSTMT, Syntax: start of new statement expected

SYNOF, Syntax: OF expected

SYNPARMLST, Syntax: actual parameter list

SYNPARMSEC, Syntax: parameter section expected

SYNPERIOD Syntax: “.” expected

SYNPROMOD, Syntax: PROGRAM or MODULE expected

SYNQUOSTR, Syntax: quoted string expected

SYNRBRAC, Syntax: “]” expected

SYNRESWRD, Syntax: reserved word cannot be redefined

SYNRPAREN, Syntax: “)” expected

SYNRPASEM, Syntax: “;” or “)” expected

SYNRTNTYPCNF, Syntax: routine heading, type identifier or conformant parameter expected
SYNSEMI, Syntax: “;” expected

SYNSEMIEND, Syntax: “;” or END expected

SYNSEMMODI, Syntax: “;”, “::”, “^”, or “[” expected

SYNSEMRB, Syntax: “;” or “]” expected

SYNSEOTEN, Syntax: “;”, OTHERWISE or END expected

SYNTHEN, Syntax: THEN expected

SYNTODOWN, Syntax: TO or DOWNTO expected

SYNSEOTRP, Syntax: “;”, OTHERWISE, or “)” expected

SYNTYPCNF, Syntax: type identifier or conformant parameter expected

SYNTYPID, Syntax: type identifier expected

Error: The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNTYPPACK, Only ARRAY, FILE, RECORD or SET types can be PACKED

Warning: You cannot pack any type other than the structured types listed in the message.

SYNTYPSPEC, Syntax: type specification expected

SYNUNEXDECL, Syntax: declaration encountered in executable section

SYNUNTIL, Syntax: UNTIL expected

SYNXTRASEMI, Syntax: "; ELSE" is not valid Pascal, ELSE matched with IF on line "line number"

Error: The compiler either detected an error in a lexical or syntactical element in your program, or it failed to find such an element where one was expected.

TAGNOTORD, Tag type must be an ordinal type

Error: The type of a variant record's tag field must be one of the ordinal types.

TOOIDXEXPR, Too many index expressions; type has only "number of dimensions" dimensions

Error: A call to the UPPER or LOWER function specified an index value that exceeds the number of dimensions in the dynamic array.

TOOMANYIFS, Conditional compilation nesting level exceeds implementation limit

Error: %IF directives may only be nested 32 deep.

TOPROGRAM, TO BEGIN/END DO not allowed in PROGRAM

Error: TO BEGIN DO and TO END DO declarations are only allowed in modules.

TYPCNTDISCR, Type can not be discriminated in this context

Error.

TYPFILSIZ, Type contains one or more FILE components, size attribute is illegal

Error: The allocation size of a FILE type cannot be controlled by a size attribute; therefore, you cannot use a size attribute on any type that has a file component.

TYPHASFILE, Type contains one or more FILE components

Error: Many operations are illegal on objects of type FILE and objects of structured types with file components; for example, you cannot initialize them, use them as value parameters, or read them with input procedures.

TYPHASNOVRNT, Type contains no variant part

Error: You can only use the formats of the NEW, DISPOSE, and SIZE routines that allow case labels to be specified when their parameters have variants.

TYPPTRFIL, Type must be pointer or FILE

Error: You cannot use the syntax "Variable^" to refer to an object whose type is not pointer or FILE.

TYPREF, %REF not allowed for this type

Error: The %REF foreign mechanism specifier cannot be used with schematic variables.

TYPSTDESCR, %STDESCR not allowed for this type

Error: The %STDESCR mechanism specifier is allowed only on objects of type CHAR, PACKED ARRAY [1..n] OF CHAR, VARYING OF CHAR, and arrays of these types.

TYPVARYCHR, Component type of VARYING must be CHAR

Error.

UNALIGNED, "variable name" is UNALIGNED

Error or Warning: You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

UNAVOLACC, Volatile access appears unaligned, but must be aligned at run-time to ensure atomicity and byte granularity

Warning: The code generator was unable to determine if a volatile access was aligned or not. It generated two sequences; one sequence will perform the atomic access if it was aligned properly; the second sequence accesses the object, but may contain a timing window where incorrect results may occur.

UNBPNTRET, "routine name" is not UNBOUND—frame-pointer not returned

Warning: The IADDRESS function returns only the address of the procedure value (on OpenVMS VAX systems, the entry mask of the routine is called). This address may be sufficient information to successfully invoke an unbound routine, but not a bound routine. (Bound routines are represented as a pair of addresses: one pointing to the procedure value and the other to the frame pointer to the routine in which the routine was declared.)

UNCALLABLE, Routine "name" can never be called

Information.

UNCERTAIN, "Variable name" has not been initialized

Information.

UNDECLFRML, Undeclared formal parameter "symbol name"

Error: A formal parameter name listed in a nonpositional call to a routine does not match any of the formal parameters declared in the routine heading.

UNDECLID, Undeclared identifier “symbol name”

Error: In Pascal, an identifier must be declared before it is used. There are no default or implied declarations.

UNDSCHILL, Undiscriminated schema type is illegal

Error: An undiscriminated schema type does not have any actual discriminants. Without discriminants, the type size, any nested ARRAY bounds, and the offset of any nested RECORD fields are unknown.

UNINIT, "Variable name" is fetched, not initialized

Information.

UNPREDRES, Calling FUNCTION “function name” declared FORWARD may yield unpredictable results

Warning: By using FORWARD declared functions in actual discriminant expressions, you can cause infinite loops at run time or access violations.

UNREAD, Variable, “variable name” is assigned into, but never read

Information.

UNSCNFVRY, UNSAFE attribute not allowed on conformant VARYING parameter

Error.

UNSEXCRNG, constant exceeds range of "datatype"

Error: The largest value allowed for an UNSIGNED value is 4,294,967,295. The largest value allowed for an UNSIGNED64 value is 18,446,744,073,709,551,615.

UNUSED, Variable, “variable name” is never referenced

Information.

UNWRITTEN, Variable “variable name” is read, but never assigned into

Warning.

UPLEVELACC, Unbound “routine name” precludes uplevel access to “variable name”

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

UPLEVELGOTO, Unbound “routine name” precludes uplevel GOTO to “label name”

Error: A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

USEDBFDECL, “symbol name” was used before being declared

Warning.

USEINISTA, Use initial-state (VALUE clause) on TYPE or VAR declaration

Information: Nonstatic variables, such as those created from schema types, cannot be initialized in the VALUE declaration part. To initialize these variables, you must use the initial state feature.

V1DYNARR, Decommitted Version 1 dynamic array type

Error: The type syntax used to define a dynamic array parameter has been decommitted for the current version of VSI Pascal. You should edit your program to make the type definition conform to the current version conformant array syntax.

V1DYNARRASN, Decommitted Version 1 dynamic array assignment

Error: In VAX Pascal Version 1.0, dynamic arrays used in assignments could not be checked for compatibility until run time. This warning indicates that your program depends on an obsolete feature, which you should consider changing to reflect the current version syntax for conformant array parameters.

V1MISSPARM, Decommitted missing parameter syntax: correct by adding “number of commas” comma(s)

Error: An OPEN procedure called with the decommitted VAX Pascal Version 1.0 syntax fails to mark omitted parameters with commas. Your program depends on this obsolete feature, and you should insert the correct number of commas as listed in the message.

V1PARMSYN, Use of unsupported V1 omitted parameter syntax with new V2 feature(s)

Error: In a parameter list for the OPEN procedure, you cannot use both the Version 1.0 syntax for OPEN and the parameters that are new to subsequent versions of VSI Pascal.

V1RADIX, Decommitted Version 1 radix output specification

Error: In VAX Pascal Version 1.0, octal and hexadecimal values could be written by placing the keywords OCT or HEX after a field width expression. Your program uses this obsolete feature; you should consider changing it to use the current versions OCT or HEX predeclared functions.

VALOUTBND, Value to be assigned is out of bounds

Error: A value specified in an array or record constructor exceeds the subrange defined as the type of the corresponding component.

VALUEINIT, VALUE variables must be initialized

Error: Variables with both the VALUE and GLOBAL attributes must be given an initial value in either the VAR section or in the VALUE section.

VALUETOOBIG, VALUE attribute not allowed on objects larger than 32 bits

Error: Variables with the VALUE attribute cannot be larger than 32 bits because they are expressed to the linker as global symbol references.

VALUETYP, VALUE allowed only on ordinal or real types

Error.

VALUEVISIB, GLOBAL or EXTERNAL visibility is required with the VALUE attribute

Error: Variables with the VALUE attribute must be given either external or global visibility. (If the variable is given global visibility, then it must also be given an initial value.)

VARCOMFRML, Variable is not compatible with formal parameter "formal parameter name"

Error: A variable being passed as an actual parameter is not compatible with the corresponding formal parameter indicated. Variable parameters must be structurally compatible. The reason for the incompatibility is provided in an informational message that the compiler prints along with this error message.

VARNOTEXT, Variable must be of type TEXT

Error: The EOLN function requires that its parameter be a file of type TEXT.

VARPRMRTN, Formal VAR parameter may not be a routine

Error: The reserved word VAR cannot precede the word PROCEDURE or FUNCTION in a formal parameter declaration.

VARPTRTYP, Variable must be of a pointer type

Error: The NEW and DISPOSE procedures operate only on pointer variables.

VARYFLDS, LENGTH and BODY are the only fields in a VARYING type

Error: You cannot use the syntax "Variable.Identifier" to specify any fields of a VARYING OF CHAR variable other than LENGTH and BODY.

VISAUTOCON, Visibility / AUTOMATIC allocation conflict

Error: The GLOBAL, EXTERNAL, WEAK_GLOBAL, and WEAK_EXTERNAL attributes require static allocation and therefore conflict with the AUTOMATIC attribute.

VISGLOBEXT, Visibilities are not GLOBAL/EXTERNAL or EXTERNAL/EXTERNAL

Information: In repeated declarations of a variable or routine, only one declaration at most can be global; all others must be external. This message can appear as additional information for other error messages.

VRNTRNG, Variant labels do not cover the range of the tag type

Error: According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record or include an OTHERWISE clause.

WDTHONREAL, Second field width is allowed only when value is of a real type

Error: The fraction value in a field-width specification is allowed only for real-number values.

WRITEONLY, “variable name” is WRITEONLY

Warning: You cannot use a write-only variable in any context that requires the variable to be evaluated. For example, a write-only variable cannot be used as the control variable of a FOR statement.

XTRAERRORS, Additional diagnostics occurred on this line

Information: The number of errors occurring on this line exceeds the implementation's limit for outputting errors. You should correct the errors given and recompile your program.

ZERNOTALL, ZERO is not allowed for type or types containing “type name”

Error: ZERO may not be used to initialize objects of type FILE, TEXT, or TIMESTAMP or objects containing these types.

C.2. Run-Time Diagnostics

During execution, an image can generate a fatal error called an exception condition. When the VSI Pascal run-time system detects such a condition, the system displays an error message and aborts program execution.

Run-time errors can also be issued by other facilities, such as the VSI OpenVMS Sort Utility or the VSI OpenVMS operating system. VSI Pascal run-time system diagnostics are preceded by the following:

```
%PAS- F-
```

The severity level of a run-time error is F, fatal error.

Some conditions, particularly I/O errors, may cause several messages to be generated. The first message is a diagnostic that specifies the file that was being accessed (if any) when the error occurred and the nature of the error. Next, an RMS error message may be generated. In most cases, you should be able to understand the error by looking up the first message in the following list. If not, see the *OpenVMS System Messages and Recovery Procedures Reference Manual* for an explanation of the RMS error message.

All diagnostic messages contain a brief explanation of the event that caused the error. This section lists run-time diagnostic messages in alphabetical order, including explanatory message text. Where the message text is not self-explanatory, additional explanation follows. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

ACCMETINC, ACCESS_METHOD specified is incompatible with this file

The value of the ACCESS_METHOD parameter for a call to the OPEN procedure is not compatible with the file's organization or record type. You can use DIRECT access only with files that have relative organization or sequential organization and fixed-length records. You can use KEYED access only with indexed files. Make sure that you are accessing the correct file. See Chapter 7 to determine which access method you should use.

AMBVALENU, “string” is an ambiguous value for enumerated type “type”

While a value of an enumerated type was being read from a text file, not enough characters of the identifier were found to specify an unambiguous value. Specify enough characters of the identifier so that it is not ambiguous.

ARRINDVAL, array index value is out of range

You enabled bounds checking for a compilation unit and attempted to specify an index that is outside the array's index bounds. Correct the program or data so that all references to array indexes are within the declared bounds.

ARRNOTCOM, conformant array is not compatible

You attempted to assign one dynamic array to another that did not have the same index bounds. This error occurs only when the arrays use the decommitted VAX Pascal Version 1.0 syntax for dynamic array parameters. Correct the program so that the two dynamic arrays have the same index bounds. You could also change the arrays to conform to the current syntax for conformant arrays; most incompatibilities could then be detected at compile time rather than at run time. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information on current conformant arrays.

ARRNOTSTR, conformant array is not a string

In a string operation, you used a conformant PACKED ARRAY OF CHAR value whose index had a lower bound not equal to 1 or an upper bound greater than 65535. Correct the array's index so that the array is a character string.

ASSERTION, Pascal assertion failure

The expression used in the Pascal ASSERT built-in routine evaluated to false. Correct the problem that was being checked with the ASSERT built-in in the source program.

BUGCHECK, internal consistency failure “nnn” in Pascal Run-Time Library

The run-time library has detected an internal error or inconsistency. This problem may be caused by an out-of-bounds array reference or a similar error in your program. Rerun your program with all CHECK options enabled. If you are unable to find an error in your program, please submit a Software Performance Report (SPR) to VSI, including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

CANCNTERR, handler cannot continue from a nonfile error

A user condition handler attempted to return SS$_CONTINUE for an error not involving file input/output. To recover from such an error, you must use either an uplevel GOTO statement or the SYS$UNWIND system service. Modify the user handler to use one of the allowed recovery actions for nonfile errors, or to resignal the error if no recovery action is possible.

CASSELVAL, CASE selector value is out of range

The value of the case selector in a CASE statement does not equal any of the specified case labels, and the statement has no OTHERWISE clause. Either add an OTHERWISE clause to the CASE statement or change the value of the case selector so that it equals one of the case labels. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

CONCATLEN, string concatenation has more than 65535 characters

The result of a string concatenation operation would result in a string longer than 65,535 characters, which is the maximum length of a string. Correct the program so that all concatenations result in strings no longer than 65,535 characters.

CSTRCOMISS, invalid constructor: component(s) missing

The constructor did not specify sufficient component values to initialize a variable of the type. Specify more components in the constructor, use the OTHERWISE clause in the constructor, or modify the type definition to specify fewer components.

CURCOMUND, current component is undefined for DELETE or UPDATE

You attempted a DELETE or UPDATE procedure when no current component was defined. A current component is defined by a successful GET, FIND, FINDK, RESET, or RESETK that locks the component. Files opened with HISTORY:=READONLY never lock components. Correct the program so that a current component is defined before executing DELETE or UPDATE.

DELNOTALL, DELETE is not allowed for a sequential organization file

You attempted a DELETE procedure for a file with sequential organization, which is not allowed. DELETE is valid only on files with relative or indexed organization. Make sure that the program is referencing the correct file. See Chapter 7 to determine what file characteristics are appropriate for your application.

ERRDURCLO, error during CLOSE

RMS reported an unexpected error during execution of the CLOSE procedure. The RMS error message is also displayed. This message may also be issued with error severity when files are implicitly closed during a procedure or image exit. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDEL, error during DELETE

RMS reported an unexpected error during execution of a DELETE procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDIS, error during DISPOSE

An error occurred during execution of a DISPOSE procedure. An additional message that further describes the error may also be displayed. Make sure that the heap storage being freed was allocated by a successful call to the NEW procedure, and that it has not been already freed. If an additional message is shown, see the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of that message.

ERRDUREXT, error during EXTEND

RMS reported an unexpected error during execution of an EXTEND procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURFIN, error during FIND or FINDK

RMS reported an unexpected error during execution of a FIND or FINDK procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURGET, error during GET

RMS reported an unexpected error during execution of the GET procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURMAR, error during MARK

An error occurred during execution of the PAS$MARK2 procedure. An additional message is displayed that further describes the error. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURNEW, error during NEW

An error occurred during execution of the NEW procedure. An additional message is displayed that further describes the error. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDUROPE, error during OPEN

An unexpected error occurred during execution of the OPEN procedure, or during an implicit open caused by a RESET or REWRITE procedure. An additional message is displayed that further describes the error. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURPRO, error during prompting

RMS reported an unexpected error during output of partial lines to a terminal. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURPUT, error during PUT

RMS reported an unexpected error during execution of the PUT procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS message.

ERRDURREL, error during RELEASE

An unexpected error occurred during execution of the PAS$RELEASE2 procedure. An additional message may be displayed that further describes the error. Make sure that the marker argument was returned from a successful call to PAS$MARK2 and that the storage has not been already freed. If an additional message is displayed, see the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of that message.

ERRDURRES, error during RESET or RESETK

RMS reported an unexpected error during execution of the RESET or RESETK procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURREW, error during REWRITE

RMS reported an unexpected error during execution of the REWRITE procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURTRU, error during TRUNCATE

RMS reported an unexpected error during execution of the TRUNCATE procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUNL, error during UNLOCK

RMS reported an unexpected error during execution of the UNLOCK procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUPD, error during UPDATE

RMS reported an unexpected error during execution of the UPDATE procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURWRI, error during WRITELN

RMS reported an unexpected error during execution of the WRITELN procedure. The RMS error message is also displayed. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

EXTNOTALL, EXTEND is not allowed for a shared file

Your program attempted an EXTEND procedure for a file for which the program did not have exclusive access. EXTEND requires that no other users be allowed to access the file. Note that this message may also be issued if you do not have permission to extend to the file. Correct the program so that the file is opened with SHARING:=NONE, which is the default, before performing an EXTEND procedure.

FAIGETLOC, failed to GET locked component

Your program attempted to access a component of a file that was locked by another user. You can usually expect this condition to occur when more than one user is accessing the same relative or indexed file. Determine whether this condition should be allowed to occur. If so, modify your program so that it detects the condition and retries the operation later. See Chapter 7 for more information.

FILALRACT, file “file name” is already active

Your program attempted a file operation on a file for which another operation was still in progress. This error can occur if a file is used in AST or condition-handling routines. Modify your program so that it does not try to use files that may currently be in use.

FILALRCLO, file is already closed

Your program attempted to close a file that was already closed. Modify your program so that it does not try to close files that are not open.

FILALROPE, file is already open

Your program attempted to open a file that was already open. Modify your program so that it does not try to open files that are already open.

FILNAMREQ, FILE_NAME required for this HISTORY or DISPOSITION

Your program attempted to open a nonexternal file without specifying a file-name parameter to the OPEN procedure, but the HISTORY or DISPOSITION parameter specified requires a file name. Add a file-name parameter to the OPEN procedure call, specifying an appropriate file name.

FILNOTDIR, file is not opened for direct access

Your program attempted to execute a DELETE, FIND, LOCATE, or UPDATE procedure on a file that was not opened for direct access. Modify the program to specify the ACCESS_METHOD:=DIRECT parameter to the OPEN procedure when opening the file. See Chapter 7 to determine if direct access is appropriate for your application.

FILNOTFOU, file not found

Your program attempted to open a file that does not exist. An additional RMS message is displayed that further describes the problem. Make sure that you are specifying the correct file. See the *OpenVMS System Messages and Recovery Procedures Reference Manual* for a description of the additional RMS message.

FILNOTGEN, file is not in Generation mode

Your program attempted a file operation that required the file to be in generation mode (ready for writing). Modify the program to use a REWRITE, TRUNCATE, or LOCATE procedure to place the file in generation mode as appropriate. See Chapter 7 for more information.

FILNOTINS, file is not in Inspection mode

Your program attempted a file operation that required the file to be in inspection mode (ready for reading). Modify the program to use a RESET, RESETK, FIND, or FINDK procedure to place the file in inspection mode as appropriate. See Chapter 7 for more information.

FILNOTKEY, file is not opened for keyed access

Your program attempted to execute a FINDK, RESETK, DELETE, or UPDATE procedure on a file that was not opened for keyed access. Modify the program to specify the ACCESS_METHOD:=KEYED parameter to the OPEN procedure when opening the file. See Chapter 7 to make sure that keyed access is appropriate to your application.

FILNOTOPE, file is not open

Your program attempted to execute a file manipulation procedure on a file that was not open. Correct the program to open the file using a RESET, REWRITE, or OPEN procedure as appropriate. See Chapter 7 for more information.

FILNOTSEQ, file is not sequential organization

Your program attempted to execute the TRUNCATE procedure on a file that does not have sequential organization. TRUNCATE is valid only on sequential files. Make sure that your program is accessing the correct file. Correct the program so that all TRUNCATE operations are performed on sequential files.

FILNOTTEX, file is not a textfile

Your program performed a file operation that required a file of type TEXT on a nontext file. Note that the type FILE OF CHAR is not equivalent to TEXT unless you have compiled the program with the `/OLD_VERSION` qualifier. Make sure that your program is accessing the correct file. Correct the program so that a text file is always used when required.

GENNOTALL, Generation mode is not allowed for a READONLY file

Your program attempted to place a file declared with the READONLY attribute into generation mode, which is not allowed. Note that the READONLY file attribute is not equivalent to the HISTORY:=READONLY parameter to the OPEN procedure. Correct the program so that the file either does not have the READONLY attribute or is not placed into generation mode.

GETAFTEOF, GET attempted after end-of-file

Your program attempted a GET operation on a file while EOF(f) was TRUE. This situation occurs when a previous GET operation (possibly implicitly performed by a RESET, RESETK, or READ procedure) reads to the end of the file and causes the EOF(f) function to return TRUE. If another GET is then performed, this error is given. Correct the program so that it either tests whether EOF(f) is TRUE, before attempting a GET operation, or repositions the file before the end-of-file marker.

GOTOFAILED, non-local GOTO failed

An error occurred while a nonlocal GOTO statement was being executed. This error might occur because of an error in the user program, such as an out-of-bounds array reference. Rerun your program, enabling all CHECK options. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to VSI, and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

HALT, HALT procedure called

The program terminated its execution by executing the HALT procedure. This message is solely informational. None.

ILLGOTO, illegal uplevel GOTO during routine activation

An uplevel GOTO was made into the body of a routine before the declaration part of the routine was completely processed. Correct the program to avoid the uplevel GOTO until the declaration part has been completely processed.

INSNOTALL, Inspection mode is not allowed for a WRITEONLY file

Your program attempted to place a file declared with the WRITEONLY attribute into inspection mode, which is not allowed. Correct the program so that the file variable either does not have the WRITEONLY attribute or is not placed into inspection mode.

INSVIRMEM, insufficient virtual memory

The run-time library was unable to allocate enough heap storage to open the file. Examine your program to see whether it is making excessive use of heap storage, which might be allocated using the NEW procedure or the run-time library procedure LIB$GET_VM. Modify your program to free any heap storage it does not need.

INVARGPAS, invalid argument to Pascal Run-Time Library

An invalid argument or inconsistent data structure was passed to the run-time library by the compiled code, or a system service returned an unrecognized value to the run-time library. Rerun your program with all CHECK options enabled. Make sure that the version of the current operating system is compatible with the version of the compiler. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to VSI, and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

INVFILSYN, invalid file name syntax

Your program attempted to open a file with an invalid file name. The file name used can be derived from the file variable name, the value of the file-name parameter to the OPEN procedure, or the logical name translations (if any) of the file variable name and portions of the file-name parameter and your default device and directory. The displayed text may include the erroneous file name. This error can also occur if the value of the file-name parameter is longer than 255 characters. Additional RMS messages may be displayed that further describe the error. Use the information provided in the displayed messages to determine which component of the file name is invalid. Verify that any logical names used are defined correctly. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for information on file names.

INVFILVAR, invalid file variable at location "nnn"

The file variable passed to a run-time library procedure was invalid or corrupted. This problem might be caused by an error in the user program, such as an out-of-bounds array access. Rerun your program with all CHECK options enabled, and recompile all modules using the same compiler. If the problem persists, please submit a Software Performance Report (SPR) to VSI and include a machine-readable copy of your program, data, and results of a sample execution showing the problem.

INVKEYDEF, invalid key definition

Your program attempted to open a file of type RECORD whose component type contained a field with an invalid KEY attribute. One of the following errors occurred:

- A new file was being created and the key numbers were not dense.
- A key field was defined at an offset of more than 65,535 bytes from the beginning of the record.

If a new file is being created, make sure that the key fields are numbered consecutively, starting with 0 for the required primary key. If you are opening an existing file, you must explicitly specify HISTORY:=OLD or HISTORY:=READONLY as a parameter to the OPEN procedure. Make sure that the length of the record is within the maximum permitted for the file organization being used. See Chapter 7 for more information.

INVRADIX, specified radix must be in the range 2-36

The specified radix for writing an ordinal value must be in the range of 2 through 36. Modify the program to specify a radix in the proper range.

INVRECLEN, invalid record length of “nnn”

A file was being opened, and one of the following errors occurred:

- The length of the file components was greater than that allowed for the file organization and record format (for most operations, the largest length allowed is 32,765 bytes).
- The value of the RECORD_LENGTH parameter to the OPEN procedure was greater than that allowed for the file organization and record format (for most operations, the largest value allowed is 32,765 bytes).

Correct the program so that the record length used is within the permitted limits for the type of file being used. See the *VSI OpenVMS Record Management Services Reference Manual* for more information.

INVSYNBIN, “string” is invalid syntax for a binary value

While a READ or READV procedure was reading a binary value from a text file, the characters read did not conform to the syntax for a binary value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNHEX, “string” is invalid syntax for a hexadecimal value

While a READ or READV procedure was reading a hexadecimal value from a text file, the characters read did not conform to the syntax for an hexadecimal value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNENU, “string” is invalid syntax for an enumerated value

While a READ or READV procedure was reading an identifier of an enumerated type from a text file, the characters read did not conform to the syntax for an enumerated value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNINT, “string” is invalid syntax for an integer value

While a READ or READV procedure was reading a value for an integer identifier from a text file, the characters read did not conform to the syntax for an integer value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNOCT, “string” is invalid syntax for an octal value

While a READ or READV procedure was reading an octal value from a text file, the characters read did not conform to the syntax for an octal value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNREA, “string” is invalid syntax for a real value

While a READ or READV procedure was reading a value for a real identifier from a text file, the characters read did not conform to the syntax for a real value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

INVSYNUNS, “string” is invalid syntax for an unsigned value

While a READ or READV procedure was reading a value for an unsigned identifier from a text file, the characters read did not conform to the syntax for an unsigned value. The displayed message includes the text actually read and the record number in which this text occurred. Correct the program or the input data so that the correct syntax is used. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

KEYCHANOT, key field change is not allowed

Your program attempted an UPDATE procedure for a record of an indexed file that would have changed the value of a key field, and this situation was disallowed when the file was created. If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly change a key field, or recreate the file specifying that the key field is permitted to change. See Chapter 7 for more information.

KEYDEFINC, KEY “nnn” definition is inconsistent with this file

An indexed file of type RECORD was opened, and the component type contained fields whose KEY attributes did not match those of the existing file. The number of the key in error is displayed in the message. Correct the RECORD definition so that it describes the correct KEY fields, or recreate the file so that it matches the declared keys. See Chapter 7 for more information.

KEYDUPNOT, key field duplication is not allowed

Your program attempted an UPDATE or PUT procedure for a record of an indexed file that would have duplicated a key field value of an existing record, and this situation was disallowed when the file was created. If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the PUT or UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly duplicate a key field, or recreate the file specifying that the key field is permitted to be duplicated. See Chapter 7 for more information.

KEYNOTDEF, KEY “nnn” is not defined for this file

Your program attempted a FINDK or RESETK procedure on an indexed file, and the key number specified does not exist in the file. Correct the program so that the correct key numbers are used when accessing the file.

KEYVALINC, key value is incompatible with the file's key “nnn”

The key value specified for the FINDK procedure was incompatible in type or size with the key field of the file, or your program attempted an OPEN on an existing file and the key check failed. Make sure that the correct key value is being specified for FINDK and OPEN. Correct the program so that the type of the key value is compatible with the key of the file.

LINTOOLON, line is too long, exceeded record length by “nnn” character(s)

Your program attempted a WRITE, PUT, WRITEV, or other output procedure on a text file that would have placed more characters in the current line than the record length of the file would allow. The number of characters that did not fit is displayed in the message. Correct the program so that it does not place too many characters in the current line. If appropriate, use the WRITELN procedure, or specify an increased record length parameter when opening the file with the OPEN procedure.

LINVALEXC, LINELIMIT value exceeded

The number of lines written to the file exceeded the maximum specified as the line limit. The line limit value is determined by the translation of the logical name PAS$LINELIMIT, if any, or the value specified in a call to the LINELIMIT procedure for the file. As appropriate, correct the program so that it does not write as many lines, or increase the line limit for the file. Note that if a line limit is specified for a nontext file, each PUT procedure called for the file is considered to be one line. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

LOWGTRHIGH, low-bound exceeds high-bound

The lower bound of a subrange definition is larger than the higher bound. Modify the declaration so the lower bound is less than or equal to the higher bound.

MAXLENRNG, maximum length must be in range 1..65535

The maximum length for a string type is 65,535. Modify the declaration to specify a smaller amount.

MODNEGNUM, MOD of a negative modulus has no mathematical definition

In the MOD operation A MOD B, the operand B must have a positive integer value. Correct the program so that the operand B has a positive integer value.

NEGDIGARG, negative Digits argument to BIN, HEX or OCT is not allowed

Your program attempted to specify a negative value for the Digits argument in a call to the BIN, HEX, or OCT procedure, which is not permitted. Correct the program so that only nonnegative Digits arguments are used for calls to BIN, HEX, and OCT.

NEGWIDDIG, negative Width or Digits specification is not allowed

A WRITE or WRITEV procedure on a text file contained a field width specification that included a negative Width or Digits value, which is not permitted. Correct the program so that only nonnegative Width and Digits parameters are used.

NOTVALTYP, "string" is not a value of type "type"

Your program attempted a READ or READV procedure on a text file, but the value read could not be expressed in the specified type. For example, this error results if a real value read is outside the range of the identifier's type, or if an enumerated value is read that does not match any of the valid constant identifiers in its type. Correct the program or the input data so that the values read are compatible with the types of the identifiers receiving the data.

OPNDASSCOM, operands are not assignment compatible

The operands do not have the same type. Examine the declarations of the operands and make sure they have compatible types.

ORDVALOUT, ordinal value is out of range

A value of an ordinal type is outside the range of values specified by the type. For example, this error results if you try to use the SUCC function on the last value in the type or the PRED function on the first value. Correct the program so that all ordinal values are within the range of values specified by the ordinal type.

ORGSPEINC, ORGANIZATION specified is inconsistent with this file

The value of the ORGANIZATION parameter for the OPEN procedure that opened an existing file was inconsistent with the actual organization of the file. Correct the program so that the correct organization is specified. See Chapter 7 for more information.

PADLENERR, PAD length error

The length of the character string to be padded by the PAD function is greater than the length specified as the finished size, or the finished size specified is greater than 65,535. Correct the call to PAD so that the finished size specified describes a character string of the correct length. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for the rules governing the PAD function.

PTRREFNIL, pointer reference to NIL

Your program attempted to evaluate a pointer value while its value was NIL. Make sure that the pointer has a value before you try to evaluate it. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information on pointer values.

RECLENINC, RECORD_LENGTH specified is inconsistent with this file

The record length obtained from the file component's length or from the value of the record length parameter specified for the OPEN procedure was inconsistent with the actual record length of an existing file. Correct the program so that the record length specified, if any, is consistent with the file. See Chapter 7 for more information.

RECTYPINC, RECORD_TYPE specified is inconsistent with this file

The value of the RECORD_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record type of an existing file. Correct the program so that the record type specified, if any, is consistent with the file. See Chapter 7 for more information.

REFINAVAR, read or write of inactive variant

A field of an inactive variant was read or written. Correct the program so the variant is active or remove the reference to the inactive field.

RESNOTALL, RESET is not allowed on an unopened internal file

Your program attempted a RESET procedure for a nonexternal file that was not open. This operation is not permitted because RESET must operate on an existing file, and there is no information associated with a nonexternal file that allows RESET to open it. Correct the program so that nonexternal files are opened before using RESET. Either OPEN or REWRITE may be used to open a nonexternal file. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information.

REWNOTALL, REWRITE is not allowed for a shared file

Your program attempted a REWRITE procedure for a file for which the program did not have exclusive access. REWRITE requires that no other users be allowed to access the file while the file's data is deleted. Note that this message may also be issued if you do not have permission to write to the file. Correct the program so that the file is opened with SHARING := NONE, which is the default, before performing a REWRITE procedure.

SETASGVAL, set assignment value has element out of range

Your program attempted to assign to a set variable a value that is outside the range specified by the variable's component type. Correct the assignment statement so that the value being assigned falls within the component type of the set variable. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information on sets.

SETCONVAL, set constructor value out of range

Your program attempted to include in a set constructor a value that is outside the range specified by the set's component type, or a value that is greater than 255 or less than 0. Correct the constructor so that it includes only those values within the range of the set's component type. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for more information on sets.

SETNOTRNG, set element is not in range 0..255

Sets of INTEGER or UNSIGNED must be in the range of 0..255. Modify the declaration to specify a smaller range.

STRASGLEN, string assignment length error

Your program attempted to assign to a string variable a character string that is longer than the declared maximum length of the variable (if the variable's type is VARYING) or that is not of the same length as the variable (if the variable's type is PACKED ARRAY OF CHAR). Correct the program so that the string is of a correct length for the variable to which it is being assigned.

STRCOMLEN, string comparison length error

Your program attempted to compare two character strings that do not have the same current length. Correct the program so that the two strings have the same length at the time of the comparison.

SUBASGVAL, subrange assignment value out of range

Your program attempted to assign to a subrange variable a value that is not contained in the subrange type. Correct the program so that all values assigned to a subrange variable fall within the variable's type.

SUBSTRSEL, SUBSTR selection error

A SUBSTR function attempted to extract a substring that was not entirely contained in the original string. Correct the call to SUBSTR so that it specifies a substring that can be extracted from the original string. See the *VSI Pascal Reference Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-reference-manual/] for complete information on the SUBSTR function.

TEXREQSEQ, textfiles require sequential organization and access

Your program attempted to open a file of type TEXT that either did not have sequential organization, or had an ACCESS_METHOD other than SEQUENTIAL (the default) when opened by the OPEN procedure. Make sure that the program refers to the correct file. Correct the program so that only sequential organization and access are used for text files.

TRUNOTALL, TRUNCATE is not allowed for a shared file

Your program attempted to call the TRUNCATE procedure for a file that was opened for shared access. You cannot truncate files that might be shared by other users. This message may also be issued if you do not have permission to write to the file. Correct the program so that it does not try to truncate shared files. If the file is opened with the OPEN procedure, do not specify a value other than NONE (the default) for the SHARING parameter.

UPDNOTALL, UPDATE not allowed for a sequential organization file

Your program attempted to call the UPDATE procedure for a sequential file. UPDATE is valid only on relative and indexed files. Correct the program so that it does not try to use UPDATE for sequential files, or recreate the file with relative or indexed organization. If you are using direct access on a sequential file, individual records can be updated with the LOCATE and PUT procedures. See Chapter 7 to determine whether a different file organization may be appropriate for your application.

VARINDVAL, VARYING index value exceeds current length

The index value specified for a VARYING OF CHAR string is greater than the string's current length. Correct the index value so that it specifies a legal character in the string.

WIDTOOLRG, total width too large

The requested total-width for the floating point write operation overflowed an internal buffer. Examine the source program to see if the specified total-width parameter is correct. If it is correct, please submit a problem report including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

WRIINVENU, WRITE of an invalid enumerated value

Your program attempted to write an enumerated value using a WRITE or WRITEV procedure, but the internal representation of that value was outside the possible range for the enumerated type. Verify that your program is not improperly using PRED, SUCC, or type casting to assign an invalid value to a variable of enumerated type.