

VSI Pascal V6.6-149 for OpenVMS x86-64

Release Notes

Publication Date: October 2026

Operating System: VSI OpenVMS Version 9.2-2 + Update 2 or higher

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

VSI Pascal V6.6-149 for OpenVMS x86-64 Release Notes

VMS Software

Copyright © 2026 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.

All other trademarks and registered trademarks mentioned in this document are the property of their respective holders.

Table of Contents

1. Overview of VSI Pascal 4
2. VSI Pascal for OpenVMS x86-64 Systems 4
    2.1. New in This Release 4
    2.2. New in Previous Releases 4
3. VSI Pascal Documentation 5
4. Debug Support Limitations 5
5. Debug Support for Schema Types 5
6. Using 64-bit Pointer Types 6
    6.1. Language Features Not Supported With 64-bit Pointers 6
    6.2. Using 64-bit Pointers With System Definition Files 7
7. Creating Files With RMS "Undefined" Record Types 9
8. Using the POS Attribute With Natural Alignment 9
9. Using $FILESCAN With VSI Pascal 10
10. Using Condition Handlers That Return SS$_CONTINUE 10
11. Using Asynchronous RMS I/O With Pascal 11
12. Known Problems and Restrictions 11
13. STARLET Definition Files 15
14. Compiling for Optimal Performance 15
15. Alignment Faults 15
    15.1. Understanding Alignment Faults 15
    15.2. What Does the Compiler Know? 16
    15.3. What Does the Compiler Assume? 16
16. Problems Corrected Since Last Release of VSI Pascal 17

1. Overview of VSI Pascal

The VSI Pascal Compiler V6.6-149 requires OpenVMS Version 9.2-2 + Update 2 or higher. For the best debugging experience, VSI recommends OpenVMS Version 9.2-3 + Update 3 or higher.

2. VSI Pascal for OpenVMS x86-64 Systems

Several features of VSI Pascal are currently not available on OpenVMS x86-64 systems. They are listed in *Section 12, "Known Problems and Restrictions"* in these release notes.

2.1. New in This Release

This V6.6-149 release includes the following new features:

QUADRUPLE Support

Beginning with Version 6.6, most QUADRUPLE support is now available. Full QUADRUPLE support requires the system to be running Version 9.2-4. The E9.2-4 field test release does not include the complete set of changes.

/MACHINE_CODE Qualifier Support

The **/MACHINE_CODE** qualifier causes the compiler to automatically invoke **ANALYZE/OBJECT/DISASSEMBLE** and append the output to the listing file. In addition, the compiler generates additional information in the object file that enables ANALYZE to provide enhanced formatting of the generated machine code.

2.2. New in Previous Releases

The V6.5-148 release included the following new features:

Installation Procedure Now Uses PCSI Utility

The Pascal installation procedure has been converted from the older VMSINSTAL command file technology to the PCSI utility.

The VMSINSTAL script provided the ability to choose to individually install the compiler portion, the STARLET library files portion, and the examples portion. This feature existed mostly to support older installation media of the past (TK50 and TU58 cartridge tapes) as well as small system disks where the extra space of examples might be a problem. Modern systems now have larger disks and online distribution. The PCSI installation will always install the compiler, build the STARLET files, and provide the example files.

The VMSINSTAL script also provided an option to save the current version of the compiler before installing a newer version. The PCSI procedure has the same functionality. During the installation, PCSI will ask:

```
Do you want the defaults for all options? [YES]
```

If the answer is NO, you will be prompted along with the description if a "?" is specified:

```
Do you want the defaults for all options? [YES] no

    Save existing compiler image [NO] ?
```

```
Pascal provides the ability to save the existing compiler image to be
selected with the PASCAL$SET_VERSION.COM and PASCAL$SHOW_VERSIONS.COM
command files. This provides the ability to have multiple versions of
Pascal available on the system.

Save existing compiler image [NO]
```

PCSI does not know about any saved compiler so if PASCAL is removed with **PRODUCT REMOVE**, any saved compilers are not removed.

3. VSI Pascal Documentation

The VSI Pascal documentation on the VSI documentation website [http://docs.vmssoftware.com] has been updated with all the Pascal features that had been included in previous release notes.

4. Debug Support Limitations

VSI Pascal has limited debug support on OpenVMS Version 9.2-2 and Version 9.2-3 prior to update 2. The older debuggers have issues with understanding the particular dialect of DWARF debugging records emitted by the LLVM backend. In addition, older Pascal compilers before V6.5-148 did not generate sufficient DWARF information for several Pascal datatypes.

5. Debug Support for Schema Types

VSI Pascal has partial debugging support for schema types. When **/DEBUG** is used along with schema types, the compiler generates helper routines that will be used by the debugger to compute various pieces of run-time information it needs to examine or deposit into schematic variables. These routines have names that include "%BOUND", "%STRIDE", "%SIZE", and "%OFFSET" strings in their names. They should not impact the user program other than the fact that these routines are included in the generated code.

We strongly encourage you to use **/NOOPTIMIZE** along with **/DEBUG** when debugging code that contains schema types. We have tested debugging with optimizations enabled, but the impact of optimization techniques make debugging difficult.

This support for debugging schema types exposes several bugs in the existing debugger especially on OpenVMS IA-64 and OpenVMS x86-64. If you try to debug schema types with the existing debugger, you may encounter wrong answers or debug errors.

For example:

```
program test;
type
  subr(l,u:integer) = l..u;

var
  a : [volatile] integer value 3;
  b : [volatile] integer value 3;
  r : record
      f5_1 :  array [subr(a,b)] of integer;
      case f5_tag  : integer of
            1      : (f5_case_1 : integer);
            2      : (f5_case_2 : boolean);
      end; { case }
```

```
begin
r := zero;
r.f5_tag := 2;
end.
```

You can examine variable R as an entire variable, but if you ask for R.F5_CASE_1 or R.F5_CASE_2, the debugger tries to examine the wrong virtual address.

For example:

```
DBG> ex r
TEST\R
    F5_1
        [3]:            0
    F5_TAG:     2
    Variant Record with Tag Value: 2
        F5_CASE_2:      False
DBG> ex r.f5_case_1
%DEBUG-E-NOACCESSR, no read access to address 000000007F46E0FC
DBG> ex r.f5_case_2
%DEBUG-E-NOACCESSR, no read access to address 000000007F46E0FC
```

The debug engineering team is aware of this problem.

6. Using 64-bit Pointer Types

VSI Pascal includes support for 64-bit pointers. Using the [QUAD] attribute on pointer types, the compiler will create and use a 64-bit pointer instead of a 32-bit pointer. The NEW and DISPOSE procedures have been enhanced to work with 64-bit pointers.

6.1. Language Features Not Supported With 64-bit Pointers

Several language features are not supported with 64-bit pointers. These features are:

- Base types of 64-bit pointers cannot contain file types.
- The READ, READV, and WRITEV builtin routines cannot read or write into variables accessed via 64-bit pointers. For example, the following code fragment will be rejected by the compiler:

  ```
  var quad_ptr : [quad] ^integer;

  begin
  new(quad_ptr);
  read(quad_ptr^);
  end
  ```

  You can work-around this by using a temporary variable, as in:

  ```
  var quad_ptr : [quad] ^integer;
      tmp : integer;

  begin
  new(quad_ptr);
  read(tmp);
  quad_ptr^ := tmp;
  ```

```
end
```

- VSI Pascal only understands 32-bit descriptors as defined by the OpenVMS Calling Standard. VSI Pascal constructs that rely on descriptors are not supported for variables accessed via 64-bit pointers. The features rejected for 64-bit pointers are:
  - The use of %DESCR or %STDESCR on actual parameter values accessed via 64-bit pointers. For example, you cannot do the following:

    ```
    type
        s32 = packed array [1..32] of char;
    var
    qp : [quad] ^s;

    begin
    new(qp);
    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.
  - At run-time, the compiler will generate incorrect code when passing a VAR parameter that is accessed via 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] ^s32;

    procedure a( p : packed array [l..u:integer] of char );
      begin
      writeln(p);
      end;

    procedure b( var p : s32 );
      begin
      a(p);   { This will generate a bad descriptor }
      end;

    begin
    new(qp);
    b(qp^);
    end;
    ```

6.2. Using 64-bit Pointers With System Definition Files

The STARLET Definition files for Pascal have not been enhanced to reflect the new 64-bit pointer support in the compiler. For routines that have parameters that are 64-bit pointers, the Pascal definition will use a record type that is 64-bits in size. The definition files do not know about either the INTEGER64 datatype or 64-bit pointers. We will try to improve the definition files in a future release.

However, you can still use these new routines from VSI Pascal.

By using a foreign mechanism specifier (i.e., %IMMED, %REF, %STDESCR, and %DESCR) on an actual parameter, you can override the formal definition inside of definition files.

For example, here is an example of calling lib$get_vm_64 using %ref to override the definition from PASCAL$LIB_ROUTINES.PEN. Note, that the NEW predeclared routine will call lib$get_vm_64 directly. However, this example is demonstrating how to override any system parameter definition using 64-bit pointers.

```
[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$LIB_ROUTINES.PAS contains
!  the following definitions for LIB$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.
```

7. Creating Files With RMS "Undefined" Record Types

On OpenVMS IA-64 and OpenVMS x86-64, object files are created with the RMS record type of "undefined" to correctly describe these files as pure byte stream files. Prior to that, the "undefined" record type was not commonly seen on OpenVMS systems.

The VSI Pascal OPEN predeclared routine does not have direct support for "RECORD_TYPE := UNDEFINED". However, you can create a file with "undefined" record type using a USER_ACTION routine. An example file (SYS$SYSROOT:[SYSHLP.EXAMPLES.PASCAL]CREATE_UDF_FILE.PAS) has been included in the VSI Pascal kit that shows how to do it.

8. Using the POS Attribute With Natural Alignment

The description of the POS attribute does not describe the interaction with using the POS attribute with field types whose natural preferred alignment conflicts with the bit position specified.

The description of the POS attribute will be expanded to include the additional rule:

- In an unpacked array, the specified bit position must not conflict with the default preferred alignment of the field's type.

9. Using $FILESCAN With VSI Pascal

The definition of $FILESCAN in STARLET.PAS declares the 1st parameter as a value parameter accepting a string expression. The system service returns pointers back into this parameter via the item list parameter.

When a string variable is passed to this first parameter, the compiler may make a local copy of the string before calling the system service. The system service will then return addresses back into this temporary copy of the string on the stack. As the program executes further, this stack space is reused and the returned pointers become useless.

If the actual parameter passed to $FILESCAN is a PACKED ARRAY OF CHAR variable, then you can workaround this problem by using the %STDESCR foreign mechanism specifier on the actual parameter. If the actual parameter is a VARYING OF CHAR or STRING, you will have to build your own local descriptor referencing the .BODY of the VARYING OF CHAR or STRING and pass that descriptor to $FILESCAN using the %REF foreign mechanism specifier.

In both cases, you need to add VOLATILE to the variable being passed so the compiler knows that the variable must remain active after $FILESCAN has been called. This is needed since the code will be accessing pieces of the variable via the returned addresses.

10. Using Condition Handlers That Return SS$_CONTINUE

In VSI Pascal, condition handlers can do one of the following things after doing whatever is appropriate for the error:

1. Use a non-local GOTO to transfer control to a label in an enclosing block.

2. Return SS$_CONTINUE if the handler wants the error dismissed and to continue processing.

3. Return SS$_RESIGNAL if the handler wants the system to continue searching for additional handlers to call.

4. 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's condition handler that was established with the ESTABLISH builtin routine.

The RTL's handler contains a check to prevent a user's 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 are two situations where this check may cause unexpected behavior.

The first situation is where the user's handler called $UNWIND and then returned with SS$_CONTINUE. Since the $UNWIND service was called, execution won't resume at the point of the error even if SS$_CONTINUE is returned to the system. However, the RTL's handler isn't aware that $UNWIND has been called and complains that you cannot continue for this type of error. The solution is to return SS$_RESIGNAL instead of SS$_CONTINUE after calling $UNWIND in the user's handler.

However, this solution isn't possible if you establish the LIB$SIG_TO_RET routine with the ESTABLISH builtin routine. LIB$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$SIG_TO_RET". Since LIB$SIG_TO_RET returns SS$_NORMAL which in turn is the same value as SS$_CONTINUE, the handler in the Pascal RTL will complain that you cannot continue from this type of error. The solution for this case is to establish your own handler with the ESTABLISH builtin 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 builtin routine.

The second situation where the RTL's check for SS$_CONTINUE from a user's handler can cause problems when moving code from OpenVMS VAX to OpenVMS Alpha, OpenVMS IA-64, or OpenVMS x86-64.

On OpenVMS VAX, only certain run-time errors were not allowed to return SS$_CONTINUE from a handler. These errors were those associated with the SUBSTR and PAD builtin routines as well as checking code for set constructors. On OpenVMS Alpha, OpenVMS IA-64, and OpenVMS x86-64, many more run-time errors are not allowed to return SS$_CONTINUE from a handler. The exact lists of run-time errors which can be continued and which ones cannot be continued has never been provided. The compilers may choose to generate different code in the future which might move an error from one list to the other. We recommend that do you not return SS$_CONTINUE for any Pascal run-time error that is not due to a file operation.

11. Using Asynchronous RMS I/O With Pascal

The USER_ACTION parameter on OPEN and CLOSE provides access to the underlying FAB and RAB RMS blocks that are used by the Run-Time Library for the Pascal FILE variable.

Setting the FAB$V_ASY or RAB$V_ASY flags to enable asynchronous file or record operations is not supported and should not be done. In general, the RTL assumes synchronous RMS file operations and is not prepared to use the $WAIT service when needed. For example, if an asynchronous write is performed, a subsequent STATUS builtin will almost certainly not return the status of the 'in flight' I/O operation. In addition, performing asynchronous operations on non-sequential files will almost certainly not work as expected. Since the RTL assumes synchronous operations, it does not use AST completion routines that would normally be used with asynchronous file operations.

If you wish to use asynchronous file RMS file operations, you should call the RMS services directly.

12. Known Problems and Restrictions

Here is a list of language features that are not yet implemented or do not work as documented in VSI Pascal for OpenVMS x86-64 systems.

1. Some forms of run-time checking enabled with **/CHECK** do not detect the errors.

2. Using PROCEDURE parameters in conjunction with non-local GOTOs can result in a memory leak in the Pascal RTL and LIBRTL. This issue is fixed in E9.2-4.

3. The compiler does not correctly handle some non-local GOTOs when uplevel variable access is also used. [NS9373]

4. Using TO BEGIN DO in a MODULE will cause an ACCVIO. [NS9045]

```
MODULE P;
TO BEGIN DO
```

```
BEGIN
    IF TRUE THEN RETURN;
END;
END.
```

One visible difference is that expressions involving UNSIGNED8 and UNSIGNED16 values are performed with overflow detection enabled (just like any INTEGER expression). So if you multiply the largest UNSIGNED16 value with itself, the operation will produce an overflow where you would not expect an overflow. For example:

```
VAR A,B : UNSIGNED16 VALUE 65535;

A := A * B;
```

will produce an integer overflow where you would expect the rules for unsigned arithmetic to produce the value 1.

5. Using Discriminated Schema as Formal Discriminant Types

   Extended Pascal allows a discriminated ordinal schema type to be subsequently used as the type of a formal schema discriminant. For example:

```
TYPE SUBR(L,U:INTEGER) = L..U;
     DSUBR = SUBR(expression,expression);
     SCH1(D:DSUBR) = ARRAY [1..D] OF INTEGER;
     SCH2(D:DSUBR) = RECORD
                      CASE D OF
                      1: (F1:INTEGER);
                      2: (F2:CHAR);
                     END;
```

   VSI Pascal does not currently support this construct.

6. Files with Schema Components

   Extended Pascal allows file components to contain schematic items and therefore provide run-time sized file components. For example:

```
TYPE SUBR(L,U:INTEGER) = L..U;
     FILE_COMP = ARRAY [SUBR(expr,expr)] OF INTEGER;

VAR  F : FILE OF FILE_COMP;
```

   VSI Pascal does not currently support this feature and requires that the component sizes of files be known at compile-time.

7. Using Formal Discriminants Inside Initial State Specifiers

   Extended Pascal allows the formal discriminant to appear in an initial state specifier in the schema definition. For example:

```
TYPE R(D:INTEGER) = RECORD
                     F1 : INTEGER VALUE D;
                     END;
     A(D:INTEGER) = ARRAY [1..D] OF INTEGER VALUE [OTHERWISE D];
```

   VSI Pascal does not currently support this feature and requires that all initial state values be compile-time expressions.

8. Changing Variants When Selector Is A Discriminant

   If a formal discriminant is used as a variant tag, it is illegal to change the variant once the variable has been created. For example:

```
TYPE R(D:INTEGER) = RECORD
                    CASE D OF
                    1: (ONE : INTEGER);
                    2: (TWO : INTEGER);
                    OTHERWISE (OTHERS : BOOLEAN);
                    END;

VAR V : R(1);

BEGIN
V.TWO := 2; { Is illegal since it changes variants from 1 to 2 }
END;
```

   VSI Pascal does not currently generate run-time checking code to detect this violation.

9. The AND_THEN and OR_ELSE Boolean operators do not short-circuit when used in constant expressions. All constant expressions currently do full evaluation in a left-to-right order. For example:

```
CONST X = FALSE AND_THEN (1 DIV 0 = 0);
```

   This example will currently generate a compilation error instead of correctly defining the constant X to be the value FALSE.

10. The VSI Pascal compiler currently allows you to use the SIZE function on TIMESTAMP variables. As in the case for file variables, these types are abstract objects and the compiler should not permit assumptions about their size to be used.

11. When the buffer variable of a file is passed as a VAR parameter, the allocation size of the formal VAR parameter must match that of the components of the file. Failure to do so will result in an Internal Compiler Error. For example:

```
PROGRAM A;
VAR
   F : PACKED FILE OF 0..65535;
   G : FILE OF [WORD] 0..65535;

PROCEDURE P( VAR I : INTEGER ); EXTERNAL;

BEGIN
P(F^); { causes an Internal Compiler Error }
P(G^); { causes an Internal Compiler Error }
END.
```

12. The LSE templates for the Pascal language are shipped as part of DECset. These templates have not yet been updated to reflect all the new language features added to VSI Pascal.

13. The HELP file has undergone extensive revision which included renaming some topic strings. This will cause problems with using the language-sensitive help feature from LSE.

14. Due to an interaction with the OpenVMS Linker and OpenVMS Image Activator, unknown results will occur if a compilation unit places a file variable in a [COMMON] block and then a shareable image is made from the object file. To share data in a shareable image, use environment files or use the [GLOBAL]/[EXTERNAL] attributes.

15. When using the SYS$SYSTEM:PASCAL$SET_VERSION.COM command file to select older versions of the VSI Pascal, the older compiler may issue an error if the **/VERSION** qualifier is specified. Normally, older compilers will simply ignore any qualifiers that were added after its release. However, due to the implementation of **/VERSION**, older compilers will issue errors while trying to process other qualifiers. If you receive these errors, do not use **/VERSION** to determine the older compiler's version. You will need to look in the first line of a listing file or in the **ANALYZE/IMAGE** output.

16. Jumping into a WITH statement with a GOTO statement may result in an Internal Compiler Error if compiled with **/OPTIMIZE**. Such programs are illegal in nature as bypassing the prologue of the WITH statement skips around the code that precomputes the address of the record used in the WITH statement.

17. The ALIGNED attribute only allows up to 8192 byte boundaries at present (i.e., ALIGNED(13)). Support for larger alignments will be considered for a future release if there is customer demand.

18. Incomplete support for INTEGER64/UNSIGNED64; Currently, the following uses of the INTEGER64/UNSIGNED64 datatypes are unsupported:

    - Literals requiring more than 32-bits cannot be used in the declaration section. This implies that you cannot write such things as:

      ```
      CONST
        BigNum = 12345678912345678;
      ```

      but you CAN write things such as:

      ```
      i64 := 12345678912345678;
      ```

    - INTEGER64/UNSIGNED64 expressions cannot be used as case selectors or variant record tags.
    - Subranges requiring more than 32-bits cannot be declared (since you cannot specify literals large enough to construct them).
    - Array index types cannot be INTEGER64/UNSIGNED64 (since you cannot specify subranges of them).
    - The predeclared constants MAXINT64 and MAXUNSIGNED64 are not present. However, you can use LOWER(INTEGER64), LOWER(UNSIGNED64), UPPER(INTEGER64), and UPPER(UNSIGNED64) in an executable section to obtain the same values.
    - INTEGER64/UNSIGNED64 types cannot be used in variable typecasts.

19. 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.

    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 instead. For example:

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

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

13. STARLET Definition Files

The contents of STARLET.PAS and other definition files provided during the VSI Pascal installation are derived or extracted from a file provided by the OpenVMS system.

The Pascal files are produced during the OpenVMS build process and are shipped compressed inside SYS$LIBRARY:STARLETPAS.TLB. The Pascal installation extracts the Pascal source files, compiles them, and places them and the precompiled environment files on the system. The SYS$LIBRARY:STARLETPAS.TLB file is not used once the installation is finished.

Programs that provide their own Pascal version of system constants, data structures, or entry points may have to be modified if these items are provided by the OpenVMS system in the future. For example, the CLI$_ constants are now being provided by the system.

14. Compiling for Optimal Performance

The following command line will result in producing the fastest code from the compiler.

```
PASCAL /NOZERO_HEAP /OPTIMIZE /NOCHECK
```

15. Alignment Faults

15.1. Understanding Alignment Faults

The Alpha and IA-64 architectures have rules limiting the use of unaligned data items. These rules allow the underlying implementations to be faster than if they allowed unaligned data accesses.

The x86-64 architecture is much more forgiving about unaligned data accesses. From our studies, the overhead is extremely small. However, much of the following might be interesting to programmers wanting to be as efficient as possible. The various tools like **`MONITOR ALIGN`**, **`ANALYZE/SYSTEM FLT`**, and **`DEBUG SET BREAK`** are not available on OpenVMS x86-64. The release notes for the Alpha and IA-64 versions of Pascal contain information about those commands and how to find/resolve alignment faults.

On Alpha, if a LDWU (load word), LDL (load longword), LDQ (load quadword), STW (store word), STL (store longword), or STQ (store quadword) instruction uses an address that is not a multiple of the size of the data being accessed (ie, word, long, or quadword), an exception is generated. The hardware does not support such loads or stores.

The exception is handled by the Alpha PAL (Privileged Architecture Library) code. Since the PAL code has exclusive access to the machine (it sits between the hardware and OpenVMS), the PAL code can execute multiple instructions to access adjacent longwords or quadwords and perform the unaligned memory fetch or store. It can do so atomically on even a multiple CPU system since it understands OpenVMS page table entries and can prevent other active CPUs from deleting the virtual memory being accessed. After the PAL code is finished, it dismisses the exception and processing continues. Unless requested, the PAL code does not even inform OpenVMS that a fault occurred. The overhead with the PAL code fixing the alignment fault is measurable, but reasonable. There is some context saved to get into and return from the PAL code. The actual fixup of the misaligned data is probably on the order of a dozen instructions or so plus the minimal PAL state save/restore sequence.

Many Alpha applications have had alignment faults for years without any noticeable performance penalty. However some applications are slower without realizing that alignment faults have slowed it down.

On IA-64 the situation is much the same. If a LD2 (load word), LD4 (load longword), LD8 (load quadword), ST2 (store word), ST4 (store longword), or ST8 (store quadword) instruction uses an address that is not a multiple of the size of the data being accessed (i.e., word, long, or quadword), an exception is generated. In some cases, the hardware itself may fixup certain unaligned accesses, but in most cases an exception is raised.

Unlike Alpha, the IA-64 has no PAL code to handle the exception. On OpenVMS IA-64, the exception is handled by the operating system. To ensure that no other active CPUs can delete the underlying memory, OpenVMS has to acquire various memory management spinlocks, save considerable state, etc. before it can execute instructions to access adjacent longwords or quadwords and perform the unaligned memory fetch or store. After OpenVMS is finished, there is overhead to release the spinlocks and restore all the saved state. The combined overhead is considerable. The overhead may be in the thousands or tens of thousands instructions plus the impact of acquiring/releasing spinlocks. If other CPUs are creating/deleting virtual memory, the fixup of the unaligned data access will be delayed further. The reverse is also a problem. If one CPU is processing many alignment faults, other CPUs may be delayed trying to create/delete virtual memory.

15.2. What Does the Compiler Know?

Contrary to popular belief, using the PACKED keyword or the **/ALIGN=VAX** DCL qualifier does not cause alignment faults. In these situations the compiler knows when certain fields are unaligned. Consider the following:

```
var
   r : packed record
       f1 : char;
       f2 : integer;
       end;
```

The compiler knows that field F2 is 1 byte from the beginning of the record. It is a longword integer that is not on a longword memory boundary. When the compiler fetches (or stores) the field, the compiler will not simply use an LDL/LD4 instruction. Instead, it will generate several instructions and either fetch the enclosing quadword and shift/extract the desired longword or use smaller instructions like LDB/LD1 and fetch the integer in pieces and combine them together into a register.

In summary, if the compiler can tell at compile-time that a particular fetch or store is unaligned, it will generate several instructions that will not fault instead of a single instruction that always will fault.

Of course, executing 6 or 8 instructions will be slower than executing 1 instruction, but you are avoiding the alignment fault. If the data becomes aligned by removing the PACKED keyword, removing the **/ALIGN** qualifier, using explicit POS attributes, etc. then additional performance could be achieved. However, this has nothing to do with alignment faults. See the *Pascal User Manual* [https://docs.vmssoftware.com/vsi-pascal-for-openvms-user-manual/] or SYS$HELP:PASCAL_RECORD_LAYOUT_GUIDE.MEM for additional information on changing the layout of record fields for additional performance.

15.3. What Does the Compiler Assume?

Alignment faults can occur when the compiler assumes something that is actually not true. In Pascal, this involves either dereferencing pointers or accessing parameters.

The Pascal compiler assumes that pointers point to memory that is at least quadword aligned. The NEW builtin and the underlying LIB$GET_VM library routine provide aligned memory. Consider the following:

```
type rt = record
          f1 : integer;
          f2 : integer;
          end;

var p : ^rt;

begin
new(p);
p^.f1 := p^.f2;
end
```

The compiler assumes that since the pointer is aligned that the F1 and F2 fields are also aligned. The compiler will generate LDL/LD4 instructions to fetch the fields. If the pointer variable is assigned a value that is not quadword aligned, the assumption the compiler made is now false. This unaligned pointer will cause alignment faults to occur when the application runs.

There are several ways for a pointer to get an unaligned value. One common way is to use the IADDRESS predeclared routine to assign the pointer instead of using NEW. With IADDRESS it is possible to place a non-quadword aligned value into the pointer. Another way would be to share the pointer with a non-Pascal routine which did not know the Pascal compiler's rules.

The Pascal compiler assumes that parameters passed by reference point to variables that are aligned on their appropriate boundary depending on whether VAX or NATURAL alignment was requested. Consider the following:

```
type rt = record
          f1 : integer;
          f2 : integer;
          end;

procedure a(var p : rt);
   begin
   p.f1 := p.f2;
   end;
```

The Pascal compiler assumes that the parameter passed to routine A is aligned on a longword boundary if compiled with the default **/ALIGN=NATURAL** or just aligned on a byte boundary if compiled with **/ALIGN=VAX**. If routine A is called with an address of an argument that is not properly aligned, the assumption the compiler made is now false. This unaligned parameter will cause alignment faults to occur. This situation can occur when routine A is called from a non-Pascal routine or if the caller used the IADDRESS predeclared routine or a typecast to override Pascal's type system.

16. Problems Corrected Since Last Release of VSI Pascal

- The compiler did not generate the correct debug DWARF information for some Pascal string types. To get proper behavior, you also need the debugger from OpenVMS Version 9.2-2 + Update 1.
- The compiler would sometimes ACCVIO when using the **/DEBUG** qualifer.
- The compiler would generate an assertion if the same name was as both a routine name and a static variable in another routine. For example:

  ```
  module bug;
  ```

```
[global] procedure x; begin end;
[global] procedure y; var x : [static] integer; begin end;
end.
```

- The compiler would overwrite the stack when making the local copy of a VARYING OF CHAR parameter.

```
type pstr = varying [1812] of char;
procedure test(p1 : pstr); begin writeln(p1); end.
```

- The compiler would sometimes generate include pc-line DWARF information with "line 0" information. This confused TRACEBACK into printing messages like:

```
Error: traceback pc= 0000011f was not found
```

- The compiler would generate an assertion with using **/DEBUG** on a MODULE that contained no code.

```
[ENVIRONMENT] MODULE BUG;
CONST C = 13;
END.
```

- The compiler would sometimes ACCVIO when using **/DEBUG** on a program that uses %INCLUDE files which declare types and variables in different source scopes.

- Calling a routine that uses the [TRUNCATE] attribute may result in the wrong number of arguments passed to the target routine. For example, the following program will get an ACCVIO.

```
PROGRAM Trunc001 (OUTPUT);
VAR
    A : CHAR := '1';
    B : CHAR := '2';
    C : INTEGER := 0;

PROCEDURE Values (A : CHAR;
                  B : [TRUNCATE] CHAR;
                  C : [TRUNCATE] INTEGER);
    BEGIN
    IF PRESENT (A) THEN WRITE ( A  ,'-');
    IF PRESENT (B) THEN WRITE ( B  ,'-');
    IF PRESENT (C) THEN WRITE ('3!');
    WRITELN;
    END;

BEGIN
Values(A);
Values(A,B);
Values(A,B,C);
END.
```

- The compiler would sometimes generate incorrect code for non-local GOTOs when optimization and checking is enabled.

- The compiler would generate incorrect DWARF for certain bit fields with explicit size attributes.

```
type
  rec = packed record
        f4 : [pos(2),bit(3)] 0..7;
        f5 : [pos(9),bit(7)] 0..127;
```

```
        f6 : [pos(17)] integer;
        end;
```

- The compiler would generate incorrect DWARF for packed SET types.

```
type chsubt = 'a'..'f';
var  pschsub : packed set of chsubt;
begin
pschsub := ['a','d','f'];
end
```

- The compiler would mis-compile a program that uses a nested routine that has the same name as an outer-level routine.

```
$ type levelmod.pas
[environment('levelmod')] module levelmod(output);
[global] procedure printit; begin writeln('printit inside module
 levelmod'); end;
end.
$ type level.pas
[inherit('levelmod')]
program level0(input,output);

procedure level1;
 procedure printit; begin writeln('printit inside of level1'); end;

begin printit; end;

procedure level2;
 procedure printit; begin writeln('printit inside of level2'); end;
begin printit; end;

begin
printit;  ! From levelmod environment
level1;
level2;
end.
```

- The compiler would generate incorrect DWARF for BOOLEAN variables with explicit size attribute.

```
type byte_bool = [byte] boolean;
     word_bool = [word] boolean;
```

- The compiler would generate an "invalid tag" internal message when generating DWARF information for certain SET types.
- Tracebacks from Pascal code would occasionally cause the TRACEBACK utility to print "`Error: traceback pc= xxxxxxxx was not found`" due to incomplete DWARF information.
- Pascal integer overflow checking when assigning large unsigned numbers into INTEGER variables was missing or incomplete.
- The Pascal installation has been converted from the old VMSINSTAL mechanism to the PCSI **`PRODUCT INSTALL`** mechanism. See the section in the release notes for more information.
- The debug information for variant records was incorrect. The output is now correct when using the **`DEBUG EXAMINE/VARIANT`** command.

- The debug information for routine prologue/epilogue has been improved for better stepping behavior.
- The compiler would generate incorrect code when assigning into an odd-sized field in a record parameter passed by value.

```
REC_UML = [ALIGNED(3)] RECORD
     UM_UID : INTEGER ;
     UM_FZG : [BYTE] UNSIGNED8 ;
     UM_LIN : [BYTE(3)] 0..9999 ;

PROCEDURE PROC_VALUE(UML : REC_UML) ; { Value Parameter }
BEGIN
    WITH UML DO
    BEGIN
        WRITELN('PROC_VALUE :                          ', UM_LIN, HEX(UM_UID)) ;
        UM_LIN := 1234 ;
        WRITELN('Values should be 1234 and FFFFFFFF : ', UM_LIN, HEX(UM_UID)) ;
        WRITELN ;
    END ;
END ;
```