VSI Pascal X6.3-139 for OpenVMS x86-64 Systems
Release Notes

17 August 2023

This document contains information about VSI Pascal X6.3-139 including new features, differences between X6.3-139 and previous versions, corrections included, and other topics. This file is of interest to both system management and application programmers.

CONTENTS

CHAPTER 1

VSI PASCAL X6.3-139 FOR OPENVMS X86-64 SYSTEMS RELEASE NOTES

1.1 Overview Of VSI Pascal

This is the first field test release of the VSI Pascal compiler for OpenVMS x86-64 systems. It is the same frontend as used on othe OpenVMS platforms to ensure maxiumum language compatibility. It uses LLVM as the code-generator.

The VSI Pascal compiler requires OpenVMS V9.2-1 or later.

1.2 VSI Pascal For OpenVMS X86-64 Systems Field Test

The following features of VSI Pascal are currently not available on OpenVMS x86-64 systems.

- The QUADRUPLE is not supported and will result in internal compiler errors. Unlike C and Fortran, there is no command line qualifier to downgrade QUADRUPLE to DOUBLE. We expect to add QUADRUPLE support in an upcoming release and well as a command line option to downgrade QUADRUPLEs to DOUBLEs

- The /MACHINE_CODE qualifier is ignored. We are still working on getting the generated code listing out of LLVM. In the meantime, you can do ANALYZE/OBJECT/DISASSEMBLE and append the output to the compiler listing file

- Non-local GOTOs do not currently work

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

1.3 New Features In VSI Pascal

The VSI Pascal compiler contains the following new features not listed in the documentation based on customer requests. Unless specified, the following new features have been added to all VSI Pascal OpenVMS targets.

1. The /USAGE=64BIT_TO_DESCR keyword as been added to 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. This new option will suppress the check.

2. The /CDD_QUAD_TYPE=keyword DCL qualifier has been added to control how the %DICTIONARY directive translates quadword and octaword sized items from the CDD Dictionary.

   The keywords available are:

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

   2. 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.

   3. 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.

3. The /IDENT=ident-or-string DCL qualifier has been added to specify a module-ident from the command line. This qualifier is identical to the [IDENT(quoted-string)] module-level attribute already available in the language. An explicit [IDENT(quoted-string)] attribute in the source file will override the /IDENT DCL qualifier. /IDENT=ABC will yield an

ident string of ABC. /IDENT="abc" will yield an ident string of abc.

4. The /PEN_CHECKING_STYLE=keyword DCL qualifier has been added to specify the desired environment file checking method from the command line. This qualifier is identical to the [PEN_CHECKING_STYLE(keyword)] module-level attribute already available in the language. It accepts the same keywords as the attribute: COMPILATION_TIME, IDENT_STRING, and NONE. The default is COMPILATION_TIME. An explicit [PEN_CHECKING_STYLE(keyword)] attribute in the source file will override the /PEN_CHECKING_STYLE DCL qualifier.

5. The /ASSUME=BYTE_ALIGNED_POINTERS keyword has been added to specify that the compiler should assume that all pointers point to memory that is only aligned on byte boundaries.

   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 pointers 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.

6. The IN and NOT IN operators have been enhanced to accept string arguments to determine if the string on the left hand side of the operator is IN, or NOT IN the string on the right hand side, respectively. This enhancement can be used to make programs easier to understand.

   The

   ```
   string1 IN string2
   ```

   expression is identical to

   ```
   INDEX(string2,string1) <> 0
   ```

   and the

```
   string1 NOT IN string2
```

expression is identical to

```
   INDEX(string2,string1) = 0
```

7. The SUBSTR predeclared function has been enhanced to allow the third operand, the substring size, to be optional. When omitted, the SUBSTR function will return the string that starts at the start index and continues to the end of the string.

   The

```
   SUBSTR(string,start_index)
```

   is identical to

```
   SUBSTR(string,start_index,length(string)-string_index+1)
```

8. The %FLOAT directive has been added to determine the floating point default. The %FLOAT directive expands to either "VAX_FLOAT" or "IEEE_FLOAT" depending on the setting of the /FLOAT DCL qualifier or the [FLOAT] module-level attribute.

9. Several directives have been added to create floating literals of a specific type regardless of the current default floating type of the module.

   - %F_FLOAT - produce a VAX F_floating literal
   - %D_FLOAT - produce a VAX D_floating literal
   - %G_FLOAT - produce a VAX G_floating literal
   - %S_FLOAT - produce an IEEE S_floating literal
   - %T_FLOAT - produce an IEEE T_floating literal

   The syntax is:

```
   %x_FLOAT floating-point-literal
```

   where "x" is 'F','D','G','S', or 'T'.

   VSI Pascal has had the ability to declare and use both VAX and IEEE floating point types in the same module using the predeclared types, F_FLOAT, D_FLOAT, G_FLOAT, S_FLOAT, and T_FLOAT. However, floating point literals were always parsed

based on the default floating type in the module. With the addition of the new directives, you can now write code like:

```
lib$wait(%f_float 1.0);
```

which will pass the correct floating literal to LIB$WAIT regardless of the default floating type of the module.

10. The BIN, OCT, HEX, DEC, and UDEC builtins are now supported in compile-time expressions. However, while the BIN, OCT, and HEX builtins accept expressions of any time when used in run-time expressions, they only support ordinal types when used in compile-time expressions.

11. Two new statements named SELECT and SELECTONE have been added. Patterned after the BLISS statements of the same name, the SELECT and SELECTONE statements look much like the CASE statement except for one very powerful feature. Namely, the labels of a SELECT or SELECTONE statement can be run-time expressions as opposed to the CASE statement which only allows constant expressions.

    The syntax for the SELECT statement is:

```
SELECT select-selector OF
[[{{select-label-list},...: statement};...]]
[[ [[OTHERWISE {statement};...]]
   [[ALWAYS {statement};...]] ]]
[[;]]
END
```

    where 'select-label-list' is

```
expression [[.. expression]]
```

    The expressions in the 'select-label-list' can be full run-time expressions.

    When two expressions are provided as a lower and upper bound, they must be of the same ordinal type. There is no check to ensure that the lower bound expression is less than or equal to the upper bound expression. If that occurs then there are simply no values of the select-selector that can be in the range.

    The SELECT statement checks to see if the value of the select-selector is contained in the select-label-list. If so, then the corresponding statement is executed. The select-label-lists are checked in the same lexical order that they appear in the source file. The same value can appear in more than one select-label-list.

The optional OTHERWISE and ALWAYS clauses can appear in either order. The ALWAYS clause is always executed. The OTHERWISE clause is executed only if none of the prior statements (except for an optional ALWAYS statement) have been executed.

The syntax for the SELECTONE statement is almost identical but does not provide for an ALWAYS clause.

```
SELECTONE select-selector OF
[[{{select-label-list},...: statement};...]]
[[ OTHERWISE {statement};... ]]
[[;]]
END
```

Unlike the SELECT statement, the SELECTONE statement stops processing after it executes the first statement that corresponds to a select-label-list that contains the select-selector value.

While the SELECT/SELECTONE statements can be used similar to the CASE statement. For example,

```
SELECT expression OF
 1: WRITELN('ONE');
 2: WRITELN('TWO');
 OTHERWISE WRITELN('not ONE or TWO')
END
```

a more subtle (and powerful) form uses the Boolean constant 'TRUE' as the select-selector. For example,

```
SELECTONE True OF
 expression < 10: WRITELN('Value is small');
 expression < 100: WRITELN('Value is medium');
 expression < 1000: WRITELN('Value is big');
 OTHERWISE WRITELN('Value is too big');
END

SELECTONE True OF
 expression = "AAA": writeln('String is AAA');
 expression = "BBB": writeln('String is BBB');
 expression = "CCC": writeln('String is CCC');
 OTHERWISE writeln('unknown string');
END

FOR i := 1 TO 10 DO
  SELECT True OF
   ODD(i): WRITELN('value ',i:1,' is odd');
   (i MOD 3) = 0:
           WRITELN('value ',i:1,' is also a multiple of 3');
   END;
```

1.4 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 optmizations enabled, but the impact of optimization techiques make debugging difficult.

This support for debugging schema types exposes several bugs in the existing debugger especially on OpenVMS I64 and OpenVMS x86-64. If you try to debug schema types with the existing debugger, you may encounter wrong answers or debug errors. We

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.

1.5 Limited Support For 64-bit Pointer Types

VSI Pascal has been enhanced to provide support for 64-bit pointers. By 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.

1.5.1 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
```

- Since VSI Pascal only understands 32-bit descriptors as defined by the OpenVMS Calling Standard, any VSI Pascal construct that relies on descriptors is 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

```
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. We will add a run-time check for this situation in a future release. 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;
```

1.5.2 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 (ie, %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 starting with Pascal V5.9, 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.
```

1.6 Creating Files With RMS "Undefined" Record Types

On OpenVMS I64 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.

1.7 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.

1.8 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.

1.9 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 problem in moving code from OpenVMS VAX to OpenVMS Alpha, OpenVMS I64, or OpenVMS x86-64.

On OpenVMS VAX, only certain run-time errors were not allowed to return SS$_CONTINUE from a handler. These errors for those associated with the SUBSTR and PAD builtin routines as well as checking code for set constructors. On OpenVMS Alpha, OpenVMS I64, and OpenVMS x86-64, many more run-time errors are not allowed to return SS$_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.

1.10 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.

1.11 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

1. UNSIGNED8 and UNSIGNED16 Predeclared Subranges

   The UNSIGNED8 and UNSIGNED16 predeclared subrange types are documented as being subranges of UNSIGNED. However, they are actually subranges of INTEGER with positive values that correspond to an UNSIGNED subrange of the same size. This subtle distinction in the definition is almost impossible to detect from a program and shouldn't be a problem in the general case.

   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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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 definiting the constant X to be the value FALSE.

7. 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.

8. 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.
```

9. 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.

10. 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.

11. 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.

12. 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.

13. 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.

14. The ALIGNED attribute only allows upto 8192 byte boundaries at present (ie, ALIGNED(13)). Support for larger alignments will be considered for a future release if there is customer demand.

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

    a) 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;
```

    b) INTEGER64/UNSIGNED64 expressions cannot be used as case selectors or variant record tags.

    c) Subranges requiring more than 32-bits cannot be declared (since you cannot specify literals large enough to construct them).

    d) Array index types cannot be INTEGER64/UNSIGNED64 (since you cannot specify subranges of them).

    e) 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.

f) INTEGER64/UNSIGNED64 types cannot be used in variable typecasts.

16. 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;
```

1.12 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.

On OpenVMS Alpha systems prior to V8.2, the Pascal files are converted from binary data shipped by OpenVMS (SYS$LIBRARY:STARLETSD.TLB). The Pascal installation extracts the binary information, converts it to Pascal source files, compiles them, and places the precompiled environment files on the system. The SYS$LIBRARY:STARLETSD.TLB file is not used once the installation is finished.

On OpenVMS x86-64, OpenVMS I64 and OpenVMS Alpha V8.2 and later, 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 been provided by the system.

Finally, be aware that OpenVMS Alpha and OpenVMS I64 systems may not provide the exact same set of definitions.

1.13 Compiling For Optimal Performance

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

1.14 Alignment Faults

1.14.1 Understanding Alignment Faults

The Alpha and Itanium 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.

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 noticable performance penalty. However some applications are slower without realizing that alignment faults have slowed it down.

On Itanium 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 (ie, 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 Itanium has no PAL code to handle the exception. On OpenVMS I64, 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 tring to create/delete virtual memory.

1.14.2 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 achived. However, this has nothing to do with alignment faults. See the Pascal User Manual or SYS$HELP:PASCAL_RECORD_LAYOUT_GUIDE.MEM for additional information on changing the layout of record fields for additional performance.

1.14.3 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.

1.14.4 Correcting Alignment Faults

You can tell the Pascal compiler that certain pointers may point to unaligned data by adding the [ALIGNED(0)] attribute to the pointer declaration. For example,

```
type p_to_aligned_int   = ^ integer;
     p_to_unaligned_int = ^ [aligned(0)] integer;
```

For pointers with type p_to_aligned_int, the compiler will use longword instructions to fetch/store memory. For pointers with type p_to_unaligned_int, the compiler will use additional instructions to reference memory that would not generate an alignment fault.

Alternatively, the /ASSUME=NOBYTE_ALIGNED_POINTERS DCL qualifier causes the compiler to assume that all pointers may point to unaligned memory. This qualifier is a quick way to reduce alignment faults, but does sacrifice performance by making all pointer references use additional instructions.

For unaligned parameters, you would place an [ALIGNED(0)] attribute on the formal parameter definition. For example,

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

procedure a(var p : rt);
   begin
   p.f1 := p.f2; ! assume longword alignment
```

```
   end;

procedure b(var p : [aligned(0)] rt);
   begin
   p.f1 := p.f2; ! assume only byte alignment
   end;
```

1.14.5 Locating Alignment Faults

There are several ways to determine if your application is experencing alignment faults.

All the following examples are done using the following program:

```
program foo;
type pint = ^integer;

procedure a(p : pint);
  begin
  p^ := p^ + 1;
  end;

var p : pint;
    buf : array [1..10] of char;

begin
p::integer := iaddress(buf[2]);
while true do
  a(p);
end.

$ PASCAL/NOOPT/DEBUG FLT
$ LINK/DEBUG FLT
```

1.14.5.1 MONITOR ALIGN (OpenVMS I64 Only)

The MONITOR DCL command has been enhanced to include an ALIGN option. It gives a system-wide view of alignment faults. You cannot determine which process or image is generating alignment faults with this command.

1.14.5.2 ANALYZE/SYSTEM FLT Extension (OpenVMS I64 Only)

The ANALYZE/SYSTEM utility has an alignment fault extension named FLT. You can use FLT to collect alignment fault data over a period of time. You can then examine the fault information to determine which process and more importantly which PC caused the alignment fault. For example,

```
$ analyze/system

OpenVMS system analyzer

SDA> FLT
Alignment Fault Tracing Utility FLT commands:

        FLT LOAD
        FLT UNLOAD

        FLT START TRACE [/BUFFER=pages]
                        [/BEGIN=pc_range_low]
                        [/END=pc_range_high]
                        [/MODE=(K,E,S,U)]          (default is ALL modes)
                        [/INDEX=pid]               (default is ALL processes)
        FLT STOP  TRACE
        FLT SHOW  TRACE [/SUMMARY]

SDA> FLT LOAD
FLT$DEBUG load status = 00000001
SDA> FLT START TRACE
Tracing started...
SDA> ! wait sufficient time to collect meaningful data
SDA> FLT STOP TRACE
SDA> FLT SHOW TRACE
SDA> FLT UNLOAD
FLT$DEBUG unload status = 00000001
```

Please remember to FLT UNLOAD when finished with your analysis. When FLT is loaded, even if it isn't actually collecting data, it will prevent other fault alignment tools from operating properly.

When analyzing the FLT output, the FLT SHOW TRACE command will show all the alignment faults that occurred during the trace (within the limit of the buffer controlled by /BUFFER=pages). Each line shows the faulting PC as well as the EPID. You can also use the SHOW TRACE /SUMMARY to find the faulting PC with the highest fault count although that does not show the EPID value. Once you determine the EPID of the process you are interested in, you can use the following commands to determine the image being run:

```
SDA> SET PROCESS/INDEX=xxx
SDA> SHOW PROCESS/CHANNEL
SDA> SHOW PROCESS/IMAGE
```

The image being run is usually the 2nd channel listed in the output.

Once you have set the process, the FLT SHOW TRACE/SUMMARY will show the correct module name in the image.

With the image name from SHOW PROCESS/CHANNEL, the module name from FLT SHOW TRACE/SUMMARY, and the faulting PC, you have enough information to find the instruction and ultimately the source statement that generated the faulting instruction. EXAM/INSTRUCTION 'faulting-pc' can show the instructions if the image is still in memory. Using the link map from the image, you can determine which source module contributed code for that PC value. Subtract the module's base address from the faulting PC to determine the offset in the module. Using the compiler listing file (compiled with /LISTING/MACHINE), you can find the instruction at the offset. The source line number for that instruction can then be found on the right-hand side of the listing file. With the source line number, you can search backwards in the listing file to finally find the source line. It will probably be a pointer dereference or parameter access.

1.14.5.3 PCA SET UNALIGNED (OpenVMS I64 Only)

PCA (Performance Coverave Analyzer) can be used to collect alignment fault information on an image.

```
$ PASCAL/NOOPT/DEBUG FLT
$ LINK/DEBUG FLT
$ DEFINE LIB$DEBUG PCA$COLLECTOR
$ RUN FLT

        PCA Collector Version V4.9


PCAC> SET UNALIGNED
PCAC> GO

$ PCA FLT.PCA

        Performance and Coverage Analyzer Version V4.9

PCAA> TABULATE/UNALIGNED
```

1.14.5.4 DEBUG SET BREAK /UNALIGNED (OpenVMS Alpha And OpenVMS I64 Only)

The symbolic debugger can stop at instructions that generate alignment faults. For example,

```
$ RUN FLT

         OpenVMS I64 Debug64 Version V8.3-009

%DEBUG-I-INITIAL, Language: PASCAL, Module: FOO

DBG> set break /unaligned
DBG> go
%DEBUG-I-DYNLNGSET, setting language PASCAL
Unaligned data access: virtual address = 000000007ACE3B61, PC = 0000000000010071
break on unaligned data trap preceding FOO\FOO\A\%LINE 6+64
     6:   p^ := p^ + 1;
DBG> examine/instruction .pc-1
FOO\FOO\A\%LINE 6+63:               ld4         r10 = [r10] ;;
DBG> examine/source .pc-1
module FOO
     6:   p^ := p^ + 1;
DBG> go
Unaligned data access: virtual address = 000000007ACE3B61, PC = 0000000000010081
break on unaligned data trap preceding FOO\FOO\A\%LINE 7
     7:   end;
DBG> examine/instruction .pc-1
FOO\FOO\A\%LINE 6+79:               st4         [r32] = r10
DBG> examine/source .pc-1
module FOO
     6:   p^ := p^ + 1;
```

1.14.5.5 Alignment Fault System Services (OpenVMS Alpha And OpenVMS I64 Only)

There are several OpenVMS system services that can collect alignment fault information. One of them, SYS$PERM_REPORT_ALIGN_FAULT, will cause alignment faults to generate a message to SYS$OUTPUT. Consider the following programs:

```
$ type enable_align_report.pas
[inherit('starlet')]
program enable_align_report;
begin
$perm_report_align_fault;
end.

$ type disable_align_report.pas
[inherit('starlet')]
program disable_align_report;
begin
$perm_dis_align_fault_report;
end.
```

```
$! Compile and link
$ pascal enable_align_report
$ link enable_align_report
$ pascal disable_align_report
$ link disable_align_report
$! Turn on alignment fault reporting
$ run enable_align_report
$ run flt
%SYSTEM-I-ALIGN, data alignment trap, virtual address=000000007ACE3B61, function
=00000000, PC=0000000000010071, PS=0000001B
%SYSTEM-I-ALIGN, data alignment trap, virtual address=000000007ACE3B61, function
=00000001, PC=0000000000010081, PS=0000001B
%SYSTEM-I-ALIGN, data alignment trap, virtual address=000000007ACE3B61, function
=00000000, PC=0000000000010071, PS=0000001B
^CONTROL-Y^
$ exit
$! Turn off alignment fault reporting
$ run disable_align_report
```

The messages provide the faulting PC. Using the link map and listing files, you can track the faulting PC back to the source line that caused the alignment fault.

1.14.6 Why /USAGE=PERFORMANCE Does Not Help Finding Alignment Faults

The /USAGE=PERFORMANCE DCL qualifier flags variables, record fields, and array elements that are not optimally aligned or optimally sized.

As mentioned earlier, when the compiler knows that an item is not properly aligned, it will generate several instructions to fetch or store the data in pieces instead of using a single instruction which might get an alignment fault. It is these items that are flagged by the /USAGE=PERFORMANCE qualifier. The message is essentially saying: "The compiler had to generate several instructions to avoid an alignment fault. If you would properly align your data, we could generate a single instruction and be even better."

Also mentioned earlier, alignment faults occur when the compiler DOES NOT know when an item is unaligned due to typecasts or explicit pointer manipulation which violate the compiler's assumptions. The compiler thinks the item is aligned properly and will generate single instructions to fetch or store the item. The /USAGE=PERFORMANCE qualifier will never flag such an item as poorly aligned because the compiler believes it is properly aligned.

In summary, properly aligning your data items will help performance by allowing the compiler to generate single memory reference instructions instead of several instructions. In general, the performance gain might be just a few percentage points of improvement although some programs may see higher gains. However, eliminating alignment faults

has a much greater impact on performance.

1.15 Problems Corrected Since Last Release Of VSI Pascal

- The compiler incorrectly rejected valid hex-format UNSIGNED constants.

  ```
  CONST NODE_STOP_TAG = %XFFFFFFF0;

  generated

  %PASCAL-E-UNSEXCRNG, Constant exceeds range of UNSIGNED
  ```

- Various assertions in LLVM and the GEM-to-LLVM converter have been resolved. The Pascal source to trigger these assertions would be too complicated to describe here but operations like string concatention are involved.

  ```
  Instruction does not dominate all uses!
    %63 = getelementptr inbounds i8* %CANDIDATE, i32 %57, !dbg !31
    call x86_64_sysvcc void @"OTS$MOVE"(i8* %62, i64 %61, i8* %63)
  Broken module found, verification continues.

  and

  assert error: expression = PointeeType == cast<PointerType>(Ptr->getType()->getS
  calarType())->getElementType(), in file /llvm$root/include/llvm-project-10/llvm/
  include/llvm/IR/Instructions.h at line 945
  VSI Pascal Fatal Error has occurred
  ```

- The control expression for the %IF conditional compilation directive always evaluated to TRUE.

- The optimizer would incorrectly remove a data structure used by the RTL for processing files with the [KEY] attribute.

- The PRESENT and ARGUMENT_LIST_LENGTH builtins would return a value that is one past the end of the argument list length.