VMS Software

VSI C V7.6-001 for OpenVMS x86-64

Release Notes

Publication Date: November 2024

Operating System: VSI OpenVMS x86-64 Version 9.2-1 or higher

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

VSI C V7.6-001 for OpenVMS x86-64 Release Notes

VMS Software

1. Introduction

This document contains the release notes for the VSI C V7.6-001 compiler for OpenVMS for x86-64 systems, which is a native x86-64 image generating native x86-64 object modules. When installed, the operation and behavior of the compiler is very similar to that of VSI C V7.4 for OpenVMS IA-64.

The native VSI C compiler in this kit identifies itself with a version number, for example:

```
$ CC /VERSION
VSI C x86-64 V7.6-nnn (GEM 50XB9) on OpenVMS x86_64 V9.2-2
$
```

See Section 5, “Known Restrictions in the V7.6 Compiler” for important information about missing or incomplete features in the compiler.

For additional information on the compiler, see also:

- *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/]
- *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/]
- *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]
- Enter the command **HELP CC** at the $ prompt.

2. Prerequisites

The VSI C compiler requires a minimum of V9.2-1 to install and execute. We also strongly encourage you to install the V9.2-2 Update 2 kit as it contains additional debugger fixes.

3. Differences Between the IA-64 C Compiler and the x86-64 C Compiler

This C compiler for x86-64 behaves very much like the current native IA-64 VMS C compiler (V7.4) in terms of command line options, language features, and so on.

3.1. Linkage #pragmas

The "#pragma linkage_ia64" pragma is not supported on x86-64 platforms. The "#pragma linkage_alpha" pragma is treated as the same as "#pragma linkage".

3.2. Builtin Functions

The philosophy for the builtin functions is that most any existing uses of IA-64 builtins should continue to work under x86-64 where possible, but that the compiler will issue diagnostics where it would be preferable to use a different builtin for x86-64. For this reason, the builtins.h header has not been removed nor have any of the IA-64 declarations been conditionalized out. Instead, it contains comments noting which ones are not available (or not the preferred form) for x86-64. Furthermore, a significant number of the __PAL builtins from Alpha systems have been implemented as system services on OpenVMS x86-64 instead of actual compiler builtins, but using an implementation technique that is transparent to source code.

- There is no support for the asm/fasm/dasm intrinsics (actually declared in <c_asm.h>), or any similar mechanism to insert arbitrary sequences of machine instructions into the generated code. The generation of specific machine instructions can only be accomplished using the builtins declared in builtins.h, or by calling functions written in assembly language.
- The functionality provided by the special-case treatment of R26 in an Alpha asm, as in asm("MOV R26,R0"), is provided by a new builtin function: __int64 __RETURN_ADDRESS(void). This builtin produces the address to which the function containing the builtin call will return (the value of R26 on entry to the function on Alpha, the value of B0 on entry to the function on IA-64, or the value from the top of the stack on entry to the function on x86-64). It cannot be used within a function specified to use non-standard linkage or in a varargs function.
- There is no compiler-based support for any of the __PAL calls other than the 24 queue-manipulation builtins. The queue-manipulation builtins generate calls to VMS system services SYS$<*name*>, where <*name*> is the name of the builtin with the leading underscores removed. Any other __PAL calls declared in builtins.h are actually supported through macros defined in the header pal_builtins.h provided in sys$library:sys$starlet_c.tlb. Note that builtins.h contains a "#include <pal_builtins.h>" at the end. Typically, a macro in pal_builtins.h effectively hides a declaration in builtins.h, and transforms an invocation of an Alpha builtin into a call to a system service (declared in pal_services.h) that will perform the equivalent function on OpenVMS x86-64. Two notable exceptions are __PAL_GENTRAP and __PAL_BUGCHK, which instead invoke the x86-64-specific compiler builtin __break2().
- There is no support for the various Alpha floating-point builtins used by the math library (e.g. operations with chopped rounding and conversions).
- Most builtins that take a retry count provoke a warning, and the compiler evaluates the count for possible side effects and then ignores it, invoking the same function without a retry count. This behavior is due to the fact that the IA-64 and x86-64 architectures lack the Alpha-specific retry behavior which is allowed by Alpha load-locked/store-conditional sequences. However, IA-64 and x86-64 do support the retry behavior for __LOCK_LONGRETRY and __ACQUIRE_SEM_LONG_RETRY, since the retry behavior in these builtins involve comparisons of data values, not just load-locked/store-conditional.
- Note that the comments in builtins.h reflect only what is explicitly present in that header itself, and in the compiler implementation. The user should also consult the content and comments in pal_builtins.h to determine more accurately what functionality is effectively provided by including builtins.h. For example, if a program explicitly declares one of the Alpha builtins and invokes it without having included builtins.h, the compiler may issue the BIFNOTAVAIL error regardless of whether or not the functionality might be available through a system service. If the compilation does include builtins.h, and BIFNOTAVAIL is issued, then most likely there is no support for that functionality; but another (remote) possibility is that there is a problem in the version of pal_builtins.h that is being included by builtins.h.

3.2.1. x86-64-Specific Builtins

The builtins.h header file contains a section at the top conditionalized just to __x86_64 with all of the planned support for x86-64-specific builtins. This section includes macro definitions for all of the registers that can be specified with the _getReg, _setReg, _getIndReg, or _setIndReg builtins. Parameters that are const-qualified require an argument that is a compile-time constant.

```
/* Clear and set interrupt flag */
void __clearInterruptFlag();
void __setInterruptFlag();

/* Load and store global descriptor table register */
void __lgdt(__int64 __address);
void __sgdt(__int64 __address);

/* Load and store interrupt descriptor table register*/
void __lidt(__int64 __address);
void __sidt(__int64 __address);

/* Load and store through the FS and GS descriptors */
unsigned char  __readFsByte (unsigned __int64 offset);
unsigned short int __readFsWord (unsigned __int64 offset);
unsigned int  __readFsLong (unsigned __int64 offset);
unsigned __int64  __readFsQuad (unsigned __int64 offset);

void __writeFsByte (unsigned __int64 offset, unsigned char data);
void __writeFsWord (unsigned __int64 offset, unsigned short int data);
void __writeFsLong (unsigned __int64 offset, unsigned int data);
void __writeFsQuad (unsigned __int64 offset, unsigned __int64 data);

unsigned char  __readGsByte (unsigned __int64 offset);
unsigned short int __readGsWord (unsigned __int64 offset);
unsigned int  __readGsLong (unsigned __int64 offset);
unsigned __int64  __readGsQuad (unsigned __int64 offset);

void __writeGsByte (unsigned __int64 offset, unsigned char data);
void __writeGsWord (unsigned __int64 offset, unsigned short int data);
void __writeGsLong (unsigned __int64 offset, unsigned int data);
void __writeGsQuad (unsigned __int64 offset, unsigned __int64 data);

void __invlpg (__int64 __address);

void __atomicAndGsLong (unsigned __int64 offset, unsigned int expression);
void __atomicAndGsQuad (unsigned __int64 offset, unsigned __int64 expression);
void __atomicOrGsLong  (unsigned __int64 offset, unsigned int expression);
void __atomicOrGsQuad  (unsigned __int64 offset, unsigned __int64 expression);
void __atomicIncrGsLong (unsigned __int64 offset);
void __atomicIncrGsQuad (unsigned __int64 offset);
void __atomicDecrGsLong (unsigned __int64 offset);
void __atomicDecrGsQuad (unsigned __int64 offset);
int  __atomicCmpSwapGsLong (unsigned __int64 offset, unsigned int compexpr,
                            unsigned int newexpr);
int  __atomicCmpSwapGsQuad (unsigned __int64 offset, unsigned __int64 compexpr,
                            unsigned __int64 newexpr);

unsigned __int64  __getReg(__Integer_Constant __whichReg);
void              __setReg(__Integer_Constant __whichReg,
                           unsigned __int64   __value);

unsigned __int64  __getIndReg(__Integer_Constant __whichIndReg,
                              __int64            __index);
void              __setIndReg(__Integer_Constant __whichIndReg,
                              __int64            __index,
                              unsigned __int64   __value);

void __break(__Integer_Constant __break_arg);
void __break2(__Integer_Constant __break_code,
            unsigned __int64 __r17_value);
int __prober(__int64 __address, unsigned int __mode);
int __probew(__int64 __address, unsigned int __mode);
__int64 __RETURN_ADDRESS(void);
```

3.3. Default Floating-Point Format

On OpenVMS x86-64 and OpenVMS IA-64 systems, **`/FLOAT=IEEE_FLOAT`** is the default floating-point representation. IEEE format data is assumed and IEEE floating-point instructions are used. There is no hardware support for floating-point representations other than IEEE, although you can specify the **`/FLOAT=D_FLOAT`** or **`/FLOAT=G_FLOAT`** compiler option.

These VAX floating-point formats are supported in the x86-64 compiler by generating run-time code that converts VAX floating-point formats to IEEE format to perform arithmetic operations, and then converting the IEEE result back to the appropriate VAX floating-point format. This behavior imposes additional run-time overhead and some loss of accuracy compared to performing the operations in hardware on Alpha and VAX systems. The software support for the VAX formats is provided to meet an important functional compatibility requirement for certain applications that need to deal with on-disk binary floating-point data.

On OpenVMS x86-64 and OpenVMS IA-64 systems, the default for **`/IEEE_MODE`** is **`DENORM_RESULTS`**, which is a change from the default of **`/IEEE_MODE=FAST`** on Alpha systems. This means that by default, floating-point operations may silently generate values that print as Infinity or NaN (the industry-standard behavior) instead of issuing a fatal run-time error as they would when using VAX floating-point format or **`/IEEE_MODE=FAST`**. Also, the smallest-magnitude nonzero value in this mode is much smaller because results are allowed to enter the denormal range instead of being flushed to zero as soon as the value is too small to represent with normalization.

The conversion between VAX floating-point formats and IEEE formats on the x86-64 architecture is a transparent process that will not impact most applications. All you need to do is recompile your application. As IEEE floating-point format is the default, unless your build explicitly specifies VAX floating-point format options, a simple rebuild for x86-64 systems will use the native IEEE formats directly. For the large class of programs that do not directly depend on the VAX formats for correct operation, this is the most desirable way to build for x86-64 systems.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on an x86-64 system, the compiler automatically generates code for converting floating-point formats. Whenever the application performs a sequence of arithmetic operations, this code does the following:

1. Converts VAX floating-point formats to either IEEE single or IEEE double floating-point formats.
2. Performs arithmetic operations in IEEE floating-point arithmetic.
3. Converts the resulting data from IEEE formats back to VAX formats.

In a few cases, arithmetic calculations might have different results because of the following differences between VAX and IEEE formats:

- Values of numbers represented
- Rounding rules
- Exception behavior

These differences might cause problems for applications that do any of the following:

- Depend on exception behavior

- Measure the limits of floating-point behaviors
- Implement algorithms at maximal processor-specific accuracy
- Perform low-level emulations of other floating-point processors
- Use direct equality comparisons between floating-point values, instead of appropriately ranged comparisons (a practice that is extremely vulnerable to changes in compiler version or compiler options, as well as architecture)

When applied to a compilation that does not contain a main program, the **`/IEEE_MODE`** qualifier does have some effect: it might affect the evaluation of floating-point constant expressions, and it is used to set the EXCEPTION_MODE used by the math library for calls from that compilation. However, the qualifier has no effect on the exceptional behavior of floating-point calculations generated as inline code for that compilation. Therefore, if floating-point exceptional behavior is important to an application, all of its compilations, including the one containing the main program, should be compiled with the same **`/IEEE_MODE`** setting.

Programs containing undefined floating-point behavior, such as assigning a negative floating-point number to an unsigned integer variable, could generate different results when compiled and run on x86-64 systems. The compiler diagnostics and runtime results can both differ. One runtime difference is the case where one system generates an exception, while the other silently produces a result.

3.4. Predefined Macros

The x86-64 compiler predefines a number of macros, with the same meanings as in compilers on prior architectures. These predefined macros are __x86_64 and __x86_64__.

4. Corrections Since Last Release

- Support for "#pragma __inline" was added.
- Support for "#pragma optimize level=0" was added.
- Tail-call optimization was disabled to conform to the Calling Standard.
- Set correct $STATUS on **`/SHOW=FULLPATH`**.
- Fixed debug info for bitfields.
- Fixed optimization breaking setjmp/longjmp/vfork.
- Fixed optimization for equations in non-default rounding mode.

5. Known Restrictions in the V7.6 Compiler

- The **`/IEEE_MODE`** qualifier is not fully functional. The default is for **`/IEEE_MODE=DENORM`**, but options like FAST, UNDERFLOW_TO_ZERO, and so on may not function correctly.
- The C compiler contains partial support for the **`/OPTIMIZE`** qualifier. It does not fully support all of the sub-options from the **`/OPTIMIZE`** qualifier. The degree of optimization will increase in future updates.
- The debug support from the compiler may not correctly describe every possible data type. We expect to improve the debug support in future releases of the compiler and OpenVMS debugger.

- The **`/MACHINE_CODE`** qualifier is currently ignored. As a temporary workaround, you can use the **`ANALYZE/OBJECT/DISASSEMBLE`** command.
- The #dictionary pragma is not currently supported.
- long double

  The long double data type is not yet fully supported. Known issues include compile-time initialization of global/static variables (including structures/unions with long double data types) and calls to math intrinsic functions.
- varargs.h vs stdarg.h

  Due to how the AMD64 ABI Calling Standard is defined, varargs.h is difficult to support. Most Linux platforms do not support it at all. We have tried to retain as much as possible, but strongly suggest that you convert to using <stdarg.h> instead. It will require source modifications, but they will work on Alpha and IA-64 so you can keep common code going forward.
- Variable initializers that contain math exceptions do not properly get signaled by the compiler.
- The **`/CHECK=UNINITIALIZED_VARIABLES`** qualifier is not supported and may cause a run-time error.
- If the **`/FIRST_INCLUDE`** qualifier is used to specify more than one header-file, and the first logical source line of the primary source file spans physical lines (i.e. it either begins a C-style delimited comment that is not completed on that line, or the last character before the end-of-line is a backslash line-continuation character), then the compiler will give an internal error. Workarounds are either to make sure that the first logical line of the primary source file does not span physical lines (e.g. make it a blank line), or to avoid specifying more than one header in the **`/FIRST_INCLUDE`** qualifier (e.g. use a single **`/FIRST_INCLUDE`** header that #includes all of the headers you want to precede the first line of the primary source file).
- When used without optimization the x86-64 compiler processes unused static functions, which results in linker warnings.

  For example, the following program will get link warnings:

  ```
  #include <stdio.h>

  extern int bar(void);

  void main(void)
  {
      puts("Success");
  }

  static int foo()
  {
      return bar(); /* Should not get undefined for bar() */
  }
  ```

  With the x86-64 compiler:

  ```
  $ cc/nooptimize test.c
  $ link test
  %ILINK-W-NUDFSYMS, 1 undefined symbol:
  %ILINK-I-UDFSYM,        BAR
  ```

```
%ILINK-W-USEUNDEF, undefined symbol BAR referenced
    section: $CODE$
    offset: %X0000000000000029
    module: TEST
    file: DKA200:[MYDIR]TEST.OBJ;1
$
```

No warnings are issued during linking on IA-64 for this same test.

- No warnings are issued on case sensitive variables.

  Definitions of variables which differ only by case are not reported. For example, the following program gets warnings from the IA-64 compiler, but the x86-64 compiler does not issue any warning:

```
int xxxx = 1;
int XXXX = 2;
int XxXx = 3;
```

  With the IA-64 compiler:

```
$ cc test.c

int XXXX = 2;
....^
%CC-W-DUPEXTERN, The declaration of "XXXX" will map to the same
external name as the declaration of "xxxx" at line number 1 in
file DKA200:[MYDIR]TEST.C;1.
at line number 2 in file DKA200:[MYDIR]TEST.C;1

int XxXx = 3;
....^
%CC-W-DUPEXTERN, The declaration of "XxXx" will map to the same
external name as the declaration of "xxxx" at line number 1 in
file DKA200:[MYDIR]TEST.C;1.
at line number 3 in file DKA200:[MYDIR]TEST.C;1
$
```

- The compiler does not issue warnings for uninitialized variables.

  References to an uninitialized variable will get a warning with the IA-64 compiler, but not with the x86-64 compiler.

- Some compiler errors are printed only to SYS$OUTPUT, not to SYS$ERROR.

  For example:

```
 $ TYPE TEST.C
// Dummy test to make sure we generate diagnostic
z;
$
$ DEFINE SYS$ERROR E.DOC
$ CC TEST.C

z;
.^
%CC-W-NOTYPES, Declaration has no type or storage class.
at line number 2 in file TEST.C;1
```

```
$ DEASSIGN SYS$ERROR
$ TY E.DOC
$
```

- The compiler does not report invalid references to restricted pointers.

  For the following example test:

  ```
  #include <stdio>
  int * __restrict a;
  int * __restrict b;
  int c = 5;

  void main(void) {
      a = &c;
      b = a;
      *b = 1;
      printf("%d\n", *a);
  }
  ```

  No warning is issued by the x86-64 compiler, whereas the IA-64 compiler does issue a warning:

  ```
  $ cc test.c
  $ cc dev1053.c

      *b = 1;
      ....^
  %CC-W-BADALIAS, Reference through restricted pointer b uses a
  pointer value based on different restricted pointer, a at line
  number 9 in file DISK1:[TEST]TEST.C;1
  $
  ```