

CPU Compatibility Check for OpenVMS

VMS Software, Inc. (VSI)

Boston, Massachusetts, USA

Table of contents

1. CPU Compatibility Check for OpenVMS 3
   - 1.1 Prerequisites 3
   - 1.2 Running the Script 3
   - 1.3 Interpreting the Results 8
2. Known Issues 10
   - 2.1 OSError: Failed to set RWX 10
   - 2.2 Virtual Machines 10

1. CPU Compatibility Check for OpenVMS

OpenVMS x86-64 requires that the CPU supports certain features that are not present in all x86-64 processors.

Run the Python script below in order to determine whether your system supports the necessary features and is capable of running OpenVMS x86-64. The script can be executed on Windows, OS X, BSD, or Linux.

> **Note**
>
> This check is not exhaustive. It only covers the features that can be determined through the use of the CPUID instruction. There are other requirements which cannot be checked this way, so a successful execution of these checks does not guarantee that your system is suitable for running OpenVMS.

1.1 Prerequisites

The system on which you will run the script must have Python version 2.7 or 3 installed. If you plan to run OpenVMS as a virtual machine guest, the check should be performed on the host system, not on a virtual machine guest.

1.2 Running the Script

1. Copy the following script into a `.py` file on your system.

```
# -*- coding: utf-8 -*-
#
#     Quick test to see if a cpu might be capable of running
#     VSI OpenVMS 9.x for x86-64.
#
# Copyright (c) 2020-2026 VMS Software, Inc.
#
# Based on cpuid.py (https://github.com/flababah/cpuid.py)
#
# Copyright (c) 2018 Anders Høst
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of
# this software and associated documentation files (the "Software"), to deal in
# the Software without restriction, including without limitation the rights to
# use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
# the Software, and to permit persons to whom the Software is furnished to do so,
# subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
# FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
# COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
# IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
```

```
# CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

from __future__ import print_function

import platform
import os
import ctypes
import struct
import sys
from ctypes import c_uint32, c_int, c_long, c_ulong, c_size_t, c_void_p, POINTER,
CFUNCTYPE

# Posix x86_64:
# Three first call registers : RDI, RSI, RDX
# Volatile registers         : RAX, RCX, RDX, RSI, RDI, R8-11

# Windows x86_64:
# Three first call registers : RCX, RDX, R8
# Volatile registers         : RAX, RCX, RDX, R8-11

# cdecl 32 bit:
# Three first call registers : Stack (%esp)
# Volatile registers         : EAX, ECX, EDX

_POSIX_64_OPC = [
        0x53,                    # push   %rbx
        0x89, 0xf0,              # mov    %esi,%eax
        0x89, 0xd1,              # mov    %edx,%ecx
        0x0f, 0xa2,              # cpuid
        0x89, 0x07,              # mov    %eax,(%rdi)
        0x89, 0x5f, 0x04,        # mov    %ebx,0x4(%rdi)
        0x89, 0x4f, 0x08,        # mov    %ecx,0x8(%rdi)
        0x89, 0x57, 0x0c,        # mov    %edx,0xc(%rdi)
        0x5b,                    # pop    %rbx
        0xc3                     # retq
]

_WINDOWS_64_OPC = [
        0x53,                    # push   %rbx
        0x89, 0xd0,              # mov    %edx,%eax
        0x49, 0x89, 0xc9,        # mov    %rcx,%r9
        0x44, 0x89, 0xc1,        # mov    %r8d,%ecx
        0x0f, 0xa2,              # cpuid
        0x41, 0x89, 0x01,        # mov    %eax,(%r9)
        0x41, 0x89, 0x59, 0x04,  # mov    %ebx,0x4(%r9)
        0x41, 0x89, 0x49, 0x08,  # mov    %ecx,0x8(%r9)
        0x41, 0x89, 0x51, 0x0c,  # mov    %edx,0xc(%r9)
        0x5b,                    # pop    %rbx
        0xc3                     # retq
]

_CDECL_32_OPC = [
        0x53,                    # push   %ebx
        0x57,                    # push   %edi
        0x8b, 0x7c, 0x24, 0x0c,  # mov    0xc(%esp),%edi
        0x8b, 0x44, 0x24, 0x10,  # mov    0x10(%esp),%eax
        0x8b, 0x4c, 0x24, 0x14,  # mov    0x14(%esp),%ecx
        0x0f, 0xa2,              # cpuid
        0x89, 0x07,              # mov    %eax,(%edi)
        0x89, 0x5f, 0x04,        # mov    %ebx,0x4(%edi)
        0x89, 0x4f, 0x08,        # mov    %ecx,0x8(%edi)
        0x89, 0x57, 0x0c,        # mov    %edx,0xc(%edi)
        0x5f,                    # pop    %edi
        0x5b,                    # pop    %ebx
```

```
        0xc3                    # ret
]

is_windows = os.name == "nt"
is_64bit   = ctypes.sizeof(ctypes.c_voidp) == 8

class CPUID_struct(ctypes.Structure):
    _fields_ = [(r, c_uint32) for r in ("eax", "ebx", "ecx", "edx")]

class CPUID(object):
    def __init__(self):
        if platform.machine() not in ("AMD64", "x86_64", "x86", "i686"):
            raise SystemError("Only available for x86")

        if is_windows:
            if is_64bit:
                # VirtualAlloc seems to fail under some weird
                # circumstances when ctypes.windll.kernel32 is
                # used under 64 bit Python. CDLL fixes this.
                self.win = ctypes.CDLL("kernel32.dll")
                opc = _WINDOWS_64_OPC
            else:
                # Here ctypes.windll.kernel32 is needed to get the
                # right DLL. Otherwise it will fail when running
                # 32 bit Python on 64 bit Windows.
                self.win = ctypes.windll.kernel32
                opc = _CDECL_32_OPC
        else:
            opc = _POSIX_64_OPC if is_64bit else _CDECL_32_OPC

        size = len(opc)
        code = (ctypes.c_ubyte * size)(*opc)

        if is_windows:
            self.win.VirtualAlloc.restype = c_void_p
            self.win.VirtualAlloc.argtypes = [ctypes.c_void_p, ctypes.c_size_t,
ctypes.c_ulong, ctypes.c_ulong]
            self.addr = self.win.VirtualAlloc(None, size, 0x1000, 0x40)
            if not self.addr:
                raise MemoryError("Could not allocate RWX memory")
        else:
            self.libc = ctypes.cdll.LoadLibrary(None)
            self.libc.valloc.restype = ctypes.c_void_p
            self.libc.valloc.argtypes = [ctypes.c_size_t]
            self.addr = self.libc.valloc(size)
            if not self.addr:
                raise MemoryError("Could not allocate memory")

            self.libc.mprotect.restype = c_int
            self.libc.mprotect.argtypes = [c_void_p, c_size_t, c_int]
            ret = self.libc.mprotect(self.addr, size, 1 | 2 | 4)
            if ret != 0:
                print()
                print("Failed to set RWX.")
                print()
                print("The heap cannot be made executable. On Linux, this may be caused
by SELinux.")
                print("Please try temporarily allowing executable heap memory by
issuing:")
                print("    $ setsebool selinuxuser_execheap on")
                print("    $ python vmscheck.py")
                print("    $ setsebool selinuxuser_execheap off")
                print()
                raise OSError("Failed to set RWX")
```

```
        ctypes.memmove(self.addr, code, size)

        func_type = CFUNCTYPE(None, POINTER(CPUID_struct), c_uint32, c_uint32)
        self.func_ptr = func_type(self.addr)

    def __call__(self, eax, ecx=0):
        struct = CPUID_struct()
        self.func_ptr(struct, eax, ecx)
        return struct.eax, struct.ebx, struct.ecx, struct.edx

#import struct
#import cpuid

def cpu_vendor(cpu):
    _, b, c, d = cpu(0)
    return struct.pack("III", b, d, c).decode("utf-8")

def cpu_name(cpu):
    return "".join((struct.pack("IIII", *cpu(0x80000000 + i)).decode("utf-8")
            for i in range(2, 5))).strip()

"""
@param {leaf} %eax
@param {sublead} %ecx, 0 in most cases
@param {reg_idx} idx of [%eax, %ebx, %ecx, %edx], 0-based
@param {bit} bit of reg selected by {reg_idx}, 0-based
"""
def is_set(cpu, leaf, subleaf, reg_idx, bit):
    regs = cpu(leaf, subleaf)

    if (1 << bit) & regs[reg_idx]:
        return "Yes"
    else:
        return "--"

def is_intel_or_amd(cpu):
    if (cpu_vendor(cpu) == "GenuineIntel") or (cpu_vendor(cpu) == "AuthenticAMD"):
        return "Yes"
    else:
        return "--"

def is_intel(cpu):
    if (cpu_vendor(cpu) == "GenuineIntel"):
        return "Yes"
    else:
        return "--"

def is_amd(cpu):
    if (cpu_vendor(cpu) == "AuthenticAMD"):
        return "Yes"
    else:
        return "--"

if __name__ == "__main__":
    cpu = CPUID()

    comp_tests=[is_intel_or_amd(cpu), is_set(cpu, 1, 0, 2, 19), is_set(cpu, 1, 0, 2,
26), is_set(cpu, 1, 0, 3, 4), is_set(cpu, 1, 0, 3, 9), is_set(cpu, 1, 0, 3, 12)]
    comp_status=True
    for i in comp_tests:
        if i=="--":
            comp_status=False
```

```
            break

    print()
    print("OpenVMS 9.x compatibility quick-check")
    print()
    print("Summary:")
    if comp_status:
        if is_intel(cpu)=="Yes":
            print("Your CPU appears to be compatible with OpenVMS 9.2 and up.")
        else:
            print("Your CPU appears to be compatible with OpenVMS 9.2-1 and up.")
    else:
        print("Your CPU is not compatible with OpenVMS.")

    print()
    print("Detailed info:")
    print()
    print("Vendor ID : %s" % cpu_vendor(cpu))
    print("CPU name  : %s" % cpu_name(cpu))
    print()
    print("Necessary for OpenVMS 9.x:")
    print("Intel or AMD CPU : %s" % is_intel_or_amd(cpu))
    print("SSE 4.1          : %s" % is_set(cpu, 1, 0, 2, 19))
    print("XSAVE            : %s" % is_set(cpu, 1, 0, 2, 26))
    print("TSC              : %s" % is_set(cpu, 1, 0, 3, 4))
    print("APIC             : %s" % is_set(cpu, 1, 0, 3, 9))
    print("MTRR             : %s" % is_set(cpu, 1, 0, 3, 12))
    print()
    print("Optional for OpenVMS 9.x:")
    print("PCID             : %s" % is_set(cpu, 1, 0, 2, 17))
    print("X2APIC           : %s" % is_set(cpu, 1, 0, 2, 21))
    print("FSGSBASE         : %s" % is_set(cpu, 7, 0, 1, 0))
    print("MD_CLEAR         : %s" % is_set(cpu, 7, 0, 3, 10))
    print("XSAVEOPT         : %s" % is_set(cpu, 0xd, 1, 0, 0))
```

2. Run the script by executing the following command from the directory where the script is located:

```
python script-file-name.py
```

You should see an output similar to the following:

```
OpenVMS 9.x compatibility quick-check

Summary:
Your CPU appears to be compatible with OpenVMS 9.2 and up.

Detailed info:

Vendor ID : GenuineIntel
CPU name  : Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz

Necessary for OpenVMS 9.x:
Intel CPU : Yes
SSE 4.1   : Yes
XSAVE     : Yes
TSC       : Yes
APIC      : Yes
MTRR      : Yes
```

```
Optional for OpenVMS 9.x:
PCID      : Yes
X2APIC    : Yes
FSGSBASE  : Yes
MD_CLEAR  : Yes
XSAVEOPT  : Yes

Possible future optionals:
TSX-HLE   : Yes
TSX-RTM   : Yes
UMIP      : --
```

1.3 Interpreting the Results

These first lines identify the processor manufacturer and model name.

```
Vendor ID : GenuineIntel
CPU name  : Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
```

The `Necessary for OpenVMS 9.x` section indicates whether the CPU supports the features that are absolutely necessary for OpenVMS to function (Streaming SIMD Extensions 4.1, XSAVE instructions, Time Stamp Counter, Advanced Programmable Interrupt Controller, and Memory Type Range Registers). If any of these do not say `Yes`, OpenVMS will not boot on this machine.

```
Necessary for OpenVMS 9.x:
Intel CPU : Yes
SSE 4.1   : Yes
XSAVE     : Yes
TSC       : Yes
APIC      : Yes
MTRR      : Yes
```

The `Optional for OpenVMS 9.x` section checks for optional processor features that OpenVMS V9.x can use. OpenVMS will offer better performance if running on a processor that supports these features, but they are not strictly necessary. These features are Process Context Identifiers, x2APIC, FS and GS base instructions, invulnerability to MDS, and the XSAVEOPT instruction.

```
Optional for OpenVMS 9.x:
PCID      : Yes
X2APIC    : Yes
FSGSBASE  : Yes
MD_CLEAR  : Yes
XSAVEOPT  : Yes
```

The `Possible future optionals` section indicates support for CPU features that current versions of OpenVMS do not use, but which may be needed in future versions of OpenVMS for better

performance. These are two variants of Transactional Synchronization Extensions and User-Mode Instruction Prevention.

```
Possible future optionals:
TSX-HLE    : Yes
TSX-RTM    : Yes
UMIP       : --
```

2. Known Issues

2.1 OSError: Failed to set RWX

The Python script uses a technique to execute code from the heap. On certain systems, e.g. SELinux, there may be policies to prevent this. Executing the script on these systems may result in an error message similar to the following:

```
OSError: Failed to set RWX
```

If this occurs, you can try executing the script as a root or temporarily disabling heap execution prevention, e.g. on SELinux:

```
$ sudo setsebool selinuxuser_execheap on
```

2.2 Virtual Machines

Running the script on a virtual machine may provide unreliable results. Some hypervisors are aware of which guest OS runs on the virtual machine, and may disable CPU features which that OS does not need.