VSI C V7.4 for OpenVMS Integrity Servers Release Notes

March 24, 2015

© Copyright 2015 Hewlett-Packard Company
© Copyright 2015 VMS Software Inc.

Contents

1 Introduction 1
2 Installation 1
2.1 Multiple Version Support 3
3 Major Differences from Native Alpha Compiler 5
3.1 #pragma linkage 5
3.1.1 Register Name Mapping for #pragma linkage 5
3.1.1.1 Integer Register Mapping 6
3.1.1.2 Floating-point Register Mapping 8
3.1.1.3 Diagnostics 8
3.1.2 New #pragma linkage_<target> 9
3.1.2.1 Register names for #pragma linkage_ia64 9
3.1.2.2 Register names for #pragma linkage_alpha 10
3.1.3 New keyword "standard_linkage" 10
3.2 Builtin Functions 10
3.2.1 Alpha compatibility 10
3.2.2 IA64 specific 12
3.3 Floating Point 15
3.4 Predefined Macros 18
4 Restrictions when using V7.3 18
5 Enhancements, Changes, and Problems Corrected in V7.4 18
6 Enhancements, Changes, and Problems Corrected in V7.3 19
7 Enhancements, Changes, and Problems Corrected in Version 7.2 22
8 Known Problems in V7.4 25
9 Reporting Problems 26

1 Introduction

This document contains the release notes for VSI C V7.4 for OpenVMS Integrity Servers (I64), which is a native I64 image generating native I64 object modules. When installed, the operation and behavior of the compiler is very similar to that of Compaq C V7.1 for OpenVMS Alpha.

The native VSI C compiler in this kit identifies itself as:

VSI C V7.4 for OpenVMS I64

Note

This kit does not provide any header files. All header files that were provided by C compiler kits for previous versions of OpenVMS are now supplied as part of the OpenVMS base system.

For additional information on the compiler, see also:

- HP C User's Guide for OpenVMS Systems
- Enter the command HELP CC at the $ prompt.

For additional information about the VSI C language and its supported library routines, see also:

- HP C Language Reference Manual
- HP C Run-Time Library Reference Manual for OpenVMS Systems

2 Installation

To install VSI C for OpenVMS I64 systems, set the default directory to a writeable directory to allow the IVP to succeed. Then run the PRODUCT INSTALL command, pointing to the kit location. For example:

```
$ SET DEFAULT SYS$MANAGER
$ PRODUCT INSTALL C/SOURCE=node::device:[kit_dir]
```

After installation, these C release notes will be available at:

SYS$HELP:CC.RELEASE_NOTES

SYS$HELP:CC_RELEASE_NOTES.PS

Here is a sample default installation log:

```
$ PRODUCT INSTALL C/SOURCE=DEVO$:[CC.KITTING]

The following product has been selected:
    HP I64VMS C V7.1-155                  Layered Product

Do you want to continue? [YES]

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for
any products that may be installed to satisfy software dependency requirements.

HP I64VMS C V7.1-155: HP C for OpenVMS Industry Standard

    Copyright 2003 Hewlett-Packard Development Company, L.P.

    This software product is sold by Hewlett-Packard Company

    PAKs used: C

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

    Copyright 2004 Hewlett-Packard Development Company, L.P.

      HP, the HP logo, Alpha and OpenVMS are trademarks of
      Hewlett-Packard Development Company, L.P. in the U.S. and/or
      other countries.

      Confidential computer software. Valid license from HP
      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.

Do you want to review the options? [NO]

Execution phase starting ...

The following product will be installed to destination:
    HP I64VMS C V7.1-155                  DISK$ICXXC_SYS:[VMS$COMMON.]
The following product will be removed from destination:
    HP I64VMS C V7.1-145                  DISK$ICXXC_SYS:[VMS$COMMON.]

Portion done: 0%...40%...50%...60%...70%...80%...90%...100%

The following product has been installed:
    HP I64VMS C V7.1-155                  Layered Product
The following product has been removed:
    HP I64VMS C V7.1-145                  Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for HP I64VMS C V7.1-155 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully
```

```
HP I64VMS C V7.1-155: HP C for OpenVMS Industry Standard

    The release notes are located in the file SYS$HELP:CC.RELEASE_NOTES

      for the text form and SYS$HELP:CC_RELEASE_NOTES.PS for the postscript form.
```

2.1 Multiple Version Support

Version 7.3 adds optional support for having multiple versions of the C compiler on your system. It works by appending an ident name to a previously installed compiler and saving it alongside the new compiler from this kit. Users on your system can then execute the sys$system:decc$set_version.com and sys$system:decc$show_versions.com command procedures to select the desired compiler for a given process and to view the list of available compiler versions.

To set this up, have your system administrator run the installation procedure, answering NO to the question about default options:

```
Do you want the defaults for all options? [YES] NO <RET>
```

Then answer YES to the question about making alternate compilers available:

```
Would you like to set up your system for running alternate versions of C? [NO] YES <RET>
```

Users can then execute the decc$set_version.com command procedure with an argument to set up process default logicals that point to alternate compiler versions. For more information on using decc$set_version.com and decc$show_version.com see section: "Enhancements, Changes, and Problems Corrected in V7.3".

Sample installation for multiple-version Support:

```
$ product install c /source=C$:[disk1.cpri.bvbv.bl50.saveset]

The following product has been selected:
    HP I64VMS C V7.3-18                    Layered Product

Do you want to continue? [YES] yes

Configuration phase starting ...

You will be asked to choose options, if any, for each selected product and for
any products that may be installed to satisfy software dependency requirements.

HP I64VMS C V7.3-18: HP C for OpenVMS Industry Standard

    Copyright 2003, 2004-2007 Hewlett-Packard Development Company, L.P.

    This software is sold by Hewlett-Packard Company

    PAKs used: C or C-USER

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

    HP I64VMS VMS V8.3 [Installed]
```

```
    * Configuration options for this referenced product cannot
    * be changed now because the product is already installed.
    * (You can use PRODUCT RECONFIGURE later to change options.)

    Copyright 2003, 2004-2007 Hewlett-Packard Development Company, L.P.

      HP, the HP logo, Alpha and OpenVMS are trademarks of
      Hewlett-Packard Development Company, L.P. in the U.S. and/or
      other countries.

      Confidential computer software. Valid license from HP
      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.


     Multi_Version Support:
     If you would like to set up your system to be able to run different
     versions of the compiler then answer yes. The installation procedure will
     then copy the previously installed C compiler and associated files along
     side the new compiler in this kit with a suffix appended to the name that
     corresponds to the version number.  Users may then execute
     DECC$SET_VERSION.COM to run an alternate version of the compiler
     and DECC$SHOW_VERSIONS.COM to show available versions at anytime
     after this installation.

    Would you like to set up your system for running alternate versions of C? [NO] YES

Do you want to review the options? [NO]

Are you satisfied with these options? [YES]

Execution phase starting ...

The following product will be installed to destination:
    HP I64VMS C V7.3-18                   DISK$ICXXSYS:[VMS$COMMON.]
The following product will be removed from destination:
    HP I64VMS C V7.2-22                   DISK$ICXXSYS:[VMS$COMMON.]

Portion done: 0%...40%...50%...60%...70%...80%...90%...100%

The following product has been installed:
    HP I64VMS C V7.3-18                   Layered Product
The following product has been removed:
    HP I64VMS C V7.2-22                   Layered Product

%PCSI-I-IVPEXECUTE, executing test procedure for HP I64VMS C T7.3-18 ...
%PCSI-I-IVPSUCCESS, test procedure completed successfully

HP I64VMS C V7.3-18: HP C for OpenVMS Industry Standard

    The release notes are located in the file SYS$HELP:CC.RELEASE_NOTES

      for the text form and SYS$HELP:CC_RELEASE_NOTES.PS for the postscript form.

    A startup file SYS$STARTUP:DECC$STARTUP.COM has been provided.
```

```
It contains commands which can be executed after the product install
procedure has been run and at startup to allow for the best compilation
performance.  You may want to invoke this command file from your
system's site-specific start up file.  This command file does not
have to be invoked for correct operation of VSI C.
```

3 Major Differences from Native Alpha Compiler

This native C compiler for I64 behaves very much like the current native Alpha C compiler (V7.1) in terms of command line options, language features, etc. Primary differences are in the support for `#pragma linkage`, builtin functions, default floating-point representation and controls, and predefined macros.

3.1 #pragma linkage

The behavior of this pragma has been changed dramatically to help deal with the large VMS code base that has used the pragma with the assumption that the target platform would be Alpha, adhering to the OpenVMS Calling Standard for Alpha (this pragma does not exist in the VAX compiler). Also, new variants of the pragma have been added that help clarify the platform-dependent nature of what the pragma specifies; these new pragmas can only be used in code that is properly conditionalized to the target platform.

Because a mismatched linkage between caller and callee tends to cause hard-to-diagnose problems at run-time, and because linkage mismatches often occur when calling undocumented interfaces written in a low-level language for which the user may not have source code, the compiler generates a .vms_linkage section that assists the linker in diagnosing conflicts in linkage specifications between function definitions and function calls across compilation units.

3.1.1 Register Name Mapping for #pragma linkage

The `#pragma linkage` directive is inherently platform-dependent because it uses the names of specific target-machine registers and specifies how they are to be treated in a particular call interface. There may be both hardware and calling-standard constraints that conflict with the behavior specified by #pragma linkage. E.g. on Alpha, register R0 is specified in the calling standard as the first function return value register; but on IA64, R0 is a read-only zero-value register that cannot be assigned a different value (similar to R31 on Alpha).

Like uses of the inline asm function on Alpha, uses of this pragma ought to have been coded conditionally for the target system, to be reviewed and modified or removed whenever porting to a different target.

However, in practice it was found that a significant amount of C code exists that uses this pragma unconditionally (or conditionalized on `#ifndef __VAX`) assuming Alpha register names and calling standard conventions. To reduce the number of source code changes actually required to make this code functional for IA64, it was decided that the pragma would be modified for the IA64 target to accept Alpha register names and conventions, and automatically map these whenever possible to specific IA64 registers according to a correspondence established at the calling-standard level. At the same time, a new version of the pragma, `#pragma linkage_ia64` was implemented. This pragma specifically interprets register names as IA64 hardware registers.

When VSI C for I64 encounters the older `#pragma linkage` directive, by default it will emit a SHOWMAPLINKAGE informational message showing the IA64-specific form of the directive, `#pragma linkage_ia64`, with the IA64 register names that replaced the Alpha register names. Ideally, over time each such message should be examined to determine whether or not a special linkage is necessary or desirable when the application is built for OpenVMS I64, and source code changes made along one of the following lines:

- If the linkage is required to meet an externally-imposed interface, change the source to use `#pragma linkage_ia64` as shown by the `SHOWMAPLINKAGE` message, under target-specific conditional compilation (e.g. `#if __ia64`).
- If the purpose of the linkage on Alpha is to improve performance or debugging characteristics, and your source code contains the definition of the called function as well as providing the declaration and linkage used by all callers of the function, then consider whether the linkage as it was mapped by the compiler to the IA64 registers is likely to provide a significant benefit on IA64. Within a `#if __ia64` conditional, you may want to specify a different `linkage_ia64` that would likely give more of a benefit on IA64 than the default mapping from Alpha registers. Or you may decide that there is no significant advantage to using a special linkage on IA64, and specify the new keyword `standard_linkage`. (See Section 3.1.3)

3.1.1.1 Integer Register Mapping The following table shows the mapping that VSI C applies to the (Alpha) integer register names used in `#pragma linkage` directives when they are encountered in VSI C for OpenVMS I64. Note that the six standard parameter registers on Alpha (R16-R21) are mapped to the first six (of eight) standard parameter registers on IA64, which happen to be stacked registers (see Section 3.1.2.1 for the naming convention used for stacked registers). This mapping attempts to maximize source code compatibility for linkages that may be used in existing C code for Alpha.

Alpha	IA64
R0	R8
R1	R9
R2	R28
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R26
R9	R27
R10	R10
R11	R11
R12	R30
R13	R31
R14	R20
R15	R21
R16	R32 (in parameter or result, else ignored)
R17	R33 (in parameter or result, else ignored)
R18	R34 (in parameter or result, else ignored)
R19	R35 (in parameter or result, else ignored)
R20	R36 (in parameter or result, else ignored)
R21	R37 (in parameter or result, else ignored)
R22	R22
R23	R23
R24	R24
R25	R25
R26	no mapping
R27	no mapping
R28	no mapping
R29	R29
R30	R12
R31	R0

3.1.1.2 Floating-point Register Mapping

Alpha	IA64
F0	F8
F1	F9
F2	F2
F3	F3
F4	F4
F5	F5
F6	F16
F7	F17
F8	F18
F9	F19
F10	F6
F11	F7
F12	F20
F13	F21
F14	F14
F15	F15
F16	F8
F17	F9
F18	F10
F19	F11
F20	F12
F21	F13
F22	F22
F23	F23
F24	F24
F25	F25
F26	F26
F27	F27
F28	F28
F29	F29
F30	F30
F31	F0

3.1.1.3 Diagnostics In some cases, the Alpha compiler silently ignores linkage registers, e.g. if a standard parameter register like R16 is specified in a "preserved" option. When compiled for IA64, this will get a MAPREGIGNORED informational, and the SHOWMAPLINKAGE output may not be correct.

If there is no valid mapping to IA64 registers, the NOMAPPOSSIBLE warning is given, and the linkage is ignored.

There are two special situations that can arise when floating-point registers are specified in a linkage.

- Only IEEE-format values are passed in floating-point registers under the OpenVMS Calling Standard for IA64. VAX-format values are passed in

integer registers. Therefore, if a compilation uses VAX-format floating point (/G_FLOAT, /FLOAT=G_FLOAT, or /FLOAT=D_FLOAT command line qualifiers), any `linkage` pragma that specifies floating-point registers will produce an error (E-level diagnostic). Note that this includes the use of floating-point registers in `linkage` options that do not involve passing values, such as the preserved and notused options.

- The mapping of floating-point registers is many-to-one in two cases: Alpha registers f0 and f16 both map to IA64 f8, and Alpha f1 and f17 both map to IA64 f9. A valid Alpha linkage may well specify uses for both f0 and f16, and/or both f1 and f17. Such a linkage cannot be mapped to IA64. But because of the way this situation is detected, the MULTILINKREG warning message that is produced can only identify the second occurrence of an Alpha register that got mapped to the same IA64 register as some previous Alpha register. The actual pair of Alpha registers in the source are not identified, and so the message can be very confusing. E.g. an option like preserved(f1,f17) will get a MULTILINKREG diagnostic saying that f17 was specified more than once.

3.1.2 New #pragma linkage_<target>

There are two new variants of the linkage pragma, `#pragma linkage_alpha` and `#pragma linkage_ia64`. These variants require that register names be specified in terms of the target machine named in the pragma - the register names will never be mapped to a different target. These pragmas produce the UNAVAILPRAGMA informational and are ignored if encountered when compiling code for a target other than the one specified.

3.1.2.1 Register names for #pragma linkage_ia64

For `#pragma linkage_ia64`, valid registers for the preserved, nopreserve, notused, parameters, and result options include integer registers R3 through R12 and R19 through R31, and floating-point registers F2 through F31.

In addition, the parameters and result options also permit integer registers R32 through R39 to be specified, according to the following convention. On IA64, the first eight integer input/output slots are allocated to stacked registers, and thus the calling routine refers to them using different names than the called routine. The convention for naming these registers in either the parameters or result option of a linkage_ia64 directive is always to use the hardware names as they would be used within the CALLED routine: R32 through R39. The compiler automatically compensates for the fact that within the calling routine these same registers are designated using different hardware names.

3.1.2.2 Register names for #pragma linkage_alpha

The rules for allowed uses of register names with `#pragma linkage_alpha` remain as documented for `#pragma linkage` in the native Alpha compiler. It specifies Alpha register names and does no mapping. The only difference between `linkage` and `linkage_alpha` is the behavior when processed for a target machine other than Alpha. For source code compatibility, a future version of the Alpha compiler will add #pragma linkage_alpha, and recognize the new keyword `"standard_linkage"`. (See Section 3.1.3)

3.1.3 New keyword "standard_linkage"

All forms of the linkage directive now accept a new keyword `"standard_linkage"`, which tells the compiler to use the normal linkage conventions appropriate to the target platform, as specified in the calling standard. When standard_linkage is specified, it must be the only option in the parenthesized list following the linkage name. This can be useful to confine conditional compilation to the pragmas that define linkages, without requiring the corresponding use_linkage pragmas to be conditionally compiled as well.

Code that is written to use linkage pragmas as intended, treating them as target-specific without implicit mapping, might have a form like this:

```
#if defined(__alpha)
#pragma linkage_alpha special1 = (__preserved(__r1,__r2))
#elif defined(__ia64)
#pragma linkage_ia64 special1 = (__preserved(__r9,__r28))
#else
#pragma message ("unknown target, assuming standard linkage")
#pragma linkage special1 = (standard_linkage)
#endif
```

Code compiled for IA64 that deliberately relies on the register mapping performed by #pragma linkage should either ignore the SHOWMAPLINKAGE informational, or disable it (`#pragma message disable(SHOWMAPLINKAGE)`)

3.2 Builtin Functions

3.2.1 Alpha compatibility

The philosophy for the builtin functions is that most any existing uses of Alpha builtins should continue to work under IA64 where possible, but that the compiler will issue diagnostics where it would be preferable to use a different builtin for IA64. For this reason, the `builtins.h` header has not removed or conditionalized out any of the Alpha declarations. Instead, it contains comments noting which ones are not available or not the preferred form for IA64. Furthermore, a significant number of the `__PAL` builtins for Alpha have been implemented as system services on OpenVMS I64 instead of actual

compiler builtins, but using an implementation technique that is transparent to source code that calls the builtins on Alpha.

Specific changes:

- 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. 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). It will also be supported in a future version of HP C for Alpha. The 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 IA64). 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 new 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 I64. Two notable exceptions are `__PAL_GENTRAP` and `__PAL_BUGCHK`, which instead invoke the IA64-specific compiler builtin `__break2()`.
- There is no support for the various 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 is because the retry behavior allowed by Alpha load-locked/store-conditional sequences does not exist on the IA64 architecture. The "exceptions" to this are `__LOCK_LONG_RETRY` and `__ACQUIRE_SEM_LONG_RETRY`, because in these cases the retry behavior involves comparisons of data values, not just load-locked/store-conditional.

- The __CMP_STORE_LONG and __CMP_STORE_QUAD builtins produce either a warning or an error, depending on whether or not the compiler can determine that the source and destination addresses are identical. If the addresses can be seen to be identical, the compiler treats it as the new __CMP_SWAP_ form and issues a warning. Otherwise it is an error. The __CMP_SWAP_ forms are not supported in V6.5 of Compaq C for Alpha; but they are intended to be supported in a future update of HP C for Alpha.
- 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. E.g. 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.

3.2.2 IA64 specific

The header file contains a section at the top conditionalized just to __ia64 with all of the planned support for IA64-specific builtins. This includes macro definitions for all of the registers that can be specified to the __getReg/__setReg/__getIndReg/__setIndReg builtins. Parameters that are const-qualified require an argument that is a compile-time constant.

```
/* Intel-compatible */
unsigned __int64    __getReg(const int __whichReg);
void                __setReg(const int __whichReg,
                             unsigned __int64 __value);
unsigned __int64    __getIndReg(const int __whichIndReg,
                                __int64 __index);
void                __setIndReg(const int __whichIndReg,
                                __int64 __index,
                               unsigned __int64 __value);
void __break(const int __break_arg);  /* Native IA64 arg */
void __dsrlz(void);
void __fc(__int64 __address);
void __fwb(void);
void __invalat(void);
void __invala(void);        /* alternate spelling of __invalat */
void __isrlz(void);
void __itcd(__int64 __address);
void __itci(__int64 __address);
void __itrd(__int64 __whichTransReg, __int64 __address);
void __itri(__int64 __whichTransReg, __int64 __address);
void __ptce(__int64 __address);
void __ptcl(__int64 __address, __int64 __pagesz);
void __ptcg(__int64 __address, __int64 __pagesz);
void __ptcga(__int64 __address, __int64 __pagesz);
void __ptri(__int64 __address, __int64 __pagesz);
void __ptrd(__int64 __address, __int64 __pagesz);
void __rsm(const int __mask);
void __rum(const int __mask);
void __ssm(const int __mask);
void __sum(const int __mask);
void __synci(void);
__int64 /*address*/ __thash(__int64 __address);
__int64 /*address*/ __ttag(__int64 __address);

/* These Intel _Interlocked intrinsics will be added to Alpha. */
unsigned __int64 _InterlockedCompareExchange_acq(
                        unsigned int       *__Destination,
                        unsigned __int64  __Newval,
                        unsigned __int64  __Comparand);
unsigned __int64 _InterlockedCompareExchange64_acq(
                        unsigned __int64 *__Destination,
                        unsigned __int64  __Newval,
                        unsigned __int64  __Comparand);

unsigned __int64 _InterlockedCompareExchange_rel(
                        unsigned int       *__Destination,
                        unsigned __int64  __Newval,
                        unsigned __int64  __Comparand);
unsigned __int64 _InterlockedCompareExchange64_rel(
                        unsigned __int64 *__Destination,
                        unsigned __int64  __Newval,
                        unsigned __int64  __Comparand);
```

```
/* GEM-added builtins */

void __break2(__Integer_Constant __break_code,
              unsigned __int64 __r17_value);
void __flushrs(void);
void __loadrs(void);
int __prober(__int64 __address, unsigned int __mode);
int __probew(__int64 __address, unsigned int __mode);
unsigned int __tak(__int64 __address);
__int64 /*address*/ __tpa(__int64 __address);
/*
** The following were added to the GEM compiler for IA64, but will
** also be implemented for Alpha.
**
** Nota Bene:
**   _Interlocked* built-ins return the old value and have the
**   newval and comparand arguments in a different order than
**   __CMP_SWAP* built-ins that return the status (1 or 0).
**   Forms without trailing _ACQ or _REL are equivalent to
**   the _ACQ form.  On Alpha, _ACQ generates MB after the swap,
**   _REL generates MB before the swap.
*/
int __CMP_SWAP_LONG(volatile void *__addr,
                    int __comparand,
                    int __newval);
int __CMP_SWAP_QUAD(volatile void *__addr,
                    __int64 __comparand,
                    __int64 __newval);

int __CMP_SWAP_LONG_ACQ(volatile void *__addr,
                        int __comparand,
                        int __newval);
int __CMP_SWAP_QUAD_ACQ(volatile void *__addr,
                        __int64 __comparand,
                        __int64 __newval);

int __CMP_SWAP_LONG_REL(volatile void *__addr,
                        int __comparand,
                        int __newval);
int __CMP_SWAP_QUAD_REL(volatile void *__addr,
                        __int64 __comparand,
                        __int64 __newval);

/*
**  Produce the value of R26 (Alpha) or B0 (IA64) on entry to the
**  function containing a call to this builtin.  Cannot be invoked
**  from a function with non-standard linkage.
*/
__int64 __RETURN_ADDRESS(void);
```

3.3 Floating Point

This section describes floating-point behavior on I64 systems.

IEEE Now the Default

On OpenVMS I64 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 I64 compiler by generating run-time code that converts VAX floating-point formats to IEEE format to perform arithmetic operations, and then converts the IEEE result back to the appropriate VAX floating-point format. This 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 I64 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 Intel Itanium architecture is a transparent process that will not impact most applications. All you need to do is recompile your application. Because IEEE floating-point format is the default, unless your build explicitly specifies VAX floating-point format options, a simple rebuild for I64 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 I64 systems.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on an I64 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.

Where no arithmetic operations are performed (VAX float fetches followed by stores), no conversion will occur. The code handles such situations as moves.

VAX floating-point formats have the same number of bits and precision as their equivalent IEEE floating-point formats. For most applications, the conversion process will be transparent and, therefore, a non-issue.

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)

You can test an application's behavior with IEEE floating-point values by first compiling it on an OpenVMS Alpha system using /FLOAT=IEEE_ FLOAT/IEEE_MODE=DENORM.

If that produces acceptable results, then simply build the application on the OpenVMS I64 system using the same qualifier.

If you determine that simply recompiling with an /IEEE_MODE qualifier is not sufficient because your application depends on the binary representation of floating-point values, then first try building for your I64 system by specifying the VAX floating-point option that was in effect for your VAX or Alpha build. This causes the representation seen by your code and on disk to remain unchanged, with some additional runtime cost for the conversions generated

by the compiler. If this is not an efficient approach for your application, you can convert VAX floating-point binary data in disk files to IEEE floating-point formats before moving the application to an I64 system.

/IEEE_MODE Notes

On Alpha systems, the /IEEE_MODE qualifier generally has its greatest effect on the generated code of a compilation. When calls are made between functions compiled with different /IEEE_MODE qualifiers, each function produces the /IEEE_MODE behavior with which it was compiled.

On I64 systems, the /IEEE_MODE qualifier primarily affects only the setting of a hardware register at program startup. In general, the /IEEE_MODE behavior for a given function is controlled by the /IEEE_MODE option specified on the compilation that produced the main program: the startup code for the main program sets the hardware register according the command-line qualifiers used to compile the main program.

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. But 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.

Even on Alpha systems, the particular setting of /IEEE_MODE=UNDERFLOW_ TO_ZERO has the following characteristic: its primary effect requires the setting of a runtime status register, and so it needs to be specified on the compilation containing the main program in order to be effective in other compilations.

Differences in Undefined Behavior

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 I64 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.

More Information

For more information on I64 floating-point behavior, see the white paper *OpenVMS floating-point arithmetic on the Intel Itanium architecture* at http://www.hp.com/products1/evolution/alpha_retaintrust/download/i64-floating-pt-wp.pdf .

3.4 Predefined Macros

The compiler predefines a number of macros, with the same meanings as in the native Alpha compiler, except that it does not predefine any of the macros that specify the Alpha architecture, but instead it predefines the macros __ia64 and __ia64__, as is the practice in the Intel and gcc compilers for IA64. Also note that the change in floating-point representation from G_FLOAT to IEEE is reflected in the macros that are predefined by default. In particular, _IEEE_FP is now set by default (as it is on Alpha for /float=ieee/ieee=denorm).

We note that some users have tried defining the macro __ALPHA explicity with /define or in a header file as a quick "hack" to deal with source code conditionals that were written to assume that if __ALPHA is not defined then the target must be a VAX. Doing this will cause the CRTL headers and other VMS headers to take the wrong path for IA64 - you must not define any of the Alpha architecture predefined macros when using this compiler.

4 Restrictions when using V7.3

- The patch kit VMS821I_LIBOTS-V0100 is required when running this compiler on OpenVMS Integrity 8.2-1 or when running applications on OpenVMS Integrity 8.2-1 that are built with HP C V7.3 for OpenVMS Integrity Servers. The bug fix contained in this OpenVMS Integrity patch kit has already been incorporated into OpenVMS Integrity 8.3, so the patch kit is not needed when running on OpenVMS Integrity 8.3 or higher.

5 Enhancements, Changes, and Problems Corrected in V7.4

VSI C V7.4 is essentially the latest V7.3 compiler rebranded for VMS Software Inc.

6 Enhancements, Changes, and Problems Corrected in V7.3

HP C V7.3 is primarily a bug-fix release of the compiler. One major enhancement that has been added, however, is multiple version support.

The following are compiler enhancements and problems fixed in this version:

- Version 7.3 adds optional support for having multiple versions of the C compiler on your system. It works by appending an ident name to a previously installed compiler and saving it alongside the new compiler from this kit. Users on your system can then execute the sys$system:decc$set_version.com and sys$system:decc$show_versions.com command procedures to select the desired compiler for a given process and to view the list of available compiler versions.

  To set this up, have your system administrator run the installation procedure, answering NO to the question about default options:

  ```
  Do you want the defaults for all options? [YES] NO <RET>
  ```

  Then answering YES to the question about making alternate compilers available:

  ```
  Would you like to set up your system for running alternate versions of C? [NO] YES <RET>
  ```

  Users can then execute the decc$set_version.com command procedure with an argument:

  ```
  $ @sys$system:decc$set_version  V7.2-022
  ```

  Or without an argument:

  ```
  $ @sys$system:decc$set_version

  The following  HP C compiler(s) are available in SYS$COMMON:[SYSEXE]

  Filename                        Version     Defaults
  ------------------------------------------------------------
  DECC$COMPILER.EXE               V7.3-018    System Default
  DECC$COMPILER_S07_01-013.EXE    S7.1-013
  DECC$COMPILER_T07_03-017.EXE    V7.3-018
  DECC$COMPILER_V07_01-011.EXE    V7.1-011
  DECC$COMPILER_V07_02-001.EXE    V7.2-001
  DECC$COMPILER_V07_02-022.EXE    V7.2-022    Process Default

  Enter Version number or SYSTEM: V7.3-018
  ```

Notice that when decc$show_versions.com is executed without an argument, it displays a list of possible compilers and prompts you for a version number. Also notice that you can revert to the installed compiler by selecting SYSTEM as the version number.

The decc$set_version.com command procedure sets up the logicals DECC$COMPILER and DECC$COMPILER_MSG to point to the location of the target compiler and its message file. In addition, it issues a SET command to select the appropriate CDL file to select the correct set of qualifiers for the specified compiler version. Please remember that SET commands are not inherited by subprocesses. Make sure that all subprocesses reissue the necessary decc$set_version.com command procedure.

For a sample installation with multi-version support, please see the installation section.

- A decc$startup.com file was added to the PCSI product install procedure. It contains commands which can be executed after the product install procedure has been run and at startup to allow for the best compilation performance. You may want to invoke this command file from your system's site-specific start up file. This command file does not have to be invoked for correct operation of HP C.

- On rare occassions, previous versions of the compiler could generate code which caused alignment fault traps at runtime when dereferencing a pointer to a struct with longword alignment. This problem has been fixed.

- In rare cases, bad code could be generated for some instances field copies, when compiling /OPT.

- In rare cases, bad code could be generated for some shift operations when compiled /NOOPT. The problem was not evident when compiling /OPT.

- A new option to the /POINTER_SIZE=LONG qualifier is available. When /POINTER_SIZE=LONG=ARGV is specified, the argv argument to main will be comprised of long pointers instead of the short pointers. This can make using long pointers easier as the pointer size of argv will match the default pointer size for the compilation.

NOTE: This is all IPF only. This support has not been added to the Alpha compiler.

- The compiler now gives a RETPARMCONST error diagnostic if a constant is used as the argument to a builtin function parameter that specifies the address of a variable in which to store a result value (e.g. the second parameter of __PAL_INSQHIL). This is because the address of a variable is never a compile-time constant. Previously, specifying a constant could sometimes cause the compiler to crash.

- The compiler now diagnoses the use of excessively large integer values in #line directives. Under C99, an implementation is only required to accept values as large as 2147483647. Larger values now produce a LINETOOLARGE warning. An optional XTRALARGE informational can be requested to report values greater than 32767, which was the C90 requirement.

- The compiler no longer issues spurious warnings for constant expressions within the unevaluated part of a short-circuited constant expression involving the "||" or "&&" operators. Previously, only the ternary "?:" operator in a constant expression suppressed warnings in its unevaluated operand.

- The optional FALLOFFEND diagnostic is now correctly detected and reported in more cases, particulary within functions that are inlined. Programs that previously compiled cleanly with this diagnostic enabled may now report the diagnostic.

- The evaluation of compound literals with side effects could sometimes cause those side effects to occur more than once, depending on the way in which the compound literal was used. For example, when used as an argument to printf, side effects in compound literal arguments could occur three times.

- Some optimization problems that could cause bad code to be generated have been corrected.

- The V7.2-001 release changed the definition of the __STDC__ predefined macro to have a value of 0 when compiled with /STANDARD=RELAXED. This was a change from earlier releases when the value was 1. This change was made because the C Standard defines __STDC__ to be 1 in a conforming implementation, but using /STANDARD=RELAXED makes the compiler non-conforming. After product release, we found that this change caused certain OpenVMS headers to fail. To resolve the issue, this compiler update defines __STDC__ to have a value of 2 when /STANDARD=RELAXED is specified.

- In certain cases the compiler could crash with the following:

```
Code cell integrity check:
Operand literal value out of range:
```

  This problem has been corrected.

- Bad code could be generated for certain field copies, when compiling /OPT.

- Bad code could be generated for certain shift operations when compiled /NOOPT. The problem was not evident when compiling /OPT.

7 Enhancements, Changes, and Problems Corrected in Version 7.2

HP C Version 7.2 is a bug-fix release of the compiler. The following problems are fixed in this version:

- The use of excessively long argument lists (producing the compile-time informational message ARGLISTR255), could in some cases cause the compiler to ACCVIO when the /TIE qualifier was specified. While a compilation that gets one or more ARGLISTR255 diagnostics should no longer access violate when /TIE is specified, if any of the call sites that get the diagnostic end up crossing the boundary between native and translated images, such calls will fail at run-time because the run-time support for calls between native and translated images relies on accurate contents of the argument information register. The ARGLISTR255 diagnostic means that the call requires an argument count value greater than 255, which cannot be represented in the argument information register.

  Also a different, unrelated, situation where /TIE could trigger a compiler assertion failure was fixed.

- Added __fci( ) builtin function to generate the fc.i instruction. The declaration will be added to <builtins.h> in future C RTL headers supplied on the base system. It can be used with an explicit declaration of "extern void __fci(__int64);".

- A new option to the /POINTER_SIZE=LONG qualifier is available. When /POINTER_SIZE=LONG=ARGV is specified, the *argv* argument to `main` is comprised of long pointers instead of short pointers. This can make using long pointers easier because the pointer size of *argv* will match the default pointer size for the compilation.

- #pragma module module-name [module-ident | "module-ident"]

  For consistency, a module-name whether explicit or default considers the /NAMES qualifier and ignores #pragma names.

  if the module-name is too long:

  - A warning is generated if /NAMES=TRUNCATED is specified.
  - There is no warning if /NAMES=SHORTEN is specified.

    A shortened external name incorporates all the characters in the original name. If two external names differ by as little as one character, their shortened external names will be different.

  If the optional module-ident or "module-ident" is too long a warning is generated. A #pragma module directive containing a "module-ident" that is too long is not ignored.

  The default module-name is the filename of the first source file. The default module-ident is "V1.0" They are treated as if they were specified by a #pragma module directive.

  If the module-name is longer than 31 characters:

  - and /NAMES=TRUNCATE is specified, truncate to 31 characters, or less if the 31st character is within a Universal Character Name.
  - and /NAMES=SHORTENED is specified, shorten the module-name to 31 characters using the same special encoding as other external names. Lowercase characters in the module-name are converted to upper case only if /NAMES=UPPERCASE is specified.

A module-ident that is longer than 31 characters is treated as if /NAMES=(TRUNCATED,AS_IS) were applied, truncating it to 31 characters, or less if the 31st character is within a Universal Character Name.

The default module-name comes from the source file name which always appears in the listing header along with a blank module-name and ident. The module-name (and ident) appear in the listing header only if they come from a #pragma module directive or differ from the default. The heading and sub-heading fields of the listing header are not affected.

- In some cases, compiling with the /TIE command line qualifier would cause an internal compiler error. This failure was triggered by source code containing a function call with both of the following characteristics:

  1. One of the arguments was of a struct type, and its position and size were such that part of the struct value was passed in a register and the rest of the value passed on the stack.
  2. That argument was not the last one in the call.

- In some cases, when a #include-file contained a function definition as its first declaration, compiler-generated traceback information was incorrect. If a traceback was generated at run-time, the traceback itself could report the following error:

```
Error: file number is out of range
TRACEBACK - Exception occurred during traceback processing
```

- In some cases, optimized code using the memmove function produced incorrect results when there was overlap between the input and output buffers and the destination was not aligned to a quadword boundary. For example, copying bytes 0-2 of a quadword into bytes 1-3 of the same quadword.
- In some cases, incorrect code could be generated for unaligned access to an unsigned 32-bit longword value, if that value was used in certain operations such as an integer divide.
- The machine_code listing for certain instructions that both read and write a memory location (such as fetchadd) incorrectly showed the read and written operand on the left-hand side of the "=" sign instead of the right-hand side. These kinds of instructions are usually generated only by calls to builtin functions. Not a code generation problem, strictly a listing problem.

- This version of the compiler contains support for generation of a new section type in the object file that improves the ability to identify the source code that corresponds to a shortened CRC'd external name. Future versions of the OpenVMS linker will use this section to emit more understandable error messages when these symbols are undefined and will be able to generate a special section in the linker map file showing the CRC'd name and its corresponding original name.

8 Known Problems in V7.4

- The compiler might emit an erroneous BADANSIALIASn message.

  In some situations the compiler's loop unrolling optimization can generate memory accesses in the code stream that never actually execute at run-time, but that would violate the ANSI aliasing rules if they were to execute. In such a situation, the compiler might emit an erroneous BADANSIALIASn message, where n is a number or is omitted.

  If the violations take place only in machine instructions that will not execute at run-time, these messages can be safely ignored.

  To determine whether or not particular instances of a BADANSIALIASn message are erroneous, recompile the module with the /OPT=3DUNROLL=3D1 qualifier. Any BADANSIALIASn messages that disappear under that qualifier can be safely ignored, so you may want to add appropriate `#pragma message` directives to the source, localized to the specific source lines known to be safe. This is preferable to disabling the message for the whole compilation, since in all other cases the message indicates a real potential for code generation that will not work as intended. And this is generally preferable to disabling the ANSI_ALIAS or loop unrolling optimizations, since that would likely degrade performance, although the amount of degradation is not predictable, and in unusual cases it might even improve performance. As always when making changes to performance-critical code, it is best to measure the impact.

- 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 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).

- This version of the C compiler contains support for a new section type which maps CRC'd external names to their original unencoded form. This will permit future linkers to emit more understandable messages for encoded external names. However, the current version of ANAL/OBJECT on OpenVMS 8.2-1 and earlier will issue an error mesage %ANALYZE-E-ELF_UNKNWNSEC, Unrecognized Elf Section Type 60000007 for C modules containing long names that are compiled /names=shortened. Please ignore this message.

9 Reporting Problems

Please report problems or offer feedback using the mechanisms specified in the "Read Before Installing" document.