HP C V7.3 for OpenVMS Alpha Release Notes

April 3, 2007

Contents

Tables

1 Introduction

This document contains the release notes for HP C V7.3 for OpenVMS Alpha. Note that most online documentation is being provided in html format as well as in its traditional format. The html versions of manuals are provided along with the bookreader versions on the documentation CDROM. The online help files are also available in html format (help cc continues to work in its usual way). The URL to access the online help with a local browser is given at the start of the text-based help cc command. For additional information on the compiler, see also:

- The HP C User's Guide for OpenVMS systems
- Enter the command HELP CC

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

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

The release notes for the HP C Run Time Library are contained in the Run Time Components for OpenVMS Alpha.

2 Installation Notes

2.1 Installation Requirements

HP C V7.3 requires OpenVMS Alpha V7.3-2 or higher.

Following are disk space requirements for installation of HP C for OpenVMS Alpha, Block Cluster Size=1:

	w/o optional documents	with optional documents
Disk space required for installation:	150,000 blocks	250,000 blocks
Disk space required for use (permanent):	100,000 blocks	160,000 blocks

For more information about installing the kit, refer to the HP C Installation Guide accompanying these release notes.

2.2 Header Files

The installation kit will replace the DECC$RTLDEF.TLB in SYS$LIBRARY unless it finds that the creation date of the file on this kit is earlier than the creation date of the existing file on your system. Whenever DECC$RTLDEF.TLB is replaced, the kit will also place reference copies of the *.H forms of the headers in

- SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF]
- SYS$COMMON:[DECC$LIB.REFERENCE.SYS$STARLET_C]

The compiler does not normally search these reference areas, instead it searches for and reads these headers directly from the text library files SYS$LIBRARY:DECC$RTLDEF.TLB and SYS$LIBRARY:SYS$STARLET_C.TLB. The *.H form of the headers are provided as a convenience to users for reference purposes such as searching and browsing, which are not directly supported for text libraries.

2.3 Startup Procedure

HP C for OpenVMS Alpha provides an optional startup procedure, SYS$STARTUP:DECC$STARTUP.COM . This procedure may be invoked by the system startup procedure to perform an install on the compiler and its associated message file to improve compiler performance.

3 Migrating from VAX C to HP C

If you are migrating from VAX C to HP C, you might find the following publication useful:

Compaq C Migration Guide for OpenVMS VAX Systems (Order number: AA-Q5AVA-TE)

This guide is included with the Compaq C for OpenVMS VAX product, to which it primarily applies. However, the following sections of this guide might also prove helpful if you are porting VAX C code to HP C on an Alpha system:

- The following sections in Chapter 1—*Migrating to the Compaq C Run-Time Library*:
  - *Potential Migration Concerns and Solutions*
  - *Behavior Differences Between the VAX C RTL and Compaq C RTL*
  - *Compaq C RTL Obsolete Features*

- *Debugging and the Compaq C RTL Object Library*
- Some subsections of *C RTL Interoperability Concerns*

- All of Chapter 2—*Migrating to the Compaq C Compiler*

For more information on Linking to the HP C RTL on OpenVMS Alpha systems, see the section *RTL Linking Options on Alpha Systems* in Chapter 1 of the *HP C Run-Time Library Reference Manual* included in your product documentation.

For more information on features helpful in migrating to HP C, see the *HP C User's Guide* included in your product documentation. Such features include:

- Command-line qualifiers:

  /STANDARD
  /WARNINGS
  /EXTERN_MODEL
  /[NO]SHARE_GLOBALS

- Preprocessor directives:

  #pragma message
  #pragma extern_model

The HP C User's guide also contains a "Migrating from VAX C" appendix that you might find useful. This appendix summarizes the features that distinguish HP C for OpenVMS Systems from VAX C Version 3.2. You might also want to read the Common Pitfalls appendix of the user's guide.

Note that while the /STANDARD=VAXC qualifier enables a number of language features and behaviors that aid in building programs developed with the VAX C compiler, it is sometimes the case that differences between the two compiler implementations can produce unexpected behavior differences in the compiled program. Also note that the /STANDARD=VAXC qualifier only affects the language dialect the compiler accepts. The HP C compiler has separate qualifiers that control other environmental characteristics (e.g. /extern_model, /share_globals, and /nested_include) that may affect the ease of building a VAX C code base with HP C.

4 Installing and Using Multiple Compiler Versions

HP C V7.3 X7.3-103 provides limited support for installing and using multiple versions of the compiler on the same node.

During installation of V7.3, if a V6.0 or higher version of the compiler is already installed, you will be given the opportunity to preserve that compiler rather than overwrite it. If you choose to preserve the currently-installed compiler, you will then be given an opportunity to keep the currently-installed compiler as the system default and install the new compiler as an alternate. By default, preserving the currently installed system compiler is performed by making it an alternate compiler, and installing the new compiler as the system default.

The choice to use an alternate compiler instead of the installed system compiler can be made by users, by running a command procedure that changes the behavior of the cc command for the process that invokes it.

4.1 Displaying and selecting the compiler version

This kit provides two command procedures to display and control which C compiler is used by a process.

- SYS$SYSTEM:DECC$SHOW_VERSIONS.COM

  This procedure displays what C compilers are available on the system, together with their version numbers. It also displays which compiler is the default for the current process. An example:

```
@SYS$SYSTEM:DECC$SHOW_VERSIONS.COM

The following C compiler(s) are available
in SYS$SYSTEM:

Filename                      Version
-------------------------------------
DECC$COMPILER.EXE             V6.4-005
DECC$COMPILER_V06_04-005.EXE  V6.4-005
DECC$COMPILER_V06_04-006.EXE  V6.4-006
DECC$COMPILER_V06_02-008.EXE  V6.2-008    Process Default
```

- SYS$SYSTEM:DECC$SET_VERSION.COM

  This procedure either sets up process logicals that point to an alternate C compiler in SYS$SYSTEM and it issues a "$SET COMMAND" to a corresponding cld file in SYS$SYSROOT:[SYSHLP.CC$ALPHA_CLD] to establish the valid set of CC command line options, or else it removes the process logicals and does an appropriate "$SET COMMAND" to revert back

to using the default system compiler. The procedure takes one argument, a version number or "SYSTEM" (if no arguments are specified you will be prompted). The SYSTEM argument selects the installed system compiler, which is the one displayed with the filename DECC$COMPILER.EXE in the output of DECC$SHOW_VERSIONS.COM. Alternate compilers are shown in the output of DECC$SHOW_VERSIONS.COM with their version number appended to the simple filename.

Alternate compilers must be located in SYS$SYSTEM and their names must be based upon the compiler version number. For example the V6.2-008 compiler is given the name: "SYS$SYSTEM:DECC$COMPILER_V06_02-008.EXE".

To select a compiler, either pass a full ident string or enough of the ident string to be unique. For example: to select the V6.2-008 compiler from our list above we can pass V6.2-008 or V6.2 to the DECC$SET_VERSION.COM routine. In this example, to select a 6.4 compiler, a full ident string would be required to distinguish between the V6.4-005 and the V6.4-006 compilers.

```
$@SYS$SYSTEM:DECC$SET_VERSION.COM V6.2-008

$ SHOW LOGICAL DECC$COMPILER*

(LNM$PROCESS_TABLE)

"DECC$COMPILER" =
  "SYS$SYSTEM:DECC$COMPILER_V06_02-008.EXE"
"DECC$COMPILER_MSG" =
  "SYS$MESSAGE:DECC$COMPILER_MSG_V06_02-008.EXE"

$ @SYS$SSYTEM:DECC$SET_VERSION 6.4

The following 6.4 Compaq C compiler(s) are available
in SYS$SYSTEM:

Filename                      Version
--------------------------------------
DECC$COMPILER.EXE             V6.4-005
DECC$COMPILER_V06_04-006.EXE  V6.4-006
DECC$COMPILER_V06_04-005.EXE  V6.4-005


Ambiguous version number, please be specify a
  full version number, ex: V6.4-005
Version number : V6.4-005

$ SHOW LOGICAL DECC$COMPILER
   "DECC$COMPILER" =
     "SYS$SYSTEM:DECC$COMPILER_V06_04-005.EXE"
$ SHOW LOGICAL DECC$compiler_msg
   "DECC$COMPILER_MSG" =
     "SYS$MESSAGE:DECC$COMPILER_MSG_V06_04-005.EXE"
```

When this procedure is run in a process, subsequent CC commands invoke the selected compiler version (until the procedure is run again). However, when spawning a process, the DECC$SET_VERSION should be re-issued by the spawned process in order to have the spawned process use the correct values in the DCL command table when processing command line qualifiers. Note that MMS spawns processes, so your MMS files should be modified to include a DECC$SET_COMMAND if you are not using the installed compiler when building. The process-level logicals and the "$SET COMMAND" issued by DECC$SET_VERSION do not affect other processes or users on the system.

4.2 Side effects and restrictions on multiple versions

When you install this kit, it provides the latest DECC$RTLDEF.TLB, and the latest documentation, even if you select the option of having the new compiler as the alternate compiler. The new DECC$RTLDEF.TLB does not adversely impact a preexisting V6-based compiler because they are upwardly compatible.

Beginning with the V6.4A (ident V6.4-006) a set of CLD files for all the officially released compilers from V6.0 onward will be placed in SYS$SYSROOT:[SYSHLP.CC$ALPHA_CLD]. The V6.4A CLD file will be inserted into the default system DCL table ONLY if you select the new compiler as the installed compiler. This is in contrast to the original version of V6.4 (ident V6.4-005) which unconditionally updated the DCL tables with a V6.4 CLD file. This is also in contrast to what is stated in the installation guide for V6.4 (The installation guide was not updated to reflect the changes in V6.4A.)

Because of these differences in how CLD files are handled if you have installed V6.4-005 you should reinstall the compiler version which you wish to be the default compiler in order to get the default DCL tables to match your default compiler, and then V6.4A (or a higher version number) may be installed as an alternate compiler. If you are not able to reinstall an old compiler kit, issue a DECC$SET_VERSION <version-num> as a workaround to get the correct command tables. Alternatively, you may ask the system administrator to update the system DCL tables with the correct CLD file from SYS$SYSROOT:[SYSHLP.CC$ALPHA_CLD].

Please remember that if you spawn a process, your DCL tables are not inherited by the spawned process, even though your logical tables are inherited. If you do not re-run DECC$SET_COMMAND (or alternatively issue a "SET COMMAND") your spawned process will use the default DCL tables.

If you have mismatched CLD files you may see some of the following symptoms:

- If you are accidentally using an old compiler option with a newer CLD file, you will not see an error when a new option is used with an older compiler that does not support it. Instead, the option will be silently ignored. For example, a V6.2 compiler should produce the following error when passed the /first_include qualifier: "%DCL-W-IVQUAL, unrecognized qualifier ...."

```
$cc/ver
  Compaq C V6.2-008 on OpenVMS Alpha G7.3
$cc /first_include foo.c  ! /first_include new in 6.4
$!No complaint
$! use of decc$set_version in your spawed process fixes this
$ @SYS$SYSTEM:DECC$SET_VERSION V6.2
$cc /first_include foo.c
  %DCL-W-IVQUAL, unrecognized qualifier -
  check validity, spelling, and placement
  \FIRST_INCLUDE\
```

- If you attempt to use a new compiler with an older CLD file, you will find that the new compiler options are not accepted.

```
$ cc /ver
  Compaq C V6.4-006 on OpenVMS Alpha G7.3
$ cc /first_include test.c
  %DCL-W-IVQUAL, unrecognized qualifier -
  check validity, spelling, and placement
$! use of decc$set_version in your spawed process fixes this
$ @SYS$SYSTEM:DECC$SET_VERSION V6.4
$ cc /first_include foo.c
$!No complaint
```

- Another type of error that you may see with mismatched CLD files is a %CLI-F-SYNTAX error followed by a traceback. For example:

```
  $ cc /arch=ev6_2 test.c
%CLI-F-SYNTAX, error parsing 'EV67'
-CLI-E-ENTNF, specified entity not found in command tables
%TRACE-F-TRACEBACK, symbolic stack dump follows
 [Tracebacks deleted ]
```

And because you must have the newest CLD file and header files in order to use the newest compiler, if you run an older installation procedure to put an older compiler back on your system, you must then re-run the V6.4A (or higher) installation to get the newest files.

Note that there are two logical names involved in establishing the compiler version - one for the compiler image and one for its message file. This version of the compiler will issue a diagnostic if it is invoked with the wrong version of the message file - but previous versions of the compiler do not detect

this situation. If you find that an older version of the compiler is issuing diagnostics that don't make sense for the code construct they're attached to, or if the message text is missing and only a message number is issued, check that you have matched versions of the files designated by the two logicals using the command "$ show logical decc$compiler*". The response should show matching version-numbered files as in the example selecting the 6.2 compiler. Or if you are using the system compiler, the response should be "%SHOW-S-NOTRAN, no translation for logical name DECC$COMPILER*".

4.3 Installation Procedure Changes

When you install HP C V7.3 on a system that already has a 6.0 or higher compiler installed, you will be given the opportunity to preserve the currently-installed system compiler. To do this, answer yes to the following question (the xxx will be replaced by the full version number of the existing system compiler):

```
"Should the existing xxx system compiler be preserved [NO]:"
```

If you answer no, the installation will procede in the traditional manner, overwriting the currently-installed system compiler.

If you answer yes, you will be asked an additional question. To get the traditional behavior of installing the kit compiler as the system default, answer NO to the question:

```
"Should this xxx system compiler remain the
                        default when cc is typed [NO]:"
```

Since you have previoiusly asked to preserve the existing system compiler, that compiler is made an alternate compiler before installing the new system compiler from the kit. If you answer yes to the question, the kit compiler will be installed as an alternate compiler and the existing system default compiler will remain the default.

4.4 Sample installation fragment

```
        Beginning installation of CC V6.5 at 14:26

%VMSINSTAL-I-RESTORE, Restoring product save set A ...
%VMSINSTAL-I-RELMOVED, Product's release notes moved...

  Compaq C Version V6.5 for OpenVMS Alpha Systems

  Copyright 2002 Compaq Information Technologies Group, L.P.

  Compaq and the Compaq logo are trademarks of Compaq Information
  Technologies Group, L.P. in the U.S. and/or other countries.

  Confidential computer software. Valid license from Compaq required for
...etc...
```

```
  A C V6.2-008 compiler was found on your system.
  Type YES to keep this compiler on your system
  either as the default system compiler, or as an
  alternate compiler.  Type NO to supersede C V6.2-008.

* Should the existing V6.2-008 system compiler
   be preserved [NO]: yes

  Type NO to have the compiler on this kit become the
  default system compiler and to have the currently
  installed compiler saved as an alternate compiler.
  Type YES to keep the current system compiler as the
  default compiler, and to have the compiler on this
  kit available as an alternate compiler.  Alternate
  compilers can be invoked with the cc command after
  invoking SYS$SYSTEM:DECC$SET_VERSION.COM passing
  a version_number.

* Should this V6.2-008 system compiler remain the
   default when cc is typed [NO]: no

        Product:      C
        Producer:     DEC
        Version:      6.5
        Release Date: 01-NOV-2001
```

5 Enhancements and bug fixes

5.1 Enhancements in V7.3

This version is largely a bug-fix release, the list of bugs fixed is given in Section 5.16. There are only two new features, which are intended to make it more command-line compatible with the corresponding C compiler version for I64:

- The /CHECK qualifier now accepts optional keywords ALL and NONE. This is for consistency with similar qualifiers and other VMS compilers. It permits all checks except for named ones to be enabled, without needing to specify the names of all supported checks. E.g. /CHECK=(ALL,NOBOUNDS) enables all run-time checking code except for bounds checking.

- The new I64-only /CHECK=ARG_INFO qualifier is recognized and ignored (with optional QUALNA diagnostic). This is intended to help support maintenance of common build scripts for Alpha and I64.

5.2 Enhancements in V7.1

This version contains the following new features and enhancements:

- New builtins for I64 compatibility.

  The __CMP_STORE_* builtins are not fully functional on I64, and should be considered deprecated. New builtin functions __CMP_SWAP*, _InterlockedCompareExchange*, and __RETURN_ADDRESS have been added for source compatibility with the I64 compiler. See <builtins.h> for prototypes.

  The Alpha architecture has the somewhat unusual concept of a "lock region" in the load-locked/store-conditional paradigm used to implement atomic update sequences. This concept is exposed by the __CMP_STORE_* builtin functions, which have the ability to test a value from one location, and store a new value into a different location in the same lock region with guaranteed atomicity. On IA64, there is no lock region wider than the location being updated, so the __CMP_STORE_* builtins can only be implemented sensibly in the case that the source and destination addresses are identical. That case can be implemented efficiently using the IA64 cmpxchg instructions.

  To promote source compatibility for these kinds of low-level locks, the __CMP_STORE_* builtins are considered deprecated. They continue to work on Alpha, but on I64 they give an error unless the compiler can determine that the source and destination addresses are the same, in which case they give a warning. The most compatible replacements for existing uses of __CMP_STORE_* are the new __CMP_SWAP_* builtins, which have similar prototypes except that the destination parameter has been removed since the source and destination are identical by definition.

  But there are also new builtins with names and signatures matching those provided by Intel's IA64 compiler, _InterlockedCompareExchange*. Instead of returning a status value, these return the value fetched. If it matches the comparison value passed in, then the new value was stored; otherwise the store did not take place and the code has the value that blocked the store (this value cannot be determined when using either the __CMP_SWAP_* or __CMP_STORE_* builtins). There are four variations of this

builtin, two for longword updates (*Exchange_*) and two for quadword updates (*Exchange64_*). All return the old value as unsigned __int64, and take the new value as unsigned __int64, regardless of whether the location to update is longword or a quadword. Also note that the order of the comparand and new_value parameters are reversed relative to the __CMP_STORE_*/__CMP_SWAP_* builtins.

For each update size there are two variations, "_acq" and "_rel", corresponding to IA64 "acquire" and "release" semantics for the cmpxchg instruction. On Alpha, the *_acq forms generate a memory barrier AFTER the conditional store, while the *_rel forms generate a memory barrier BEFORE the conditional store. Note that the __CMP_SWAP_* builtins also have "_ACQ" and "_REL" forms, even though the __CMP_STORE_* builtins do not. The forms of __CMP_STORE_* and __CMP_SWAP_* with neither "_ACQ" nor "_REL" suffix do not generate any memory barrier.

Finally, there is a __RETURN_ADDRESS builtin. On Alpha it returns the value in R26 on entry to the current function, and on I64 it returns the value in B0 on entry to the current function. This builtin cannot be used in functions with non-standard linkage (#pragma linkage).

- #pragma linkage enhancements.

  #pragma linkage is implicitly target-specific because it names machine registers, and interacts with the calling standard on the target machine. However, to assist in porting significant source code bases that used the pragma without conditionalizing the use to Alpha, the I64 compiler recognizes #pragma linkage, assumes it was intended for Alpha, and attempts to map the Alpha machine registers to corresponding IA64 registers under the calling standard for OpenVMS I64. This was done to ease porting, but ideally code using the pragma should have been conditionally compiled for Alpha.

  For use in new code, two variations of the pragma have been added, #pragma linkage_alpha and #pragma linkage_ia64. These pragmas are explicitly defined to be target-specific and are never mapped to a different target - they are ignored with an informational message if encountered on a different target machine than the one they specify.

Also, all three forms of the pragma recognize 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, as shown below.

Code that is written to use linkage pragmas as intended, treating them strictly 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
```

- New keyword TARGET for /MMS_DEPENDENCIES qualifier.

The keyword TARGET may be used with an optional value to specify the name of the "target" in the generated dependency file output. By default, the target name is the simple name of the primary source file, with the file extension changed to .OBJ. If a value is provided for target, then the simple filename and extension parsed from that string is used as the name of the target. As a special case, if the value of TARGET is .OBJ, then the target name will be the simple name and extension of the object module produced by the compilation (which might be controlled by the /obj qualifier). For example:

```
CC/MMS/OBJ=OUTPUT DISK:[DIR]T.C
produces an MMS file with
T.OBJ :

CC/MMS=(TARGET=DISK:[DIR]FOO)/OBJ=OUTPUT T.C
produces an MMS file with
FOO :
```

```
CC/MMS=(TARGET=.OBJ)/OBJ=DISK:[DIR]OUTPUT T.C
produces an MMS file with
OUTPUT.OBJ :
```

- Improved diagnostics for some C99 features including flexible array member (a struct whose last member has an incomplete array type).

- Support enum types containing values beyond the range of type int. Under /stand=relaxed, the compiler now supports this extension commonly used in open source code.

- New diagnostics to report unreferenced labels, with unreferenced case labels spelled the same as a visible enumeration constant reported by default. Others can be enabled by the "questcode" or "unused" message groups.

- Better diagnostic for a mismatched comment delimiter (i.e. the character sequence "*/" appearing outside of a comment, string literal, or character constant).

- Better, more complete and accurate information in the BADANSIALIAS diagnostic.

- The /NAMES=LOWERCASE command line qualifier has no real practical value, as the only treatments of casing for global names on VMS that make sense are UPPERCASE and AS_IS. Therefore this variation is no longer supported - using it produces a warning message, and global names are processed as if /NAMES=AS_IS had been specified.

- The /STANDARD= command line qualifier now enforces the documented behavior that there is no default keyword value supplied when the qualifier is explicitly present. Although this qualifier was not intended to have a default value when specified explicitly, the behavior of previous compilers has been to treat /STANDARD the same as /STANDARD=ANSI89, which is the strict C89 mode. Note this differs from the documented default when /STANDARD is omitted altogether, which is /STANDARD=RELAXED. Beginning with this version, specifying /STARDARD without supplying a keyword will result in "%DCL-W-VALREQ" from DCL, and the compiler will not be invoked.

- The /STANDARD= command line qualifier now accepts a new keyword value, "LATEST". This keyword selects the strict mode of the latest version of the C standard that has been implemented by the compiler. In V7.1, this is equivalent to /STAND=C99.

- The STRCTPADDING diagnostic is an optional diagnostic to detect padding inserted by the compiler between members of a struct to satisfy alignment requirements. The purpose is to alert the user that changing the order of the members could save space. A new optional message, STRCTPADEND, has been added at user request to help detect possible size differences in structs compiled on other platforms.

- Diagnostic messages reported against source code in a header file that is actually contained within a text library module now provide the name of the module in the library as well as the filespec for the library.

- New command line qualifier /ACCEPT=[NO]TRIGRAPHS. Controls whether the compiler recognizes trigraphs, independently of the /STANDARD qualifier, although the /STANDARD qualifier controls the default: COMMON and VAXC modes default to /ACCEPT=NOTRIGRAPHS, all other modes default to /ACCEPT=TRIGRAPHS.

- 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 short pointers. This can make using long pointers easier because the pointer size of *argv* will match the default pointer size for the compilation.

- A command procedure CC$PRODUCT_REMOVE has been added to the HP C V7.1 kit for OpenVMS Alpha. This procedure allows you to remove the HP C compiler product. It performs the equivalent of a PCSI PRODUCT REMOVE command.

  You are required to disable the product license before issuing the command procedure to prevent a compilation from interfering with the delete process.

  If the compiler has been installed as a shared image, the command procedure will uninstall the image.

  The command procedure takes no parameters and can be run as follows:

  ```
  $ @SYS$SYSTEM:CC$PRODUCT_REMOVE

  Do you wish to proceed with removing HP C <No>? Yes [Ret]
  ```

  Enter "Yes" to remove the compiler from your system.

5.3 Enhancements in V6.5

This version contains the following new features and enhancements:

- Uses GEM BL48 backend, with best support for EV7 processors.

- Optional "_nm" suffix can be appended to any #pragma name to prevent macro expansion on that pragma. This is the opposite of the "_m" suffix introduced in V6.4.

- C99 _Pragma operator, which effectively allows pragma directives to be produced by macro expansion. Note: when specified using this operator, the tokens of the pragma, which appear together within a single string literal in this form, are not macro expanded, regardless of any suffix. But macro expansion can be accomplished if desired by using the stringization operator to form the string. For specifics on this and the

other C99 features, the C99 standard is the best source - see Section 5.5 for information on getting a copy.

- C99 constants for specific values of Infinity and NaN are supported (only when using /float=ieee). The underlying implementation-specific identifiers for these constants are:

```
__decc_float_ieee_Infinity
__decc_float_ieee_NaN
__decc_double_ieee_Infinity
__decc_double_ieee_NaN
__decc_long_double_ieee_Infinity
__decc_long_double_ieee_NaN
```

- C99 adjacent string concatenation. Wide and normal strings can be mixed, in which case the normal strings get promoted to wide and a wide result is produced.

- C99 Universal Character Names (UCNs) are accepted in identifiers, string literals, and character constants (and their wide variations).

- New `#pragma include_directory` has been added. The syntax is:

```
#pragma include_directory <string-literal>
```

  The effect of each include_directory pragma is as if its string argument (including the quotes) were appended to the list of places to search that is given its initial value by the /INCLUDE_DIRECTORY qualifier, except that an empty string is not permitted in the pragma form. It is intended to ease DCL command line length limitations when porting applications from POSIX-like environments built with makefiles containing long lists of -I options specifying directories to search for headers. Just as long lists of macro definitions specified by the /DEFINE qualifier can be converted to `#define` directives in a source file, long lists of places to search specified by the /INCLUDE_DIRECTORY qualifier can be converted to `#pragma include_directory` directives in a source file.

Note that the places to search as described in the help text for the /INCLUDE_DIRECTORY qualifier includes the use of POSIX-style pathnames, e.g. "/usr/base", and that this form can be very useful when compiling code that contains POSIX-style relative pathnames in `#include` directives. For example, `#include <subdir/foo.h>` can be combined with a place to search such as "/usr/base" to form "/usr/base/subdir/foo.h", which will be translated to the filespec "USR:[BASE.SUBDIR]FOO.H"

Note that this directive can only appear in the main source file, or in the first file specified in the /FIRST_INCLUDE qualifier. It also must appear before any #include directives.

- New keywords NOCRTL and RESTORE_CRTL have been added to #pragma extern_prefix. These keywords control whether or not the compiler will apply its default RTL prefixing to the names specified on the pragma. The effect of NOCRTL is like that of the except= keyword of the /prefix_library_entries command line qualifier. The effect of RESTORE_ CRTL is to undo the effect of a NOCRTL or a /prefix=except= on the command line.

- /ANNOTATIONS command line qualifier

```
/ANNOTATIONS=(option[,...])   D=/NOANNOTATIONS
/[NO]ANNOTATIONS
```

Controls whether or not the source listing file is annotated with indications of specific optimizations performed or, in some cases, not performed. These annotations can be helpful in understanding the optimization process.

If annotations are requested (and the /LISTING qualifier appears on the command line), the source listing section is shifted to the right and annotation numbers are added to the left of source lines. These numbers refer to brief descriptions which appear later in the source listing file.

Keywords selecting annotation of specific optimizations are:

- ALL - Selects all annotations. This output can be quite verbose, as it includes detailed output for all annotations. For more concise output for each kind of annotation, use /ANNOTATIONS=(ALL,NODETAIL), or just /ANNOTATIONS with no keywords.

- [NO]CODE - Annotates machine code listing with descriptions of special instructions used for prefetching, alignment, etc. Note that /MACHINE_CODE must also be specified in order for this keyword to have any visible effect.
- [NO]DETAIL - Provides additional level of annotation detail, where available.
- [NO]FEEDBACK - Indicates use of profile-directed feedback optimizations. Note that feedback optimizations are not implemented on OpenVMS, so this keyword has no visible effect.
- [NO]INLINING - Indicates where code for a called procedure was expanded inline.
- [NO]LOOP_TRANSFORMS - Indicates optimizations such as loop reordering and code hoisting.
- [NO]LOOP_UNROLLING - Indicates where advanced loop nest optimizations have been applied to improve cache performance (unroll and jam, loop fusion, loop interchange, etc).
- [NO]PREFETCHING - Indicates where special instructions were used to reduce memory latency.
- [NO]SHRINKWRAPPING - Indicates removal of code establishing routine context when it is not needed.
- [NO]SOFTWARE_PIPELINING - Indicates where loops have been scheduled to hide functional unit latency.
- [NO]TAIL_CALLS - Indicates an optimization where a call from routine A to B can be replaced by a jump.
- [NO]TAIL_RECURSION - Indicates an optimization that eliminates unnecessary routine context for a recursive call.
- NONE - is the same as /NOANNOTATIONS.

Specifying /ANNOTATIONS with no keywords is the same as specifying /ANNOTATIONS=(ALL,NODETAIL)

- More aggressive /OPT=INLINE=ALL

The heuristics controlling inlining have been changed in this release to provide overall better performance for the AUTOMATIC, SIZE, and SPEED inlining controls. Because these improvements rely on improvements in the compiler's ability to perform inlining in more situations, a side effect is that /OPT=INLINE=ALL has become more aggressive than it was in previous releases. Consequently, programs that had previously been compiled with /OPT=INLINE=ALL may now cause the compiler to exhaust virtual memory or take an unacceptably long time to compile. /OPT=INLINE=ALL was noted as not recommended for general use when it was introduced in V5.0. That recommendation is even stronger in this release. Programs that were measured as benefitting from /OPT=INLINE=ALL with a previous release, but which no longer can be compiled within reasonable resource limits with this release should generally be changed to use /OPT=INLINE=SPEED.

5.4 Enhancements in V6.4A

This version does not provide new functionality over V6.4, but rather it provides significant compiler bug fixes, usability improvements for multi-version support, and changes to signal.h to ease its use with /pointer_size=long. See Section 5.19 for the bugs fixed. It also uses a slightly newer version of the GEM backend, with miscellaneous tuning changes.

5.5 Enhancements in V6.4

In addition to the support for installing multiple compiler versions, this release provides most language-feature support for the new C99 standard, ANSI/ISO/IEC 9899:1999. This was published by ISO in December, 1999 and adopted as an ANSI standard in April, 2000.

Note that an official copy of the standard can be purchased and downloaded as a PDF file for less than $20US from either NCITS at http://www.techstreet.com/cgi-bin/detail?product_id=232462 or ANSI at http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999.

This release also adds new command line options and pragmas, a new version of the GEM optimizing backend for Alpha, three new C99 header files, and many new functions in the `<math.h>` header.

Note that some of the "new" run-time library support for C99 is available in earlier versions of OpenVMS through extensions that had already been implemented:

- many of the math routines in math.h that are conditionally excluded when compiling in strict ANSI89 mode match functions that were added in the C99 standard

- many library routines for complex data types (other than long double complex) are available because they are the same as used for the Fortran complex data types, and the Alpha calling standard makes them callable from either language.

And although full support for all C99 library features is not anticipated until an OpenVMS release after the "7.3" release currently in field test, the 7.3 field test release does provide the new C99 math library functions that were not already available as extensions, and it supports the long double complex data type.

Important Note: The following three changes will impact existing code.

- The `<math.h>` header supplied on this kit declares prototypes for all of the new C99 math functions, conditionalized to using Compaq C version 6.4 or greater, and to a language mode that sets the macro __STDC_VERSION__ >= 199909. As described later, this macro is set in most language modes, including the default "relaxed_ansi" mode. This is in concert with the next change noted, involving prefixing of C99 entries. If your existing code declares an external identifier that is the same as the identifier for one of the many new math or complex functions added in C99, and your program #includes `<math.h>` (or the new `<complex.h>`), then you will get a compile-time error if your declaration is incompatible with the standard declaration. If your code encounters a compile-time conflict with the declarations in math.h, you should rename the identifier in your code, as it will become a long-term burden to portability. Basically, the C standard specifies that external names declared in standard headers are reserved for that use regardless of whether or not the header is #included. Because the platform will be supporting the C99 standard, and the default "relaxed" language mode of the compiler will enable all of C99, we are forcing the namespace issue at compile-time and with facility prefixing, even though not all functions may be available at link time in currently-shipping OpenVMS libraries.

- The new default of `/PREFIX=C99_ENTRIES` for the default language mode of `/STANDARD=RELAXED` may cause unresolved references at link time if your application provides its own implementation of library functions that have been added to C99. Workarounds are to specify `/PREFIX=ANSI_C89_ENTRIES` explicitly, rename your function (especially if it does not exactly implement the behavior required by C99), or recompile both your function definition and its callers. In the latter case, you will need to remove your own implementation when the supported version of the new function becomes available in the Compaq C RTL.

- The new rules for determining the type of an integer constant could lead to some constants in your program being interpreted as having a signed type when previous compiler versions gave them an unsigned type. This could affect your program's behavior in subtle ways. The new message `intconstsigned` can be enabled to report constants in your source code that are being treated differently under the C99 rules than they were in previous releases. This message is also part of the new message group `newc99`. If your program relied on unsigned treatment, the simple fix is to add the correct suffix including a "U" or "u" to force the constant to have the expected type. Such a change would be backward compatible and portable.

The following specific enhancements were made:

- New C99 hexadecimal form of floating-point constants

  This form of constant permits floating point values to be specified reliably to the last bit of precision. It does not specify a bit pattern for the representation. Instead it is interpreted much like an ordinary decimal floating point constant except that the significand is written in hexadecimal radix, and the exponent is expressed as a decimal integer indicating the power of two by which to multiply the significand. A "P" instead of an "E" separates the exponent from the significand. Thus, for example, 1/2 can be written as 0x1P-1 or 0x.1P3. The C99 standard also adds printf/scanf specifiers for this form of value, but that support will not be present in OpenVMS run-time libraries until after the 7.3 release.

- New C99 header `<stdbool.h>`, and keyword `_Bool`

  This header is intended to be used to access the new C99-specified boolean type _Bool. It defines a macro spelled "bool" that expands to _Bool, intended to be the preferred way to refer to the type. The type is not recognized in VAXC, COMMON, or the strict ANSI89 mode. An object of this type occupies one byte and is an unsigned integer, but its value can only be either 0 or 1. It is permitted to use _Bool as the type for a bit-field. When a value of any scalar type (any arithmetic type including floating point and the new complex types, or any pointer type) is converted to _Bool, the result is zero if the value would compare equal to 0 (e.g. if the pointer is NULL), and otherwise the result is 1. The content of the header is simply as follows:

```
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

- New C99 header <complex.h> and keyword _Complex

  C99 introduces builtin complex data types similar to the Fortran type, in all three precisions (float _Complex, double _Complex, and long double _Complex). The header file <complex.h> defines a macro spelled "complex", intended to be the preferred way to refer to the types. The details of this type can be obtained from the C99 standard and the Language Reference Manual, and by examining the content of the header. Basically the type is similar to the Fortran type in its use. There is no special syntax for constants - instead there is a new keyword "_Complex_I", which has a complex value whose real part is zero and whose imaginary part is 1.0. The header file defines a macro "I" that expands to "_Complex_I", and so a complex constant with equal real and imaginary parts of 2.0 would be written "2.0 + 2.0*I".

  There are some known issues with complex types as follows:

  - The complex data types are not available when using the /float=d_float command line option. This is a permanent restriction.
  - On current versions of OpenVMS, the complex types and functions other than long double complex are available, but the long double complex type is available only in the 7.3 field test.
  - In this version of the compiler, the C99 functions cabs, cabsf, and cabsl cannot be used. This is a temporary restriction. Functions named cabs, cabsf, and cabsl have traditionally been declared in <math.h> using a struct representation to hold two floating values. This is not compatible with the calling standard for passing complex values, or with the implementation of C99 complex data types. In this version of the compiler, the traditional <math.h> declarations are preserved. If you #include <complex.h> in the same compilation as <math.h>, then if <math.h> occurs first you will get an informational message from <complex.h> noting the incompatibility. If you #include <complex.h> prior to the #include <math.h>, then there is no diagnostic, but if you call these functions the generated code will not properly access

C99 versions of them. A workaround is to write your own separately-compiled routines that take a pair of parameters of the corresponding real floating point type, and return sqrt(p1*p1 + p2*p2). These routines can then be called with a complex argument.

- New C99 header `<tgmath.h>`

  This header provides "type-generic" names for 60 math functions that provide operations for a number of different types, letting the actual argument types select the function to call instead of requiring the user to name the exact function and pass it appropriate arguments. E.g. where C89 defined two different names for the sqare root function (sqrt and sqrtl for double and long double arguments) and reserved a third name (sqrtf for float arguments), C99 defines six names because it requires that the float version be implemented and adds three new complex types (csqrt, csqrtl, and csqrtf). These names are all declared in <math.h> and <complex.h> and can be used in the traditional way. But by including this new header (which internally includes both math.h and complex.h), a call to sqrt(x) will be translated to call the appropriate function according to the type of x. For details, see the standard, the documentation, and the contents of the header.

  Known issue:

  - The type-generic implementation of the absolute value function (fabs) is not available for complex types in this release. You must use the type-specific names (cabs, cabsf, cabsl) instead.

- Language modes, `/STANDARD=C99`, message groups, `/PREFIX`

  The compiler's default language mode remains /STANDARD=RELAXED_ ANSI89, which accepts nearly all language extensions as well as standard C89 and C99 features. It excludes only K&R ("common" mode), VAX C, and Microsoft features that conflict with standard C. The /STANDARD=ANSI89 mode continues to implement strictly the 1990 ANSI/ISO C standard (commonly called C89), issuing all required diagnostics as well as a number of optional diagnostics that help detect source code constructs that are not portable under the C89 standard (digraphs from the 1994 Amendment are also recognized in this mode, even though they were not specified in

the 1990 standard). The ISOC94 keyword can still be added to any of the modes (except VAXC) to predefine the macro __STDC_VERSION__, as specified in Amendment 1 to the C89 standard.

A new mode, /STANDARD=C99, has been added that accepts just the C99 language without extensions, and diagnoses violations of the C99 standard. Since C99 is a superset of Amendment 1, and since the default mode of RELAXED_ANSI89 is a superset of C99, the macro __STDC_VERSION__ will now normally be defined with the C99-specified value of 199901L. Only in the case of adding the ISOC94 keyword to the strict ANSI89, MIA, or COMMON modes will the macro take on the Amendment 1 value of 199409L (in the absence of the ISOC94 keyword, these modes do not define the macro at all).

Since the standard is quite new, use of C99 features is not really portable in a practical sense yet. Also, the term "ANSI C" or "standard C" will be ambiguous for some time to come (i.e. do you mean C89 or C99). To help with this situation, the compiler has added three new message groups for messages that report the use of features in the following categories:

```
noc89  - features not in C89
noc99  - features not in C99
newc99 - features that are new in C99.
```

The existing group, `noansi`, which is now somewhat ambiguous in name, is retained as a synonym for noc89.

In recognition of the additional run-time library functions specified in C99, the /PREFIX_LIBRARY_ENTRIES qualifier accepts a new keyword value, C99_ENTRIES. This will enable DECC$ prefixing of all those external names that are specified in C99 (these are a superset of the external names prefixed under /PREFIX=ANSI_C89_ENTRIES and a subset of those prefixed under /PREFIX=ALL_ENTRIES). The /STANDARD=C99 qualifier causes this option to default to /PREFIX=C99_ENTRIES. But note that, as mentioned previously, new C99 run-time library functions will not be available until OpenVMS Alpha releases after V7.3. The compiler will prefix C99 entries based on their inclusion in the standard, not on the availability of their implementations in the run-time library. So calling functions introduced in C99 that are not yet implemented in the Compaq C RTL will produce unresolved references to symbols prefixed by DECC$ when the program is linked.

- C99 changes to types of integer constants

  The C99 standard introduces the type `long long int` (both signed and unsigned) as a standard integer type whose range of values requires at least 64 bits to represent. Although Compaq C on Alpha implemented the type `long long` as an extension many releases ago, the compiler followed the C89 rules for determining the type of an integer constant. Those rules specified that an unsuffixed decimal integer with a value too large to be represented in a `signed long` would be given the type `unsigned long`. Compaq C followed this rule and gave a constant too large for `signed long` the type `unsigned long` if it would fit, and only gave it a `long long` type if the value was too large for `unsigned long`.

  In standardizing the `long long` type, C99 regularized these rules and made them extensible to longer types. In particular, unsuffixed decimal integer constants are given the smallest signed integer type that will hold the value (the minimum type is still `int`). If the value is larger than the largest value of `signed long long`, then it is given the next larger implementation-defined signed integer type (if there is one). Otherwise the behavior is undefined. Since Compaq C does not implement a signed integer type longer than `long long`, it will use the type `unsigned long long` next, with a portability warning. The only portable way to specify a decimal constant that will be given an unsigned type is to use a suffix containing "`u`" or "`U`".

  Compaq C will continue to use the C89 rules in VAXC, COMMON, and strict ANSI89 modes (including MIA), but use the new C99 rules in all other modes. The complete C99 rules are as follows:

  ```
  The  type  of  an integer constant is the first of the
  corresponding list in which its value can be represented.
  ```

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l or L	long int long long int	long int unsigned long int long long int unsigned long long int
Both u or U and l or L	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll or LL	long long int	long long int unsigned long long int
Both u or U and ll or LL	unsigned long long int	unsigned long long int

If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned.

- `#pragma names`

  This pragma offers the same kinds of control over the mapping of external identifiers into object module symbols as does the command line qualifier /NAMES, and it uses the same keywords (except that the "lowercase" keyword is not supported). But as a pragma, the controls can be applied selectively to regions of declarations. The pragma has a save/restore stack that is also managed by `#pragma environment`, and so it is well-suited to

use in header files. The effect of "#pragma environment header_defaults" is to set NAMES to "uppercase,truncated", which is the compiler default.

One important use for this feature is to make it easier to use command line option /NAMES=AS_IS. Both the C99 standard and the C++ standard require that external names be treated as case-sensitive, and 3rd party libraries and Java native methods are starting to rely on case-sensitivity (C99 requires a minimum of 31 characters significant, while C++ requires all characters significant). Therefore we expect the use of /NAMES=AS_IS to become much more widespread.

The Compaq C run-time library was implemented with all symbols duplicated and spelled both in uppercase and lowercase to allow C programs compiled with any of the /NAMES= settings to work. But traditional practice on OpenVMS combined with compiler defaults of /NAMES=UPPER has resulted in nearly all existing object libraries and shared images to contain all uppercase names (both in references and in definitions), even though C source code using these libraries typically declares the names in lowercase or mixed case. Usually, the header files to access these libraries contain macro definitions to replace lowercase names by uppercase names to allow client programs to be compiled /NAMES=AS_IS. But macro definitions are problematic because every external name has to have a macro.

The new pragma allows header files to specify just once that the external names they declare are to be uppercased in the object module, regardless of the NAMES setting used in the rest of the compilation. The NAMES setting in effect at the first declaration of an external name is the one that takes effect, thus the setting specified in a header file will not be overridden by a subsequent redeclaration in the user's program (which might specify a different NAMES setting). Note that the automatic Prologue/Epilogue header file inclusion feature described in section 1.7.4 of the User's Guide (in connection with pointer_size pragmas) can also be used to specify the NAMES setting for all headers in a given directory or text library, without having to edit each header directly.

Syntax:

```
#pragma names <stack-option>
#pragma names <case-option>[, <length-option>]
#pragma names <length-option>[, <case-option>]
```

Where <stack-option> is one of:

```
    save      - save the current names state
    restore   - restore a saved names state

<case-option> is one of:

    uppercase - uppercase external names
    as_is     - do not change case

and <length-option> is one of

    truncated - truncate at 31 characters
    shortened - shorten to 31 using CRC
```

- Change to #pragma optimize

  An important change was made to the behavior of #pragma optimize, which was introduced in V6.2 The pragma is no longer controlled by #pragma environment. It still supports the save and restore keywords, but its state is completely separate from the state managed by #pragma environment. In addition, it has a new keyword, command_line. This keyword causes the optimization settings to revert to what was in effect at the start of the compilation, as specified by the CC command line qualifiers.

- New command line qualifier /FIRST_INCLUDE

  This qualifier lists one or more header file specifications that are to be included before the first line of the main source file. Each header specification is treated as if it appeared within quotes on a #include directive before the first line of source. The headers are included in the order they are specified. This qualifier can be particularly useful to shorten CC command lines with lengthy /DEFINE and/or /MESSAGE qualifiers, by converting them to equivalent #define and #pragma message directives in a file specified by /FIRST_INCLUDE.

  Syntax:

  ```
  /FIRST_INCLUDE=(file[,...])
  /NOFIRST_INCLUDE (D)
  ```

- New "_m" suffix forces pragmas to expand macros

As specified in the Language Reference Manual, there is a fixed "grandfathered" list of #pragma directives that always undergo macro expansion in the preprocessor before being translated. No other #pragma directives normally undergo macro expansion. But since there are sometimes circumstances where macro expansion is needed on a particular instance of a pragma, a general mechanism has been added such that spelling the name of a #pragma directive with a trailing "_m" suffix will cause that directive to undergo macro expansion. An example of use follows in the next bullet item for #pragma assert non_zero. The suffix is permitted on all pragmas, including those that are already specified as undergoing macro expansion (in which case it has no effect).

- New non_zero keyword for #pragma assert

  This new keyword allows you to assert that a particular constant-expression must be non-zero at compile time, and supply a text string to be output if the assertion proves false. The text string is output with a warning message that includes the source text of the expression.

```
Syntax:
   #pragma assert non_zero(<constant-expression>)
                                              <string-literal>
```

  Note that the constant-expression is a C language constant-expression, not just a preprocessor #if expression. And while #pragma assert itself does not perform macro expansion, the alternate form #pragma assert_m can be used to cause macro expansion to take place, which is most often what is desired, as in the second example below (since "offsetof" is a macro).

```
Example:
   #pragma assert   non_zero(sizeof(a) == 12) "wrong size a"
   #pragma assert_m non_zero(offsetof(s,b)==4) "wrong offset b"
```

  If the sizeof a is not 12, and the offset of member b in the struct named by type b is not 4, the following diagnostics are output:

```
 CC-W-ASSERTFAIL, The assertion "(sizeof(a) == 12)"
    was not true. wrong size a.
 CC-W-ASSERTFAIL, The assertion "(offsetof(s,b) == 4)"
    was not true. wrong offset b.
```

- New #pragma unroll

This pragma controls the amount of loop unrolling performed on a subsequent `for` loop.

```
Syntax:

  #pragma unroll (unroll_factor)

Example:

  #pragma unroll (1)
  for (i=0; i<1000; i++) {foo(i);}
```

The `unroll` pragma directs the compiler to unroll the `for` loop that follows it by the number of times specified by the unroll_factor argument. The directive must be immediately followed by the `for` statement it is to control, otherwise a warning is issued and the pragma is ignored. The unroll_factor is an integer constant in the range from zero to 255. Using a value of zero will cause the directive to be ignored and the compiler will determine the number of times to unroll the loop in its normal way. Using a value of one will prevent the loop from being unrolled.

- New use of `static` keyword in array bounds.

  C99 permits the keyword "`static`" to be used within the outermost array bound of a formal parameter in a prototype function declaration. The effect is to assert to the compiler that at each call to the function, the corresponding actual argument will provide access to at least as many array elements as are specified in the declared array bound. Consider the following two function definitions:

  ```
  void foo(int a[1000]){ ... }
  void bar(int b[static 1000]) { ... }
  ```

  The declaration of foo is absolutely equivalent to one that declares "a" to be "int *". When compiling the body of foo, the compiler has no information about how many array elements may exist. The declaration of bar differs in that it asserts to the compiler that it may assume that at least 1000 array elements exist and may be safely accessed. The intent is to provide a hint to the optimizer about what can be safely pre-fetched.

- New type keyword `_Imaginary`

C99 reserves the keyword _Imaginary for use as a type-specifier in conjunction with an experimental/optional feature called a "pure imaginary" type, specified in informative Annex G. The overall intent of the feature is to regularize the effects of "maximal IEEE" behaviors on operations involving complex types. Annex F of C99 specifies a set of "maximal IEEE" behaviors that give optional aspects of the IEEE standard a binding in C semantics, and provides that a C implementation should predefine the macro __STDC_IEC_559__ with a value of 1 if it conforms to all of the specifications in Annex F (note that IEC 60559:1989, IEC 559:1989, IEEE 754-1985, and IEEE 854-1987 are all essentially equivalent for an implementation that uses binary radix). Compaq C does not predefine __STDC_IEC_559__, and does not implement the _Imaginary type that addresses the issues of using __STDC_IEC_559__ features on the complex data types. In Compaq C, use of the _Imaginary keyword produces a warning, which is resolved by treating it as an ordinary identifier.

5.6 Enhancements in V6.2

This release primarily contains a number of new language features from the in-process revision to the C standard, C9X (expected to be C99), and from the gcc compiler (to aid compatibility with source code from Linux systems). It also has run-time performance enhancements (including tuning for the EV6 processor and per-function optimization controls) and diagnostic message improvements, as well as bug fixes and miscellaneous improvements.

The following specific enhancements were made:

- preprocessor expressions evaluated in 64 bits (C9X)

  Arithmetic expressions evaluated within the preprocessor (i.e the expression in a #if directive) are now evaluated in the type long long (64 bits) instead of long (32 bits).

- unreachcode message no longer enabled by default

  The unreachcode message, which detects unreachable code is no longer enabled by default. If you wish the compiler to output this message, it must be enabled using either a command line qualifier or a #pragma message directive.

- diagnostic for unbalanced pragma state save/restore

  The compiler now issues a diagnostic when a #pragma stack is not empty at the end of a compilation (i.e. when the program issues a `#pragma <pragma-name> save` directive without ever issuing a corresponding `#pragma <pragma-name> restore`

- additional diagnostics to help locate unmatched braces

When a closing brace is omitted, the parser error is generally reported against a location far from the point of the actual coding error. Additional diagnostics now attempt to find which opening brace was not matched, based on heuristic observation of the indentation style used in the rest of the source code. In testing, these messages have proved quite accurate, but if the source code is very inconsistent in this style, or uses an unusual style, the messages may not be very effective. We would appreciate feedback on this feature, particularly testcases where the diagnostic misidentified which brace was unmatched.

- tuning for the EV6 processor and correction to new builtin functions.

  The V6.0 compiler's support for the EV6 processor was not well-tuned and sometimes its EV56 tuning greatly outperforms its EV6 tuning when running on an EV6 processor. This has been addressed (although some programs may still run better with EV56 tuning, this should happen much less often and with a much smaller margin of difference). Also, the builtin functions _popcnt, _poppar, _leadz, and _trailz were corrected to avoid generating new EV67 instructions on EV6 machines, and their return types were changed from unsigned __int64 to __int64. The latter change represents an ease-of-use improvement (unsigned operands can cause surprising results for arithmetic expressions) and corrects an unintended incompatibility with Tru64 UNIX.

- initial support for EV67 processor

  EV67 is now accepted as a keyword for the /arch and /opt=tune qualifiers. This enables generation of the bit-counting instructions for the _popcnt, _poppar, _leadz, and _trailz builtin functions.

- per-function optimization control: `#pragma optimize`

  `#pragma optimize` sets the optimization characteristics of function definitions that follow the directive. It allows options to control optimization that are normally set on the command line for the entire compilation to be specified in the source file individually for specific functions. Note that while the controls have effects similar to corresponding command line controls, the final generated code will be somewhat different. In particular, the optimization level controls specified by the pragma do not affect the final instruction-level optimizations and scheduling - those phases of optimization are controlled only by the command-line optimization level.

```
Syntax:

  #pragma optimize <options>

  Where <options> is one of:
      save, restore, <settings>

  and <settings> is any combination of:
      <level settings>
      <unroll settings>
      <ansi-alias settings>
      <intrinsic settings>

  <level settings> sets optimization level, of the form:
      level=n
  where n is from 0 to 5.

  <unroll settings> controls loop unrolling, of the form:
      unroll=n
  where n is a non-negative integer.

  <ansi-alias settings> controls ansi-alias assumptions:
      ansi_alias=on   or   ansi_alias=off

  <intrinsic settings> controls recognition of intrinsics:
      intrinsics=on   or   intrinsics=off

  Whitespace is optional between the setting clauses and
  before and after the "=" in each clause.  The pragma is
  not subject to macro replacement.

  An example would be:

      #pragma optimize level=5 unroll=6

  If the level=0 clause is present, it must be the only
  clause present.

  This directive must appear at file scope - outside any
  function body.
```

Semantics:

1. The save and restore options save and restore the current optimization state (level, unroll count, ansi-alias setting, and intrinsic setting). This is similar to the other environment controls (message, member_ alignment...).
2. #pragma environment save and restore operations include the optimization state.
3. #pragma environment command_line resets the optimization state to that specified on the command line.

4. If the pragma does not specify a setting for one of the optimization states, that state will remain unchanged.
5. When a function definition is encountered, it is compiled using the optimization settings current at that point in the source.
6. When a function is compiled under level=0, the compiler will not inline that function. In general, when functions are inlined, the inlined code is optimized using the optimization controls in effect at the call site instead of using the optimization controls specified for the function being inlined.
7. When the VMS command line specifies /NOOPT (or /OPTIMIZE=LEVEL=0), the #pragma has no effect (except that its arguments are still validated).

- new keywords for /accept qualifier

  New keywords for the /accept command line qualifier:

  - [no]c99_keywords

    Controls whether or not the new keywords being introduced in the C standard that are in the C89 namespace for user identifiers (inline and restrict) are accepted without double leading underscores.

  - [no]gccinline

    The gcc compiler implements an inline function qualifier for functions with external linkage that gives similar capabilites to the C9X feature described below, but the details of usage are somewhat different (basically the combination of extern and inline keywords makes an inline definition, instead of the exclusive use of the inline keyword without the extern keyword). This option controls which variation of the feature is implemented.

- extern inline functions (C9X and gcc)

  A new keyword, `inline`, has been introduced which can be used as a declaration specifier in the declaration of a function. With static functions, this has the same effect as applying `#pragma inline` to the function. When the specifier is applied to a function with external linkage, besides suggesting to the compiler that calls within that translation unit be inlined, there are additional rules that allow calls to the function also to be inlined in other translation units or else called as an external function at the compiler's discretion:

  - If the inline keyword is used on a function declaration with external linkage, then the function must also be defined in the same translation unit.

- If all of the file scope declarations of the function use the inline keyword but do not use the extern keyword, then the definition in that translation unit is called an inline definition, and no externally-callable definition is produced by that compilation unit. Otherwise, the compilation unit does produce an externally-callable definition.
- An inline definition must not contain a definition of a modifiable object with static storage duration, and it must not refer to an identifier with internal linkage. These restrictions do not apply to the externally-callable definition.
- As usual, at most one compilation unit in an entire program can supply an externally-callable definition of a given function.
- Any call to a function with external linkage may be translated as a call to an external function, regardless of the presence of the inline qualifier. It follows from this and the previous point that any function with external linkage that is called must have exactly one externally-callable definition among all the compilation units of an entire program.
- The address of an inline function with external linkage is always computed as the address of the unique externally-callable definition, never the address of an inline definition.
- A call to an inline function made through a pointer to the externally-callable definition may still be inlined, or translated as a call to an inline definition, if the compiler can determine the name of the function whose address was stored in the pointer.

- intermixed declarations and code, "for" loop declarations (C9X and gcc)

  The C++ language has always allowed these, and they are now being added to C. The rules are the same as for C++. Within a compound statement, statements and declarations may be freely interspersed. This allows declarations to be placed nearer to their point of first use without introducing additional nested compound statements.

  And in the `for` statement, the first clause may be a declaration whose scope includes the remaining clauses of the `for` header and the entire loop body. This is normally used to declare and initialize a local loop control variable, e.g.

```
for (int i=0; i<10; i++)
    printf("%d\n", i);
```

- __func__ predeclared identifier (C9X and gcc)

Anywhere within the body of a function definition, code can assume that there is visible an identifier named __func__ that is declared as a static array of char initialized with the spelling of the function's name. E.g a function defined as

```
void foo(void) {printf("%s\n", __func__);}
```

will print "foo".

- compound literals (C9X and gcc)

  A compound literal is a new form of expression that constructs the value of an object, including objects of array, struct, or union type. In C89, passing a struct value to a function typically involves declaring a named object of the type, initializing its members, and passing that object to the function. A compound literal is an unnamed object specified by syntax consisting of a parenthesized type name (i.e. the same syntax as a cast operator) followed by a brace-enclosed list of initializers. Note that the initializer list can use the recently-introduced designator syntax.

  E.g. to construct an array of 1000 ints that are all zero except for array element 5, which is to have a value of 7, you can write: (int [1000]){[5] = 7}.

  A compound literal object is an lvalue. The object it designates has static storage duration if it occurs outside of all function definitions, and otherwise has automatic storage duration associated with the nearest enclosing block.

- comment introducers optionally detected within comments, to help find unterminated comments

  The new message "nestedcomment" can be enabled to report occurrences of "/*" inside of a comment introduced by "/*". This often indicates that a terminating "*/" was omitted, although certain coding practices may also produce many occurrences that are harmless. The message is also enabled by enabling the "level4", "unused", or "questcode" groups.

- __align synonym for _align (ANSI namespace)

  The long-supported _align storage class modifier is in the namespace reserved for users' identifiers in some contexts under the C standard, and so it could not be recognized as a keyword in strict ANSI mode (/standard=ansi89). An alternate spelling with two leading underscores (putting it in the namespace reserved to the C implementation) is now recognized in all modes so that the feature can be used when compiling in strict ANSI mode.

- __typeof__ unary operator (gcc)

The gcc compiler provides an operator named "__typeof__" that can be used much like the standard C operator "sizeof", except that instead of producing a value (the size of an expression or type) it produces a type (the type of the expression or type that is its single operand). It can be convenient to use in macros for generating declarations or casts that use the same type as some expression or typename supplied as an argument to the macro.

- pointer arithmetic on void and function pointers (gcc)

  Pointer arithmetic on void* pointers or pointers to functions formerly produced hard errors. The gcc compiler allows this, treating both as if they were char* pointers. This behavior has been adopted, with an appropriate warning instead of an error.

- string initializers optionally enclosed in parentheses (gcc)

  In standard C, a string literal ordinarily has type char *. A special case is made when a string literal is the initializer for an array of char, in which case it is essentially treated as an array object that provides the values (and, in the case of an incomplete array, the size) of the array of char being initialized.

  Also in standard C, a string literal enclosed in parentheses is not itself a string literal, and so this special case would not apply - instead the parenthesized literal would be treated as a single pointer of type char *, which is not a valid initializer for an object of type array of char.

  The gcc compiler allows a parenthesized string literal to initialize a char array, and the construct commonly appears in Linux source code. So that behavior has been adopted, with an appropriate warning.

- difference of addresses in same object are constants (gcc)

  If the & operator is used to obtain the addresses of two lvalues within the same object, and the lvalues are specified with integral constant expressions, then the result depends only on the layout of the object, and for practical purposes can be computed at compile time much like the integral constant expressions that are required to be produced by sizeof and offsetof.

  E.g. for any array "a" of int, the value of &a[4] - &a[3] must be sizeof(int), and sizeof(int) is an integral constant expression. But the C standard's specification of expressions that must be treated by the compiler as integral constant expressions does not include use of the & operator, and so &a[4] - &a[3] is not required to be treated as such, and Compaq C has not previously done so. But the C standard also explicitly recognizes that implementations may treat additional forms of expressions as constant

expressions, and gcc and other compilers do treat these cases as integral constant expressions. Now Compaq C does as well.

- allow #pragma use_linkage to take typedef names as well as functions

  #pragma use_linkage directive has been extended so that it can take either a typedef name or a function name. If the typedef name is a function type, then functions or pointers to functions declared with that type will have the specified linkage. This allows programmers to invoke functions that have a special linkage using a pointer to a function.

5.7 Enhancements in V6.0

- New command line qualifier /[NO]PROTOTYPES:

```
    /[NO]PROTOTYPES=(FILE=<filename>,
                    [NO]IDENTIFIERS,
                    [NO]STATIC_FUNCTIONS)

    default is /[NO]PROTOTYPES

Syntax Description:

    /[NO]PROTOTYPES
        Creates an output file containing function prototypes
        for all global functions defined in the module which
        is being compiled.

    keywords:

      IDENTIFIERS
            negatable optional parameter, which indicates that
        identifiifer names are to be included in the prototype
        declarations that appear in the output file.  The
        default is NOIDENTIFIERS

      STATIC_FUNCTIONS
        negatable optional parameter, which indicates that
        prototypes for static function definitions are to be
        included in the output file.  The default is
        NOSTATIC_FUNCTIONS

      FILE=<filename>
        an optional parameter specifying the output file name.
        When not specified the output file name will be
        have the same defaults as listing file, except that
        the .CH file extension is used instead of the .LIS
        extension.
```

- New command line qualifier /ASSUME=[NO]WEAK_VOLATILE. Specifying /ASSUME=WEAK_VOLATILE tells the compiler to generate code for assignments to objects that are specified as volatile and smaller than 32 bits without the load-locked/store-conditional sequences that in general are required to assure volatile data integrity. This option is intended for use in special hardware access situations, and should not generally be used.
- New command line qualifier, `/SHOW=MESSAGES`. This qualifier adds a new section to the listing file showing all of the compiler's diagnostic messages that are enabled at the start of the compilation, after the command line has been processed. The listing shows the message identifer, the severity, and the parameterized text of each enabled message, reflecting the effects of the /standard and /warnings command line qualifiers (except that severity-based suppression, /warnings=noinformationals or /nowarnings, is not reflected). The /warnings=verbose qualifier causes this listing to be expanded with the "Description" and "User Action" text following the text for each enabled message.
- New command line qualifier, `/CHECK=BOUNDS`. This qualifier causes the compiler to generate code to check the bounds of array-indexing expressions at runtime, and raise an exception (%SYSTEM-F-SUBRNG, arithmetic trap, subscript out of range) if the index is out of bounds. Note that the C language defines the subscript expression a[i] to be equivalent to *(a+i), relying on the implicit conversion of an array name to a pointer to the first element of an array, and on the fact that adding an integer to a pointer involves "scaling" the index by the size of the pointed-to object. So array syntax can be used either with pointers or with the names of arrays. Array bounds are only checked when an element is accessed using a declared array name (using either array notation or pointer +/- integer notation). In particular, the check is made at the point that the compiler processes an add or subtract of an array name and an integer - the result of that operation is a pointer, and so subsequent operations are not included in the checking code. Also note that the C language considers computation of the address one past the end of an array to be fully portable. Therefore, expressions that appear to compute an address allow an extra element at the end. It is only when an array name is used directly with array subscript notation that the exact upper bound is checked. E.g.

```
{
    int a[5]; // elements are 0..4, but you
              // can take the address of a[5]
    int *pa, i=6, j=-6;
    pa = a + i;          // trap, &a[6]
    pa = a + i + j;      // trap, &a[6] - 6
    pa = a + (i + j);    // no trap, &a[0]
    pa = a + (i - 1);    // no trap, &a[5]
     j = a[i - 1];       // trap, a[5]
     j = *(a + (i - 1)); // no trap, looks like &a[5]
}
```

- New informationals to report apparently unnecessary #include files and CDD records. The most useful of these, UNUSEDTOP, reports only headers explicitly #included at the top level in the compilation unit that did not provide anything used by the rest of the compilation. This message is enabled at level4. Other diagnostics to report on the effects of #includes nested within other headers, and on CDD records, are enabled at level5. All of these messages can be enabled by the message group UNUSED. Unlike any other messages, these messages must be enabled on the command line in order to be effective. The processing that analyzes the dependencies on included files is signficant, and it must be started before processing of the input files begins. Any #pragma message directives within the source have no effect on these messages, their state is determined only by processing the command line.
- Variable length arrays from the C9X review draft have been implemented. This feature permits array objects with auto storage class, and array typedefs declared at block scope, to have bounds that are runtime-computed expressions. It also permits the declaration and definition of functions whose parameters are arrays dimensioned by other parameters (similar to Fortran assumed-shape arrays). The following example illustrates both uses. Note that the definition of function sub( ) uses prototype syntax and that the dimension parameters precede the array parameter that uses them. In order to define a function with the dimension parameters following the array parameter that uses them, it is necessary to write the function definition using K&R syntax (since that syntax allows the declarations of the types of the parameters to be written in a different order from the parameters themselves). K&R function definitions should generally be avoided.

```
#include <stdio.h>
#include <stdlib.h>

void sub(int, int, int[*][*]);

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("Specify two array bound arguments.\n");
        exit(EXIT_FAILURE);
    }
    {
        int dim1 = atoi(argv[1]);
        int dim2 = atoi(argv[2]);
        int a[dim1][dim2];
        int i, j, k = 0;
        for (i = 0; i < dim1; i++) {
            for (j = 0; j < dim2; j++) {
                a[i][j] = k++;
            }
        }
        printf("dim1 = %d, dim2 = %d.",
               sizeof(a)/sizeof(a[0]),
               sizeof(a[0])/sizeof(int));

        sub(dim1, dim2, a);
        sub(dim2, dim1, a);
    }
    exit(EXIT_SUCCESS);
}

void sub(int sub1, int sub2, int suba[sub1][sub2])
{
    int i, j, k = 0;
    printf("\nIn sub, sub1 = %d, sub2 = %d.",
                      sub1,      sub2);
    for (i = 0; i < sub1; i++) {
        printf("\n");
        for (j = 0; j < sub2; j++) {
            printf("%4d", suba[i][j]);
        }
    }
}
```

Finally, note that variable length arrays can often be used in place of the non-standard alloca( ) [__ALLOCA( )] intrinsic, an important difference being that the storage allocated by __ALLOCA is not freed until return from the function, while the storage allocated for a variable length array is freed on exit from the block in which it is allocated. If __ALLOCA is called within the scope of a variable length array declaration (including within a block nested within the block containing a variable length array declaration), then the storage allocated by that call to __ALLOCA is freed

at the same time that the storage for the variable length array is freed (i.e. at block exit rather than function return). The compiler issues a warning in such cases.

5.8 Enhancements in V5.7

This release contains a number of new features aimed primarily at ease of use and programmer productivity, as well as performance and bug fix improvements.

- New command line qualifier, /OPTIMIZE=INTRINSICS. In V5.6, the special treatment of various standard runtime library functions was accomplished solely by explicit source code in the standard header files. E.g. in <string.h> there is: #define memcpy(__x, __y, __z) __MEMCPY(__x, __y, __z). As a result, the compiler would not give special treatment to calls to memcpy unless the compilation #included the standard header. Similarly, the compiler did not do compile-time analysis of printf format strings unless <stdio.h> was #included and the macro _INTRINSICS was defined. Under this new option, extern functions whose names and call signatures match those of standard library functions can be recognized automatically as special (intrinsic) even if the corresponding header file is not included. In addition, many more functions can be handled as intrinsics than were previously available through the header files. The online help for this qualifier contains the list of functions currently recognized. The default is /OPTIMIZE=INTRINSICS.
- New command line qualifier /ASSUME=NOMATH_ERRNO. The C standard requires that calls to certain library functions in <math.h> communicate error information using errno. Since this behavior effectively means that calls to such functions must be considered to have side-effects, the calls cannot be optimized. This option asserts to the compiler that the program does not depend on the setting of errno by those functions, allowing the compiler to optimize calls to them. In practice, most production-quality C programs that perform substantial floating point computations do not rely on the setting of errno by math library functions. But because the setting of errno by math functions is a traditional behavior that is also required by the standard, and the compiler cannot reliably determine whether or not the program actually depends on it, the compiler must refrain from optimizing these math function calls unless the user explicitly allows it by specifying this option. The online help for /OPTIMIZE=INTRINSICS lists the functions affected.

- New command line qualifier /ASSUME=WHOLE_PROGRAM. This qualifier asserts to the compiler that it is compiling the entire program at one time, except for "well-behaved" library routines. This is normally useful only in conjunction with /PLUS_LIST_OPTIMIZE. A well-behaved library routine is one that does not use external linkage to read or write any global variables that are visible to the compilation, and does not use external linkage to access any function defined in the compilation. It is a slightly stronger assertion than /ASSUME=NOPOINTERS_TO_GLOBALS.
- New command line qualifier EXACT_CDD_OFFSETS. This qualifier tells the compiler to use the exact offsets as specified by #pragma dictionary records, regardless of the current state of alignment controls. By default, the offsets specified by CDD may be rounded up, subject to the alignment controls in effect at the point of the #pragma.
- New command line qualifier /CHECK=POINTER_SIZE=INTEGER_CAST. Generates a runtime check to verify that no bits are lost whenever a 64-bit pointer is cast to a 32-bit integer.
- New command line qualifier /ASSUME=NOCLEAN_PARAMETERS. This qualifier tells the compiler that functions defined in this compilation may be called from a separate compilation and passed improperly-prepared arguments. In particular, functions defined to take 32-bit integer or pointer arguments will have additional code generated to sign-extend the upper 32 bits of each such argument. Note that the Calling Standard for Alpha requires that 32-bit values must be passed in 64-bit sign-extended form, even when the argument is of an unsigned type. But type mismatches across separate compilation units can cause this requirement to be violated, and use of this option can help detect/correct the problem in order to aid the developer. Ordinarily, this option should not be necessary and is not recommended for production code.
- New command line qualifier /OPTIMIZE=PIPELINE. At optimization levels of 2 or higher, this qualifier enables an optimization called "software pipelining", in which certain loops may be reordered to start some of the work of one iteration in an earlier iteration, and in some cases perform data prefetching to reduce the impact of cache misses. This optimization is included by default at level 5.
- Enhanced diagnostic message controls. The /WARNINGS command line qualifier and its matching #pragma message have had a number of new features added in an upwardly-compatible way. Refer to the online help for /WARNINGS for specific usage information. Features include:
    - Specify whether a message is issued only once per compilation, or at each occurrence.

- Reduce the severity of any message that has a default severity of informational or warning, or increase the severity of any message. Reducing a warning to an informational can allow generation of a "warning-free object module", without suppressing the diagnostic altogether. Increasing the severity of an informational or warning to an error can help enforce programming practices by causing specific diagnostics to "break" a build.
- Control optional messages using a single numeric "importance level". The "check" group of messages basically allowed enabling a large number of additional messages, some useful, some not very useful in many cases. Messages have now been grouped into 5 importance levels, named level1-level6. The default is level3. The "check" group is now treated as a synonym for level5. The "all" group is treated as a synonym for level6. The level1 and level2 groups correspond to "quiet" and slightly more "noisy" versions of Digital UNIX compilers, respectively. Enabling a level enables optional messages at that level and all lower levels. Disabling a level disables optional messages at that level and all higher levels.
- Control optional messages using functional groups. The previous functional groups (c_to_cxx, check, portable, all) have been retained, and a number of new groups have been added. Many of the names for the new functional groups correspond to groups recognized by the "lint" utility on DIGITAL UNIX: ALIGNMENT, DEFUNCT, NOANSI, OBSOLESCENT, OVERFLOW, PERFORMANCE, PREPROCESSOR, QUESTCODE, RETURNCHECKS, UNINIT, UNUSED, CDD.
- /WARNINGS=VERBOSE adds explanatory help text following each diagnostic message output.

Besides the new features, the entire set of compiler messages was reviewed and updated. As a result, the exact set of messages reported by a default compilation is somehwat different. Overall, the default level3 setting is slightly quieter, particularly because the default mode of relaxed_ansi does not report uses of language extensions. Also, the severity of many warning messages has been reduced to informational. Finally, the Messages subtopic for CC now contains useful additional information about each message.

- New command line qualifier `/SHOW=SYMBOLS`. This will add a symbol table map to the listing (if a listing is requested). This is similar, but not identical, to the output from the VAX C compiler.

- New command line qualifier /SHOW=BRIEF. This qualifier is similar to the new qualifier /SHOW=SYMBOLS, except that unreferenced symbols declared in header files are omitted.
- New command line qualifier /CROSS_REFERENCE, or equivalently /SHOW=CROSS_REFERENCE. This qualifier adds a list of line numbers at which each listed symbol is referenced (if a listing is requested). If the /SHOW qualifier is omitted, this qualifier causes the /SHOW=BRIEF symbols to be listed. When appropriate, the line number designating a reference to a symbol is annotated with a suffix indicating the way in which the symbol was used on that line, as follows:
  - = Assigned or initialized.
  - & Address taken.
  - () Function called.
  - * Simple dereference.
  - -> Member dereference.
  - . Member selection (no indirection).
  - [] Subscripted (i.e. using [] syntax).
  - b Invoked as a builtin function.
- New command line qualifier /ACCEPT=[NO]feature. This qualifier tells the compiler to accept (or reject) particular language features, regardless of the setting of the /STANDARD qualifier. There are two features that can be controlled in this way:
  - VAXC_KEYWORDS. Specifying this feature tells the compiler to recognize and process the following identifiers as keywords: _align, globaldef, globalref, globalvalue, noshare, readonly, variant_struct, variant_union. Specifying NOVAXC_KEYWORDS tells the compiler to treat these as ordinary identifiers. The default is to recognize these as keywords in all language modes other than strict ANSI and common modes.
  - RESTRICT_KEYWORD. Specifying this feature tells the compiler to recognize and process the C9X keyword restrict as a type qualifier keyword. By default, in current language modes only the reserved-namespace spelling __restrict is treated as a keyword.
- New command line qualifier /NAMES=SHORTENED. External symbol names longer than 31 characters are, by default, truncated to 31 characters by the compiler in order to conform to the linker limit, as they always have been. This new option instructs the compiler to shorten the name without losing

all information about the characters that were removed. The shortened name contains a CRC encoding of the characters removed, similar to way that the C++ compiler treats its mangled names that very often exceeed 31 characters. This allows programs containing long external names that are not unique within the first 31 characters to be linked successfully. Naturally, if a program contains external names longer than 31 characters, all of its modules must be compiled with the same setting of this qualifier in order to link successfully. The default is `/NAMES=TRUNCATED`.

- New command line qualifier `/REPOSITORY=dirspec`. This qualifier is only useful in conjunction with `/NAMES=SHORTENED`, and when the default directory specification of `[.CXX_REPOSITORY]` is not acceptable. When the compiler shortens a name under the `/NAMES=SHORTENED` option, it also writes a mapping from the shortened name to the original full-length name in the repository. The `CXXDEMANGLE` utility, which now also ships with the C compiler, can be used to find the original name corresponding to a shortened name. That utility also assumes that the shortened name repository is located in `[.CXX_REPOSITORY]` unless a different directory is explicitly specified. See the help for `CXXDEMANGLE`. An option to perform compatible shortening on long names with extern "C" linkage is planned for a future release of C++. Note that a shortened C name is formed using a convention that will never match a C++ "mangled" name, so a single repository can be used by all C and C++ compilations.
-

5.9 Enhancements in V5.6

This is primarily a maintenance release focused on bug fixes, performance, usability and message improvements, and providing V7.1 runtime library features on prior versions of VMS.

- Optimizer for Alpha now exploits the __restrict qualifier in limited ways. Future releases will expand this kind of optimization.

- Optional compile-time diagnostics and optimizations for certain kinds of format strings passed to the printf family of library functions.

  If a preprocessor macro named "_INTRINSICS" is defined prior to inclusion of the V5.6 header file stdio.h, the compiler will perform compile-time analysis of format strings and arguments passed to printf, fprintf, and sprintf when possible. When the format string passed to one of these functions is an explicit compile-time-known string, this feature permits the compiler to diagnose mismatches in number and type between the %-specifiers in the format string and the arguments to be formatted. Most

such format strings will be converted to a more efficient run-time encoding handled by new library routines. In addition, a number of special cases are recognized which will cause the compiler either to generate calls directly to lower-level library routines instead of printf, or to generate inline code, avoiding the need to do any format decoding at runtime. For example, a format such as "%s" passed to fprintf( ) can be converted to a call to fputs( ). When passed to sprintf, it might be converted either to a call to strcpy or to inline code to copy characters into the buffer. In versions of OpenVMS through V7.1, the runtime support for this feature is provided only through object modules placed in SYS$LIBRARY:STARLET.OLB by this kit.

- Command line qualifiers /ARCHITECTURE= and /OPTIMIZE=TUNE= documented in online help, allow the compiler to exploit more fully features of newer Alpha chips including byte/word memory access instructions.

- Message group C_TO_CXX. This message group contains an optional set of diagnostics that report the use of a number of C language constructs that are not compatible with, or have a slightly different meaning in, the C++ language. This group may be enabled explicitly either on the command line (/WARN=ENABLE=C_TO_CXX) or by #pragma message enable (c_to_cxx).

- New runtime check, /CHECK=POINTER_SIZE=INTEGER_CAST. This causes the compiler to generate code to check at runtime that casts from 64-bit pointer to 32-bit integer do not overflow. The expected behavior of casts to integer types is to truncate the value silently. But in porting 32-bit code to exploit 64-bit pointers, such casts can occur unintentionally and produce runtime failures that are otherwise very difficult to analyze.

- New diagnostics to detect simple expressions with side effects that are undefined in ANSI C. The C standard formalized defacto rules about side effects in terms of sequence points. An expression that modifies the same object more than once, or that modifies an object and fetches its value for a purpose other than computing the modified value, has undefined behavior unless there is an intervening sequence point. The compiler now warns about such expressions (only for objects that are simple declared variables).

- Source listings now include statement level nesting. The annotation at the left margin of the source listing now includes the statement nesting level in effect at the end of that source line. The statement nesting level appears as a simple integer before the listing line number. The block of a function definition is level 1. Outside of function definitions, this field is blank.

5.10 Enhancements in V5.5

This is primarily a maintenance release focused on bug fixes, with very limited new functionality.

- New command line qualifier `/ASSUME=NOPOINTERS_TO_GLOBALS`.

  This qualifier tells the compiler it is safe to assume that global variables have not had their addresses taken in a separate compilation. By default, the compiler assumes that global variables may have had their addresses taken in separately compiled modules, and that in general any pointer dereference may be accessing the same memory as any global variable. This is often a significant barrier to optimization. The /ANSI_ALIAS command line qualifier allows some resolution based on data type, but this new qualifier provides significant additional resolution and improved optimization in many cases. Note that this qualifier does not tell the compiler that the compilation never uses pointers to access global variables (which is seldom true of real C programs). Instead, it tells the compiler that any global variable accessed through a pointer in the compilation must have had its address taken within that compilation. In combination with /plus_list_optimize, several source modules can be treated as a single compilation for the purpose of this analysis. Since runtime libraries such as the CRTL do not take the addresses of global variables defined in user programs, it is often possible to combine source modules into a single compilation that allows this qualifier to be used effectively.

- New type qualifier "`__restrict`"

  The ongoing work to revise the ANSI C language standard will likely incorporate a new type qualifier keyword "`restrict`" (the existing ANSI type qualifiers are "`const`" and "`volatile`"). This feature has been present in the Cray C compiler for some time and is also being adopted by other vendors. The type qualifier applies only to pointer types, and its basic purpose is to assert to the compiler that memory accesses made through a pointer declared with this type do not overlap with other memory accesses within the scope of that pointer, permitting additional optimization. In this release, the qualifier (with double leading underscores to avoid violating the ANSI89 namespace) is recognized and its correct compile-time usage is verified, but it does not yet trigger additional optimizations.

- Initial macros shown in listing file

At the end of the listing file there is now a section containing a list of all macros in effect at the start of the compilation, along with their values. This includes both those predefined by the compiler (except for ANSI-mandated macros that cannot be undefined or redfined) and the result of applying all /DEFINE and /UNDEFINE qualifiers.

5.11 Enhancements in V5.3

- New qualifier keyword value /STANDARD=MS.

  This qualifier enables language compatibility features to accept some of the language extensions present in 32-bit Microsoft C compilers (such as the C compiler packaged with Visual C++) and causes the predefined macros "__MS" and "__DECC_MODE_MS" to become defined with a value of 1. It does not provide complete compatibility with a particular version of Microsoft's compiler, but a limited selection of relatively minor extensions that can ease porting of C code developed under Microsoft C. Examples include unnamed struct and unions (same syntax as unnamed unions in C++, similar function to variant struct and union in VAX C), and relaxation of pointer and integer comparisons. It does not include such major extensions as structured exception handling or thread local storage.

- New qualifier keyword value /ASSUME=[NO]HEADER_TYPE_DEFAULT

  The negated form of this value disables the compiler's supplying of a default file type extension of ".H" for source files included by the #include preprocessing directive. This feature is primarily for compatibility with the C++ compiler, where the rules for ANSI C++ header file specifications conflict with the notion of having the compiler supply a file type. The default is /ASSUME=HEADER_TYPE_DEFAULT, which enables the compiler to supply the file type ".H" for included files as it always has. Also see Section 5.11.1 for more information on changes to #include processing.

- Attribute controls for psects using #pragma extern_model

  The extern_model pragma has been enhanced to allow explicit control over most psect attributes, not just shr/noshr. The syntax is:

```
#pragma extern_model model_spec [attr[,attr]...]

where model_spec is one of:
     common_block
     relaxed_refdef
     strict_refdef "name"
     strict_refdef /* No attr specifications allowed. */
     globalvalue   /* No attr specifications allowed. */

attr is chosen from (at most one from each line):
     gbl lcl       /* Not allowed with relaxed_refdef */
     shr noshr
     wrt nowrt
     pic nopic     /* Not meaningful for Alpha */
     ovr con
     rel abs
     exe noexe
     vec novec
     0 byte 1 word 2 long 3 quad 4 octa 16 page
```

A description of these attributes may be found in table 4-4 of the DEC C User's Guide for OpenVMS Systems, and more complete information on each may be found in the OpenVMS Linker Utility Manual. The default attributes are: noshr, rel, noexe, novec, nopic. For strict_refdef the default is con, and for common_block and relaxed_refdef the default is ovr. The default for wrt/nowrt is determined by the first variable placed in the psect. If the variable has the const type qualifier (or the readonly modifier) the psect will be set to nowrt, otherwise it is set to wrt.

Restrictions on setting attributes:

`#pragma extern_model` will not set psect attributes for variables declared as tenative definitions in the relaxed_refdef model. A tentative definition is one which does not contain an initializer. For example, consider the code:

```
#pragma extern_model relaxed_refdef long
int a;
int b = 6;
#pragma extern_model common_block long
int c;
```

Psect A will be given octaword alignment (the default) because a is a tentative definition. Psect B will correctly be given longword alignment because it is initialized and therefore not a tentative definition. Psect C will also have longword alignment because it is declared in an extern_ model other than relaxed_refdef.

Also, alignment of word and byte may not currently be specified for any extern model.

NOTE: These attributes are normally used by system programmers who need to perform declarations normally done in macro. Most of these attributes are not needed in normal C programs. Also note that the setting of attributes is supported only through the `#pragma` mechanism, and not through the `/EXTERN_MODEL` command line qualifier.

- New compiler-generated psect $READONLY_ADDR$

  By default, previous versions of the compiler have placed all static-extent const data in the psect named $READONLY$. When the const data involved link-time addresses, this caused the entire $READONLY$ to become non-shared. In V5.3, static-extent const data initialized with link-time addresses is placed in a new psect named $READONLY_ADDR$, leaving $READONLY$ sharable. For example, given the declarations:

```
static const int a = 5;
static const int * const b = &a;
```

  variable a will be placed in the $READONLY$ psect because it is initialized to a true compile-time constant value, while variable b will be placed in $READONLY_ADDR$ because it is initialized to the address of a, which may differ among different image activations.

5.11.1 Changes to #include processing in V5.3

To support processing of “prologue” and “epilogue” file inclusion, the V5.2 compiler introduced substantial changes to the processing of the `#include` directive that allowed for increased code commonality between the OpenVMS Alpha and VAX versions of the compiler. In V5.3, further changes have been made to make the actual `#include` searching behavior identical for the OpenVMS VAX and Alpha compilers, and to support new ANSI C++ requirements on header file naming conventions. The following are some of the highlights of these modifications. For a complete description of #include processing, see the discussion of file inclusion in the `cc` online help for the `/include_directory` qualifier (`$ help cc/include`).

- New qualifer option, `/assume=[no]header_type_default`

  This option can disable the default file type mechanism for header files. Following VAX C, the DEC C compiler has traditionally supplied a default file type of ".H" for filenames specified without any file type extension in a `#include` directive using ANSI C syntax. Similarly, the DEC C++ compiler has supplied a default file type of ".HXX". However, the emerging ANSI standard for C++ now requires that, for example, `#include <iostream>` refer to a different file than `#include <iostream.hxx>`. V5.3 of both DEC C and DEC C++ support this capability through the /assume=noheader_ type_default qualifier option. Under this option, both DEC C and DEC

C++ supply a default file type of just "." for files named in standard-syntax #include directives. Thus, for example, if a header file directory contains two files named "STDIO." and "STDIO.H", the directive `#include <stdio>` will cause DEC C to select "STDIO.H" by default. But under /assume=noheader_type_default, "STDIO." will be selected. Besides matching the ANSI C++ requirement, this behavior is also more compatible with most other C compilers including UNIX and Windows/NT.

- Meaning of empty string in /INCLUDE_DIRECTORY

  The UNIX convention of using -I without a pathname to disable searching of the "standard places" is now fully supported by the /INCLUDE_DIRECTORY qualifier. If an empty string occurs as any element in the list of specifications supplied by this qualifier, the compiler does not search any of its default directories, logical names, or text libraries and uses only the specifications from the command line to find include files.

- ALPHA$LIBRARY system logical

  DEC C now accesses default header files and libraries using the system logical `SYS$LIBRARY` rather than `ALPHA$LIBRARY`. On an OpenVMS Alpha system, the logical name ALPHA$LIBRARY is defined identically to SYS$LIBRARY. The use of ALPHA$LIBRARY originated with internal cross-development environments that are no longer supported. If you have deliberately supplied a different definition of `ALPHA$LIBRARY` for any reason, you will need to redefine `SYS$LIBRARY` instead in order to produce the same effect on the compiler.

5.12 Enhancements in V5.2

- The command line qualifier /STANDARD=ISOC94 has been added.

  This qualifier enables digraph processing and pre-defines the macro __STDC_VERSION__ to the value 199409L. The macro __STDC_VERSION__ is used in certain header files to enable the declaration of additional wide character functions as specified in Amendment 1 to ISO/ANSI C. (Amendment 1, which specifies digraphs, alternate spellings for certain operators, and a number of new wide character functions, was officially adopted by ISO in November of 1994.)

  The meaning of /STANDARD with no keywords has been changed from /STANDARD=ANSI89 to /STANDARD=(ANSI89, ISOC94), i.e. strict conformance to the full new standard. The default mode of the compiler (/NOSTANDARD) is still RELAXED_ANSI89. The meaning of /STANDARD=ISOC94 is /STANDARD=(RELAXED_ANSI, ISOC94), i.e. requesting just the additional features from the amendment builds on the

default mode. The ISOC94 keyword may be used with any of the other keywords to the /STANDARD qualifier except VAXC.

- DEC C V5.2 has 64-bit addressing support. This support is controlled by both the new command line qualifier /POINTER_SIZE, and by the new pragmas pointer_size and required_pointer_size. In addition, the DEC C RTL interfaces in the header files are conditionalized to provide 64-bit support only when the /POINTER_SIZE qualifier is used on the command line and when the the __VMS_VER pre-defined macro is greater than or equal to 70000000 (i.e. VMS 7.0).

```
#pragma pointer_size ( operand )

where the operand can be:
     32 (or short)   32-bit pointers
     64 (or long)    64-bit pointers
     system_default  32-bit pointers (64 on Digital UNIX)
     save            remember the current pointer size
     restore         restore size to last remembered value
```

```
#pragma required_pointer_size  ( operand )

where the operand is the same as for pointer_size
```

#pragma required_pointer_size and #pragma pointer_size have the same effect except that #pragma pointer_size is enabled ONLY when the command line switch /POINTER_SIZE is used. When /POINTER_ SIZE does not appear on the command line, the pragma is ignored. #pragma required_pointer_size always takes effect regardless off whether /POINTER_SIZE has been specified. Please note that:

1. the pragmas only affect the meaning of the pointer-declarator ("*") in declarations, casts and the sizeof operator.
2. The size of a pointer is the property of the type, and so it is bound in a typedef declaration, but not in a preprocessor macro definition.
3. The size of the pointer produced by the "&" operator, or by an array name or function name in a context where it is converted to an explicit pointer, is 32 bits unless the "&" operator is applied to an object designated by a deref of a pointer whose type is a 64-bit pointer type.

```
/[NO]POINTER_SIZE = {32 | 64 | long | short}
```

This command line qualifier has the following effects:

1. Enable processing of #pragma pointer_size
2. Sets the initial default pointer size before the first pragma is seen
3. Predefines preprocessor macro __INITIAL_POINTER_SIZE to 32 or 64. In the absence of the /pointer_size qualifier, __INITIAL_POINTER_ SIZE is 0, allowing "#ifdef __INITIAL_POINTER_SIZE" to be used as a macro to test whether of not the compiler supports 64-bit pointers.
4. For /pointer_size=64 (or long), the RTL name mapping table is changed to select the 64-bit versions of malloc, calloc, and realloc (and other routines sensitive to pointer size) by default. A given call may explicitly name the 32-bit version or 64-bit version of these routines, but calls that use the normal standard names will default to calling the 64-bit version when the command line selects 64-bit pointers initially.

- /[NO]CHECK qualifier

  A new qualifier to control generation of code to perform runtime checking to help find for uninitialized variables and mixed pointer-size porblems.

```
/[NO]CHECK=([NO]UNINITIALIZED_VARIABLES,
            [NO]POINTER_SIZE=(option,...))

Valid values of option are:  [NO]ASSIGNMENT
                             [NO]CAST
                             [NO]PARAMETER
                             ALL
                             NONE
```

  This qualifier causes the compiler to emit code which will assist in identifying potentially incorrect code. This is a debugging aid.

  When /CHECK=UNINITIALIZED_VARIABLES is specified the compiler will emit code which initializes all automatic variables the value 0xfffa5a5afffa5a5a. This value is a floating NaN so it will cause a floating point trap if used in IEEE mode. If used as a pointer, it will likely causes an ACCVIO.

When /CHECK=POINTER_SIZE=(option,...) is specified the compiler will emit code to check 32-bit pointer values to ensure that the values will fit in a 32-bit pointer. If the value can not be represented by a 32-bit pointer, the runtime code will signal SS$_RANGEERR (range error). The option keywords which follow /POINTER_SIZE determine exactly which checks should be generated:

```
[NO]ASSIGNMENT - Emit a check whenever a 64-bit pointer
        is assigned to a 32-bit pointer (including use as
        an actual argument).
[NO]CAST - Emit a check whenever a 64-bit pointer is cast
        to a 32-bit pointer.
[NO]PARAMETER - Emit code that checks all formal parameters
        at function start-up to make sure that all formal
        parameters which are declared to be 32-bit pointers
        have 32-bit values.
ALL - Emit all checks.
NONE - Emit no checks.

If /CHECK=POINTER_SIZE is specified,
    the defaults are ASSIGNMENT,PARAMETER.

If /CHECK is specified, it will be the same as
    /CHECK=(UNINITIALIZED_VARIABLES,POINTER_SIZE)
which is the same as
    /CHECK=(UNINITIALIZED_VARIABLES,
            POINTER_SIZE=(ASSIGNMENT,PARAMETER))

IF /CHECK=pointer_size=ALL is specified, it's the same as
    /CHECK=(POINTER_SIZE=(ASSIGNMENT,PARAMETER,CAST))
```

The following contrived program contains a number of pointer assignments. The comment field after each line describes how to enable checking for each assignment.

```
#pragma required_pointer_size long
int *a;
char *b;
typedef char * l_char_ptr;

#pragma required_pointer_size short
char *c;
int *d;
```

```
foo(int * e)            // Check e if PARAMETER
{
    d = a;              // Check a if ASSIGNMENT
    c = (char *) a;     // Check a if CAST
    c = (char *) d;     // No checking ever
    foo( a );           // Check a if ASSIGNMENT
    bar( a );           // No checking ever - no prototype
    b = (l_char_ptr) a; // No checking ever
    c = (l_char_ptr) a; // Check a if ASSIGNMENT
    b = (char *) a;     // Check if CAST
}
```

- Implicit processing of prologue/epilogue files before and after each #include.

  When the compiler encounters a #include preprocessing directive, it first determines the location of the file or text library module to be included. It then checks to see if one or both of the two following specially named files or modules exist in the same location: __DECC_INCLUDE_PROLOGUE.H, __DECC_INCLUDE_EPILOGUE.H (in the case of a text library, the .H is stripped off). If they do, then the content of each is read into memory. The text of the prologue file (if it exists) is processed JUST BEFORE the text of the file specified by the #include, and the text of the epilogue file (if it exists) is processed JUST AFTER the text of the file specified by the #include. Subsequent #includes that refer to files from the same location use the saved text from any prologue/epilogue file found there. The prologue/epilogue files are otherwise treated as if they had been #included explicitly, and #line directives are generated for them if /prepoc output is produced, and they appear as dependencies if /mms_dependency output is produced.

  The "location" is the VMS directory containing the #included file or the text library file containing the #included module. For this purpose, "directory" means the result of using the $PARSE/$SEARCH system services with concealed device name logicals translated. So if a #included file is found through a concealed device logical that hides a search list, the check for prologue/epilogue files is still specific to the individual directories making up the search list.

  The intended purpose of this feature is largely to aid in using header files that are 64-bit "unaware" within an application being built to exploit 64-bit addressing. Current header files typically contain a section at the top that uses pragmas to save the current state of things like member_alignment and extern_model and then set them to the default values for the system, and then at the end they restore the pragmas to their previously-saved state. Mixed pointer sizes introduce another kind of state that typically needs to be saved/set/restored in headers that define fixed 32-bit interfaces to libraries and data structures. DEC C V5.0 introduced

#pragma environment to allow headers to control all of the compiler's state (message suppression, extern_model, and member_alignment) with one directive. The environment pragma now includes pointer_size as part of the state it manipulates.

But for header files that have not yet been upgraded to use #pragma environment, the /pointer_size=64 qualifier can be very difficult to use effectively. The automatic #include prologue/epilogue mechanism allows users to protect all of the header files within a single directory (or modules within a single text library) just by copying two short files into each directory or library that needs it, without having to edit each header file or library module separately. Over time, headers should be modified either to exploit 64-bit addressing (like the DEC C RTL), or to protect themselves with #pragma environment. But to ease the transition (header files come from many different places), this mechanism can be very effective. A suggested content for those files is as follows:

```
__DECC_INCLUDE_PROLOGUE.H:

#ifdef __PRAGMA_ENVIRONMENT
#pragma environment save
#pragma environment header_defaults
#else
#error "__DECC_INCLUDE_PROLOGUE.H:  \
        This compiler does not support pragma environment"
#endif


__DECC_INCLUDE_EPILOGUE.H:

#ifdef __PRAGMA_ENVIRONMENT
#pragma __environment restore
#else
#error "__DECC_INCLUDE_EPILOGUE.H:  \
        This compiler does not support pragma environment"
#endif
```

- #pragma extern_prefix {SAVE | RESTORE | __save | __restore | "prefix_to_use"}

  This feature is for use by RTL header file developers only to prefix function entry points with "prefix_to_use". Please note that the generated symbol name is all uppercase. The functionality should match that of the extern_ prefix pragma in DEC C++

  Note there is one known limitation with this feature: If the function is implicitly declared (ie, no prototype) then it does not get prefixed.

- Atomic/Interlocked builtin functions

Several of the builtins provided in previous versions of the compiler had some troublesome usability limitations. E.g. the __xxx_ATOMIC_yyy family of builtins generate a memory barrier both before and after the atomic update sequence, they always return a status value even though the status is only useful for the variant forms that specify a retry count, and they do not provide a mechanism for determining the old value (the value that was in the location when the atomic update succeeded). Despite these problems, they have been found useful in certain circumstances.

Instead of "improving" the previous builtins, they have all been retained as-is for compatibility, and a new set of builtins has been added to address the problems with the original set. For each builtin in the __xxx_ATOMIC_yyy family, DEC C V5.2 provides a new pair of builtins named __ATOMIC_xxx_yyy[_RETRY]. E.g. the original __ADD_ATOMIC_LONG is retained, and two new builtins __ATOMIC_ADD_LONG and __ATOMIC_ADD_LONG_RETRY have been added. Besides builtins corresponding to the previous __xxx_ATOMIC* names, there are new builtins corresponding to the old __TESTBIT* builtins and new __ATOMIC_EXCH_* (atomic exchange) that had no counterpart in previous versions. All of these new builtins have the following characteristics:

- No memory barriers are generated. If the application requires memory barriers before and/or after the atomic update, they must be explicitly coded (using inline assembly code or the __MB( ) builtin function).
- The return value from each function is the old value of the location that was updated.
- The functions that do not end in _RETRY continue to try until they succeed, so no status is returned.
- Those that end in _RETRY take two additional parameters specifying the number of times to attempt the update, and the address of an integer status variable to contain the completion status. Note that while this specification makes for source code that suggests accesses to the status variable would always involve a memory access (since its address is passed to something that looks like a function), these builtins are processed in a way that allows the status variable to remain in a register.

There are also new builtins to implement efficient binary spinlocks: __LOCK_LONG, __LOCK_LONG_RETRY, __UNLOCK_LONG. And builtins to implement counted semaphores: __ACQUIRE_SEM_LONG, __ACQUIRE_SEM_LONG_RETRY, __RELEASE_SEM_LONG. See the header file builtins.h and the User's Guide for more information.

5.12.1 Changes in DEC C RTL Header Files for V5.2 of DEC C/C++

The release notes in this section describe changes to the header files shipped with DEC C V5.2 for OpenVMS Systems. These header files contain enhancements and changes made to the DEC C Run-Time Library for OpenVMS Systems.

New function prototypes and structure definitions which define new functionality in the DEC C Run-Time Library correspond to new functionality added to the DEC C Run-Time Library which is shipped with OpenVMS V7.0.

- New Header Files Added

  A total of 20 new header files were added to the DEC C RTL suite of header files. Header files were added for implementation of Amendment 1 of the ISO C standard, compatibility with UNIX systems, and for introduction of new functions. Table 1 lists those headers added for DEC C V5.2.

Table 1 New DEC C V5.2 Header Files

Header File	Description
<dirent.h>	Directory Manipulation Functions
<ftw.h>	File Tree Walking
<if.h>	Socket Packet Transport Mechanism
<if_arp.h>	Socket Address Resolution Protocol
<ioctl.h>	I/O Controls for Special Files
<iso646.h>	Alternative Spelling for Language Tokens
<libgen.h>	Filename Manipulation
<memory.h>	String Handling
<mman.h>	Mapping Pages of Memory
<nameser.h>	Maximum Domain Name Size
<pwd.h>	Password File Access Functions
<resolv.h>	Resolver Configuration File
<resource.h>	Declarations for Resource Operations
<strings.h>	String Handling

(continued on next page)

Table 1 (Cont.) New DEC C V5.2 Header Files

Header File	Description
<timers.h>	Clock and Timer Functions
<times.h>	File Access and Modifications Times Structure
<tzfile.h>	Time Zone Information
<utsname.h>	User Information
<wait.h>	Declarations for Process Waiting
<wctype.h>	Wide Character Classification and Mapping

- New Functions Defined In Header Files

  OpenVMS V7.0 introduces many new DEC C RTL functions which have been added to fulfill the request of application developers and to implement those functions defined by ISO C Amendment 1. These functions have been implemented on both OpenVMS Alpha V7.0 and OpenVMS VAX V7.0. These functions are documented in the *DEC C Run-time Library Reference Manual for OpenVMS Systems*.

```
basename()      herror()        seed48()       sysconf()
bcmp()          hostalias()     seekdir()      telldir()
bcopy()         hstrerror()     setenv()       tempnam()
btowc()         index()         sethostent()   towctrans()
bzero()         initstate()     setitimer()    truncate()
closedir()      ioctl()         setnetent()    tzset()
confstr()       jrand48()       setprotoent()  ualarm()
dirname()       lcong48()       setservent()   uname()
drand48()       lrand48()       setstate()     unlink()
endhostent()    mbrlen()        sigaction()    unsetenv()
endnetent()     mbrtowc()       sigaddset()    usleep()
endprotoent()   mbsinit()       sigdelset()    vfwprintf()
endservent()    mbsrtowcs()     sigemptyset()  vswprintf()
erand48()       memccpy()       sigfillset()   vwprintf()
ffs()           mkstemp()       sigismember()  wait3()
fpathconf()     mmap()          siglongjmp()   wait4()
ftruncate()     mprotect()      sigmask()      waitpid()
ftw()           mrand48()       sigpending()   wcrtomb()
fwide()         msync()         sigprocmask()  wcsrtombs()
fwprintf()      munmap()        sigsetjmp()    wcsstr()
fwscanf()       nrand48()       sigsuspend()   wctob()
getclock()      opendir()       socket_fd()    wctrans()
getdtablesize() pathconf()      srand48()      wmemchr()
gethostent()    pclose()        srandom()      wmemcmp()
getitimer()     popen()         strcasecmp()   wmemcpy()
getlogin()      putenv()        strdup()       wmemmove()
getpagesize()   random()        strncasecmp()  wmemset()
```

```
getpwnam()      readdir()       strsep()       wprintf()
getpwuid()      rewinddir()     swab()         wscanf()
getservent()    rindex()        swprintf()
gettimeofday()  rmdir()         swscanf()
```

The following functions are specific to OpenVMS Alpha V7.0. These functions are the implementations specific to 64-bit pointers. *(Alpha only.)*

```
_basename64()  _mbsrtowcs64()  _strpbrk64()   _wcsncat64()
_bsearch64()   _memccpy64()    _strptime64()  _wcsncpy64()
_calloc64()    _memchr64()     _strrchr64()   _wcspbrk64()
_catgets64()   _memcpy64()     _strsep64()    _wcsrchr64()
_ctermid64()   _memmove64()    _strstr64()    _wcsrtombs64()
_cuserid64()   _memset64()     _strtod64()    _wcsstr64()
_dirname64()   _mktemp64()     _strtok64()    _wcstok64()
_fgetname64()  _mmap64()       _strtol64()    _wcstol64()
_fgets64()     _qsort64()      _strtoll64()   _wcstoul64()
_fgetws64()    _realloc64()    _strtoq64()    _wcswcs64()
_gcvt64()      _rindex64()     _strtoul64()   _wmemchr64()
_getcwd64()    _strcat64()     _strtoull64()  _wmemcpy64()
_getname64()   _strchr64()     _strtouq64()   _wmemmove64()
_gets64()      _strcpy64()     _tmpnam64()    _wmemset64()
_index64()     _strdup64()     _wcscat64()
_longname64()  _strncat64()    _wcschr64()
_malloc64()    _strncpy64()    _wcscpy64()
```

While each of these functions are defined in the DEC C V5.2 header files, those definitions are protected by using *if __VMS_VER >= 70000000* conditional compilation.

- Usage of Feature-Test Macros

  The header files shipped with DEC C V5.2 have been enhanced to support feature test macros for selecting standards for APIs, multiple version support and for compatibility with old versions of DEC C or OpenVMS. Please see the DEC C Run-time Library Reference Manual, section 1.5 "Feature-Test Macros for Header-File Control", for a complete description of the feature test macros that are available.

- Different Default Behavior After OpenVMS V7.0

  The functions *wait()*, *kill()*, *exit()*, *geteuid()*, and *getuid()* have new default behavior for programs which are recompiled under OpenVMS V7.0 or later. To the retain the old behavior, use the _VMS_V6_SOURCE feature-test macro, as described in the reference manual.

- Upgrade to Support 4.4BSD Sockets

As of OpenVMS V7.0, the socket definitions in the socket family of header files has added support for 4.4BSD sockets. To instruct the header files to use this support, define either *_SOCKADDR_LEN* or *_XOPEN_SOURCE_EXTENDED* during the compilation.

The functions *gethostbyaddr()*, *gethostbyname()*, *recvmsg()*, *sendmsg()*, *accept()*, *bind()*, *connect()*, *getpeername()*, *getsockname()*, *recvfrom()*, and *sendto()* have a second implementation which uses a new *sockaddr* structure defined in <socket.h>.

- Integration of Timezone Support

  The DEC C RTL on OpenVMS V7.0 has added full support for Universal Coordinated Time using a public domain timezone package. When compiling on OpenVMS V7.0, the functions *gmtime()* and *localtime()* have a second implementation which use extensions to the *tm* structure defined in <time.h>. To retain the ANSI C definition of this structure, define either *_ANSI_C_SOURCE* or *_DECC_V4_SOURCE*. Note that compiling with the */standard=ansi* qualifier implies *_ANSI_C_SOURCE*.

- Integration of ISO C Amendment 1 Behavior

  The DEC C RTL on OpenVMS V7.0 has added full support for Amendment 1 of the ISO C Standard. When compiling on OpenVMS V7.0, the functions *wcfstime()* and *wcstok()* have a second implementation which implement the semantic changes required by this amendment. To retain the XPG4 semantics of these functions, define either *_XOPEN_SOURCE* or *_DECC_V4_SOURCE*.

- Upgrade to Support 4.4BSD Curses

  Document changes to curses.h...

  Note

  The default definitions used during compilation for OpenVMS Alpha have been changed to *__VMS_CURSES* which is the same as OpenVMS VAX. To restore the original default curses package, the user must define *__BSD44_CURSES*.

- FILE Structure Changed to Increase Open File Limit

  Changes were made to the FILE type definition in <stdio.h> to support an extended open file limit in the DEC C RTL. As of OpenVMS V7.0, the number of files which can be open simultaneously will be raised from 256 to 65535 on OpenVMS AXP and 2047 on OpenVMS VAX. This number is based on the OpenVMS sysgen CHANNELCNT parameter which specifies

the number of permanent I/O channels available to the system. The maximum CHANNELCNT on OpenVMS AXP is 65535. On OpenVMS VAX it is 2047.

In order to support more than 256 open files, the field in the FILE type containing the file descriptor "_file" had to be changed from a char type to an int type.

The definition of the FILE type in stdio.h changed from:

```
typedef struct _iobuf {
    int      _cnt;           // bytes remaining in buffer
    char    *_ptr;           // I/O buffer ptr
    char    *_base;          // buffer address
    unsigned char   _flag;   // flags
    unsigned char   _file;   // file descriptor number
    unsigned char   _pad1;   // modifiable buffer flags
    unsigned char   _pad2;   // pad for longword alignment
} *FILE;
```

to:

```
typedef struct _iobuf {
    int      _cnt;           // bytes remaining in buffer
    char    *_ptr;           // I/O buffer ptr
    char    *_base;          // buffer address
    unsigned char   _flag;   // flags
    unsigned char   _padfile; // old file descriptor numbe
    unsigned char   _pad1;   // modifiable buffer flags
    unsigned char   _pad2;   // pad for longword alignment
    int             _file;   // file descriptor number
} *FILE;
```

This change was coded using the __VMS_VER macro. As such programs compiled with a version of stdio.h containing support for an increased open file limit can be linked with a version of the DEC C RTL which either does or does not contain this support. Programs compiled with a version of stdio.h providing the new FILE type definition which link on earlier OpenVMS versions obviously not be able to make use of this new functionality.

- Header <stdlib.h> No Longer Includes <types.h>

  As part of feature test macro work, <stdlib.h> no longer includes <types.h>. This will affect some DEC C V5.0 programs that included stdlib.h and expected a type such as pid_t to be defined. The user must change their source to explicitly include <types.h>.

5.13 Enhancements in V5.0

DEC C V5.0 contains the following enhancements:

- Change to the meaning of /standard=portable

  The meaning of /standard=portable was previously documented as putting the compiler in VAX C mode and enabling the portability group of messages. The DEC C compiler for OpenVMS Alpha prior to V5.0 implemented this behavior, while the DEC C compiler for OpenVMS VAX implemented this qualifier by putting the compiler in relaxed ANSI mode and enabling the portability group. Feedback from users overwhelmingly indicated a preference for the behavior of the VAX compiler. Therefore the V5.0 compiler for OpenVMS Alpha has been changed to behave the same as the VAX compiler, and the documentation for both platforms is being updated to reflect this.

- Major upgrade to the preprocessor

  The part of the compiler that implements preprocessing constructs has undergone a major upgrade. In most ways, the effect of this change should be invisible. Unless you have encountered problems with the preprocessor in previous releases that are now fixed, you should not expect to see much change except perhaps some additional compile-time improvement if you use heavily redundant #includes, or if you use #pragma message to control message reporting for preprocessor constructs. Because of the nature of the changes made, errors and warnings that are detected during preprocessing will typically be reported with different message text and different message identifiers from previous releases (including the previous field test compiler). If your code relies on the identifiers or text for messages issued by the preprocessor, you will have to assess the new messages and their identifiers to get equivalent behavior.

  The general nature of the changes to the preprocessor are to improve its reliability and its compatibility with traditional Unix C compilers, without compromising its ANSI C adherence. In particular:

  - Explicit .I file ouput

    Much more attention has been given to the content of .I files produced by the /preprocess_only qualifier, such that compiling a .I file should more closely mimic the effect of compiling the original source. This includes issues such as the following:

    - Generation of whitespace to separate tokens only where necessary to prevent accidental token pasting when the .I file is compiled.
    - Generation of # num directives and blank lines to keep a better correspondence to the original source.
    - Processing of pragmas and directives (including builtins and dictionary) such that compiling the .I file in the absence of the original header files and/or CDD repository will produce the same effect as compiling the original source in its own environment of header files and repositories.
    - #pragma message, standard, and nostandard are now also respected under /preprocess_only, so that spurious diagnostics are not produced when making a .I file.
- Token pasting

  More natural treatment of token-pasting operators that do not produce valid tokens: the pasting is simply ignored.
- Conditional preprocessing directives within macro argument lists

  More flexible treatment of the appearance of #if, #ifdef, #ifndef, #else, #elif, and #endif directives within the actual parameter list of a function-like macro invocation that spans multiple lines: the directives take effect. There is no ANSI-required behavior for such constructs, and they can easily appear when a function is changed to a function-like macro. Formerly an E-level diagnostic complaining about the syntax of the directive was issued.
- Missing macro parameters

  More natural treatment of function-like macro invocations with missing actual parameters: each missing parameter is treated like an object-like macro with an empty replacement list.
- Macro expansions in #include and #line

  More complete treatment of preprocessing directives like #include and #line, in the cases where a sequence of tokens requiring macro expansion occurs, and the result of the macro expansion is to be matched against one of the "fixed" forms.
- Error recovery

Better error recovery for preprocessor-generated diagnostics. In some cases the severity of a similar condition diagnosed by the previous version of the preprocessor has been reduced from an error to a warning or informational because the repair is what would be expected at that level. In particular, C preprocessors are sometimes applied to source code that is not really C code - the expectation is that the preprocessor would give at most a warning or informational, and the detection of an error condition resulting from the fixup made by the preprocessor can safely be left to the compiler's syntactic and semantic analysis phases.

- Macro expansions in #pragma

  More usual treatment of #pragma directives: the tokens are not subject to macro expansion. For pragmas that already have a well-established and documented behavior under DEC C, macro expansion is still performed. But for new DEC C pragmas and pragmas offering compatibility with other C compilers, macro expansion is not performed (since most other C compilers do not perform it). If an identifier used as the name of a pragma matches the name of a pragma that is defined not to have macro expansion performed, then no expansion will be performed. But unless /standard=common is specified, if the identifier is not the name of such a pragma, and it is the name of a currently-defined macro, then that macro gets expanded and the resulting token is compared to the following list of pragma names to determine if the rest of the pragma tokens should be macro expanded. This gives maximum compatibility with existing code, but allows the general behavior to be brought more in line with common practice. The pragmas that will continue to be subject to macro expansion are listed below.

  - _KAP
  - standard
  - nostandard
  - member_alignment
  - nomember_alignment
  - dictionary
  - inline
  - noinline

- module
- message
- extern_model
- builtins
- linkage
- use_linkage
- define_template (C++ only)

- #pragma environment

  New #pragma environment. This new pragma allows saving and restoring the state of all pragmas for which the compiler supports a save/restore stack. It has two additional keywords to specify a state that is consistent with the needs of system header files, or to specify a state that is the same as what was established by command line switches at the start of compilation. The primary purpose is to allow the authors of header files describing data structures that will be accessible to library code to establish a consistent compilation environment in which their headers will be compiled, without interfering with the compilation environment of the user. The syntax is:

```
#pragma environment save
#pragma environment restore
#pragma environment header_defaults
#pragma environment command_line
```

  The save and restore keywords cause every other pragma that accepts save and restore keywords to perform a save or restore operation. The header_defaults keyword sets the state of all those pragmas to what is generally desirable in system header files. This corresponds to the state the compiler would be in with no command line options specified and no pragmas processed - except that #pragma nostandard is enabled. The command_line keyword sets the state of all such pragmas to what was specified by the command line options. See the Users's Guide and help files for a description of pragmas that accept save and restore keywords, and for command line options that control behaviors that are also controllable by pragmas.

- __unaligned type qualifier:

A new type qualifier called "__unaligned" has been introduced which can be used in exactly the same contexts as the ANSI C "volatile" and "const" type qualifiers. It can be used either on a declaration or in a cast to tell the compiler specific places where it should not assume natural alignment on a pointer dereference, without making it apply to all dereferences the way the command line switch, /ASSUME=noaligned_objects would.

- DEC C V5.0 introduces support for in-line assembly code, commonly called ASMs on UNIX platforms. The use of ASMs in DEC C on OpenVMS Alpha is compatible with their use on the Digital UNIX platform. The opcodes and register names/numbers are accepted in either upper or lower case by DEC C on both platforms since conventions have traditionally differed between them. Both platforms also accept both the R-number register macro names common to VMS documentation and the so-called "software" names common to the UNIX documentation.

  Since ASMs are implemented with a function call syntax, you must include a new header file, c_asm.h, containing prototypes for the three types of ASMs, and a special pragma in order to use them.

  The syntax (in the header file) looks like this:

```
__int64 asm(const char *, ...); /* integer ops */
float fasm(const char *, ...);  /* float ops   */
double dasm(const char *, ...); /* double ops  */

#pragma intrinsic (asm)
#pragma intrinsic (fasm)
#pragma intrinsic (dasm)

where:

const char *
the first argument to the asm, fasm or dasm contains
the instruction(s) which are to be generated in-line
and the metalanguage which describes the interpretation
of the arguments.  This must be a literal (compile-time)
string or a preprocessor macro expanding to a literal
string.

....
subsequent arguments are made available to the
instructions specified in the first argument
according to the calling standard conventions, with
the first of these arguments corresponding to the
first actual argument in the calling standard.  I.e.
the character string specifying the instructions to
execute does not in itself count as an actual
argument.
```

The #pragma intrinsic directives notify the compiler that the named functions (asm, dasm, and fasm) are not normal user defined functions, and that the special asm processing should be applied. As a result, a compilation that includes c_asm.h cannot declare ordinary user functions with these names. Other modules in the same program that do not include the header file may declare functions with these names, however, since asm processing generates inline code within the calling module and never creates object module references to these names.

There is a simple metalanguage for naming registers:

```
<metalanguage>    : <register_number>
                  | <register_macro>
                  ;

<register_number> : "$" number
                  ;

<register_macro>  : "%" <macro_sequence>
                  ;

<macro_sequence>  : number
                  | <register_name>
                  | "f" number | "F" number
                  | "r" number | "R" number
                  ;
```

```
<register_name>   : "v0"   /* R0, F0 return value */
                    /* scratch registers */
                  | "t0"  | "t1"  | "t2"  | "t3"
                  | "t4"
                    /* save registers */
                  | "s0"  | "s1"  | "s2"  | "s3
                  | "s4"
                  | "s5"  | "s6"  | "s7"  | "s8"
                  | "s9"  | "s10" | "s11" | "s12"
                  | "s13"
                    /* arg registers, R16-R21 */
                  | "a0" | "a1" | "a2" | "a3"
                  | "a4" | "a5"
                    /* R26 is return addr  */
                  | "RA" | "ra"
                    /* R27 is procedure value */
                  | "PV" | "pv"
                    /* R25 is arg info     */
                  | "AI" | "ai"
                    /* R29 is frame pointer */
                  | "FP" | "fp"
                    /* R31 contains zero    */
                  | "RZ" | "rz" | "zero"
                    /* R30 is stack pointer */
                  | "sp" | "SP" | "$sp" | "$SP"
                  ;
```

The literal string which contains instructions, operands, and metalanguage must follow the general form:

```
<string_content> :  <instruc_seq>
                 |  <string_content> ";" <instruc_seq>
                 ;

<instruc_seq>    :  instruction_operand
                 |  directive
                 ;
```

Where an instruction_operand is generally recognized as an assembly language instruction, separated by whitespace from a sequence of comma-separated operands.

Note that it is possible to code multiple instruction sequences into one literal string.

Please note that there are semantic rules associated with ASMs, as well as syntax rules.

The first argument to the asm call is interpreted as the instructions to be assembled in the metalanguage, and must be fully understood by the compiler at compile-time. That is indirections, table lookups, structure dereferences etc are NOT allowed.

The remaining arguments are treated like normal function arguments, that is they are loaded into the argument registers according to the calling standard, with the first argument following the string treated as the first argument for calling standard purposes. For example:

```
In the following test, the 6 arguments are
loaded into arg registers %R16 through %R21
(equivalently %a0 through %a5 or just %0
through %5), and the result of each sub
expression is stored in the value return
register, %R0 (equivalently %v0).

if (asm("addq %R16, %R17, %R0;"
        "addq %R18, %R0,  %R0;"
        "addq %R19, %R0,  %R0;"
        "addq %R20, %R0,  %R0;"
        "addq %R21, %R0,  %R0;",
                  1,2,3,4,5,6) != 21){
    error_cnt++;
    printf ("Test failed\n");
}

     With double precision operands the same
     testcase would be written:

if (dasm("addt %F16, %F17, %F0;"
         "addt %F18, %F0,  %F0;"
         "addt %F19, %F0,  %F0;"
         "addt %F20, %F0,  %F0;"
         "addt %F21, %F0,  %F0;",
          1.0,2.0,3.0,4.0,5.0,6.0) != 21.0){
    error_cnt++;
    printf ("Test failed\n");
}
```

Note that the following does not work, and produces a compile-time warning stating that r2 is used before it is set, because the arguments are loaded into the arg registers (r16-r21), not r2. Similarly, the result of the addq instruction would not be assigned to z because the destination of the addq instruction is not the return value register (%v0 or %R0).

```
z =  asm("addq %r2, %a1 %r5",x=10,y=5);
```

The correct way of doing this is to specify an argument register number in place of r2, and make sure the code puts any result in the result register appropriate to the type of the asm (%v0 for integer, %f0 for float and double).

```
One exception to the above rule is that the return address
register (%ra or %r26) may be used in an asm without being
set.  This is handled specially; it tells the compiler to
get the return address that was in %R26 at the time the
function containing the asm was invoked.  Thus

        z = asm("mov %ra, %v0")

puts the address to which the containing function will
return into z, regardless of whether or not register R26
still holds that address at this point in the code.
```

As noted, a result register must be specified in the metalanguage for the result to appear in the expected place if the ASM is used as a value-producing function. For instructions which do not take any argument and which do not have a return type, simply leave out the arguments. For example:

```
        asm("MB");
```

- /NESTED_INCLUDE_DIRECTORY= (PRIMARY_FILE, INCLUDE_FILE, NONE) has been enhanced.

  Extended to accept the NONE keyword. This specifies that the compiler skips the first step of processing #include "file.h" directives. It starts looking for the included file in the /INCLUDE_DIRECTORY directories. It does not start by looking in the directory containing the including file nor in the directory containing the top level source file.

```
Default:
  /NESTED_INCLUDE_DIRECTORY=INCLUDE_FILE
  (current behavior)

UNIX equivalent:
  -nocurrent_include  /NESTED_INCLUDE_DIRECTORY=NONE
```

- /MMS_DEPENDENCY[=(FILE[=filespec] | [NO]SYSTEM_INCLUDE_ FILES ,...)] /NOMMS_DEPENDENCY has been added.

  The qualifier /MMS_DEPENDENCY corresponds tells the compiler to produce a dependency file. The format of the dependency file is similar to that on Digital UNIX.

```
object_file_name :<tab><source file name>

object_file_name :<tab><path to first include file>
object_file_name :<tab><path to second include file>
```

The FILE subqualifier specifies where to save the dependency file. The default file extension for a dependency file is .mms. Other than using the different default extension, this qualifier uses the same procedure as /OBJECT and /LIST for determining the name of the output file.

The SYSTEM_INCLUDE_FILES subqualifier specifies whether or not to include dependency information about system include files. That is, those included with #include <filename>. The default is to include dependency information about system include files.

```
Default: /NOMMS_DEPENDENCY

UNIX equivalents:
  -MD         /MMS_DEPENDENCY
  -M          /MMS_DEPENDENCY=(FILE=SYS$OUTPUT)
  -MM         /MMS_DEPENDENCY=(FILE=SYS$OUTPUT,-
                              NOSYSTEM_INCLUDE_FILES)
  default     /NOMMS_DEPENDENCY
```

- /[NO]LINE_DIRECTIVES has been added

  This qualifier governs whether or not # ddd directives appear in preprocess output files. Currently, there is no way to specify the form of the line directives. DEC C always generates the "old-style" # ddd form, even in ANSI modes.

```
Default:    /LINE_DIRECTIVES

UNIX equivalents:
    -P                  /NOLINE_DIRECTIVES
    -E                  /LINE_DIRECTIVES
```

- /COMMENTS=(AS_IS,SPACE)/NOCOMMENTS has been added.

  This qualifier governs whether or not comments appear in preprocess output files. If they do not appear, it specifies what replaces them.

AS_IS specifies that the comment appears in the output file. SPACE specifies that a single space replaces the comment in the output file. /NOCOMMENTS specifies that nothing replaces the comment in the output file. This may result in inadvertent token pasting.

The C and C++ preprocessor may replace a comment at the end of a line or a line by itself with nothing, even if /COMMENTS=SPACE is specified. Doing so cannot change the meaning of the program.

```
Default:
    /NOCOMMENTS in the VAXC and COMMON modes of the
    C compiler /COMMENTS=SPACE for C++ and the ANSI
    modes of the C compiler.  An explicit /COMMENTS
    on the command line defaults to /COMMENTS=AS_IS.

UNIX equivalents:
    -oldcomment         /NOCOMMENTS
    -C                  /COMMENTS=AS_IS
    default             /COMMENTS=SPACE
```

- /[NO]VERSION has been added.

  This is a completely new qualifier intended to make it easier for users to report which compiler they are using.

  This qualifier causes the compiler to print out via printf the compiler version and platform. The compiler version is the same as in the listing file.

```
Defaults:               /NOVERSION

UNIX equivalents
    -V                  /VERSION
    default             /NOVERSION
```

- /CHECK=UNINITIALIZED_VARIABLES has been added.

  This qualifier determines whether or not unitialized automatic variables are initialized to the value 0xfffa5a5afffa5a5a. This value is a floating NaN so will cause a floating point trap if used. If used as a pointer, it will likely causes an ACCVIO. This is a debugging aid.

```
Defaults:          /NOCHECK

UNIX equivalents:
   -trapuv         /CHECK=UNINITIALIZED_VARIABLES
```

- New Predefined Macros

  There are new predefined macros __DECC_VER and __VMS_VER, which map compiler version numbers and VMS version numbers respectively into an unsigned long int. The compiler version number is extracted from the compiler ident and the VMS version macro is obtained by calling sys$getsyiw(SYI$_VERSION). These string values are then changed into an integer in an implementation defined manner. It is intended that newer versions of the compiler and VMS will always have larger values for these macros. If for any reason the version string returned by sys$getsyiw(SYI$_VERSION) or even the compiler's own ident string cannot be analyzed, then the corresponding predefined macro will be defined, but it will have a value of 0. Please note that pre-5.0 compilers do not define these macros, so it is possible to distinguish a pre-5.0 compiler from a compiler that is given a bad version string by using #ifdefs or the defined operator.

```
/*__DECC_VER is not defined before V5.0
  test for a compiler V5.1 or higher */
#ifdef __DECC_VER
    #if (__DECC_VER >= 50100000)
        / * code */
    #endif
#endif

/* test for VMS 6.2 or higher */
#ifdef __VMS_VER
        #if __VMS_VER >= 60200000
                /* code */
        #endif
#endif
```

- Compile Time Performance Improvements

  Some compile-time improvements have been made for V5.0. The most notable improvement is that the preprocessor now is usually able to determine if a particular #include file that has already been processed once was guarded by the conventional "#ifndef FILE_SEEN, #define FILE_SEEN, #endif" sequence. When the compiler detects this pattern of use the

first time a particular file is included, it remembers that fact as well as the name of the macro. The next time the same file is included, the compiler checks to see if the "FILE_SEEN" macro is still defined, and if so it does not reopen and reread the file. Note that if the initial test is in the form "#if !defined" instead of "#ifndef", then the pattern is not recognized. In a listing file, #include directives that are skipped because of this processing are marked with an "X" just as if the #include line itself were excluded.

- Run Time Performance Improvements

  The compiler backend has been upgraded with general improvements to the optimizer and with specific improvements to the inliner, listed below.

- Enhancements to inline optimization:

  The heuristics for automatic inlining (/inline=automatic) have been improved somewhat to select inlining opportunities that are more likely to produce improved execution speed without increasing code size too much.

  The keywords "size" and "speed" have been added to the command line qualifier /OPTIMIZE=INLINE. The keyword "size" gives behavior similar to the intent of the keyword "automatic" in previous compilers, although it is somewhat more conservative in most cases ("automatic" is now treated as a synonym for "size"). "size" inlines functions when the compiler determines this it would improve run-time performance without significantly increasing the size of the program. The keyword "speed" performs more aggressive inlining for run-time performance, even when it may significantly increase the size of the program.

```
/[NO]OPTIMIZE=INLINE={size | speed  | automatic
                      | manual | none | all}
```

  #pragma noinline may be used to prevent inlining of any particular functions under the 3 compiler-selected forms of inlining. And #pragma inline (or the __inline qualifier as used in C++) may be used to request inlining of specific functions under the automatic or manual forms of inlining. The "all" keyword is not generally recommended as it may often increase both compilation resources and runtime size unacceptably.

- XPG4 Support

DEC C V5.0 supports the worldwide portability interfaces described in the X/Open CAE Specifications: System Interfaces and Headers, Issue 4; System Interfaces Definitions, Issue 4; and Commands and Utilities, Issue 4. These interfaces allow an application to be written for international markets from common source code. This model of internationalization is the same as found on many UNIX systems, including Digital UNIX.

Note: The new support for internationalization requires OpenVMS V6.2-FT2 or later.

5.14 Enhancements in V4.1

- #dictionary support

  DEC C V4.1 introduces support for inclusion of CDD records. The syntax and features are the same as for DEC C V4.0 on OpenVMS/VAX, except that V4.1 will also accept two new keywords, text1_to_array and text1_to_char.

```
#[pragma] dictionary "pathname" [null_terminate]
             [name(struct_name)]
             [text1_to_array | text1_to_char]

pathname: a quoted pathname for a CDD record to be
          extracted.

null_terminate: an optional keyword which adds an
          additional byte for the null character
          when a data type of text is extracted.

name():   an optional keyword to supply an alternate
          tag name or declarator, struct_name, for the
          outer level of a CDD structure.

text1_to_char: an optional keyword which forces the
          CDD type text to be translated to char, rather
          than array of char if the sizen is 1.  This is
          the default, unless null_terminate was
          specified.

text1_to_array: an optional keyword which forces the
          CDD type text to be translated to type array
          of char, even when the size is 1.  This is the
          default when null_terminate is specified.
```

5.15 Enhancements since V1.3A

- Expanded IEEE support

  The /IEEE_MODE switch allows users to control the way IEEE exceptional values are generated and handled.

```
/IEEE_MODE=
          FAST (default)
          UNDERFLOW_TO_ZERO
          DENORM_RESULTS
          INEXACT

  FAST
          During program execution, only finite values
          (no infinities, NaNs, or denorms) are created.
          Exceptional conditions, such as floating point
          overflow and divide by zero, are fatal.


  UNDERFLOW_TO_ZERO
          Generate infinities and NaNs. Flush denorms
          and underflow to zero without exceptions.
          Only modules containing main() are affected
          by this switch.  Modules not containing
          main() compiled with this switch default
          to DENORM_RESULTS.

  DENORM_RESULTS
          Generate infinities, NaNs and denorms. Do not
          trap on inexact.

  INEXACT
          Generate infinities, NaNs and denorms. Do
          trap on inexact. This is the slowest mode of
          IEEE support, and should only by used if
          inexact trapping is absolutely required.
```

- Enhanced floating point rounding support DEC C V4.1 introduces the /ROUNDING_MODE switch to allow users to specify how IEEE floating point values should be rounded.

```
/ROUNDING_MODE=NEAREST (default)
               DYNAMIC
               MINUS_INFINITY
               CHOPPED

/ROUNDING_MODE lets the user specify which of the
above four rounding modes will be employed (if
/FLOAT=IEEE_FLOAT has also been specified).
```

Note that when either /FLOAT=G_FLOAT (the default) or D_FLOAT have been specified, the only allowable rounding mode is NEAREST.

- 128 bit floating point support

```
 /L_DOUBLE_SIZE=128 (default)
                64
```

DEC C V4.1 supports "long double" as a 128 bit software emulated type, x_float. With previous compilers, "long double" was synonymous with "double" . This is no longer the case.

Please note that default behavior of "long double" is now x_float, and must be explicitly overridden to g_float, d_float, or t_float by specifying /L_DOUBLE_SIZE=64. Please see the preceding section on Targeting to Pre-6.1 Systems for a discussion on implications of this switch.

The /FLOAT= switch continues to specify which of g_float, d_float, or t_float should be generated for "double" variables.

- /PSECT_MODEL=[NO]MULTILANGUAGE

  /PSECT_MODEL=MULTILANGUAGE instructs the compiler to employ a psect allocation scheme for shared, overlaid psects which is compatible with psects generated by other Alpha VMS compilers.

  Since C structs to be shared with a FORTRAN application are padded (and FORTRAN COMMON blocks are not), usage of this switch prevents a possible linker error when multiple images created by two language translators share a common psect.

  The default is /PSECT_MODEL=NOMULTILANGUAGE, which is the old behavior.

- Enhanced compiler diagnostics

The check group, under /WARNING=ENABLE=CHECK has been embellished to detect a number of new questionable user coding practices.

- PRAGMA NOMEMBER_ALIGNMENT value

  The #pragma nomember_alignment pragma has been augmented to accept as an argument any one of the following case-insensitive values:

  ```
  byte (the default), word, longword, quadword, octaword
  ```

  This value directs the compiler to align the structure at the specified boundary.

  The compiler will, however, continue to byte align the members of the structure, as per nomember_alignment. With this behavior it is now possible to specify structures whose size is not a multiple of their alignment. This is problematic in the case of arrays of such structs. Consequently, the compiler will generate an -E- level diagnostic, along with a suggestion for the amount of required padding the user should specify in the struct to ensure size compatibility.

  When using this new alignment option, it is also suggested that the /WARNING=ENABLE=CHECK commandline switch be used to allow the compiler to check for poorly aligned structure members.

- Support for C++ style comments

  DEC C for OpenVMS Alpha supports C++ style comments in all modes except
  /STANDARD=ANSI89, because they are not allowed by the C standard.

- Suppress object module references to unused extern declarations

  DEC C no longer requires definitions for unreferenced declared objects. The following program no longer produces the linker diagnostic %LINK-W-UNDFSYMS:

  ```
  extern int x;
  #include <stdio.h>
  int main(void)
  {
    printf ("Hello, world\n");
  }
  ```

- In addition to #pragma inline function_name, the user can now suggest inlining of a function with __inline. Below, both func1 and func2 are candidates for inlining:

```
__inline void func1(void) {}
void func2 (void) {}
#pragma inline func2
```

- The redundant use of a type qualifier of a pointer (e.g., int * const const p;) produces a warning diagnostic.

- Redundant type specifiers (e.g. int int x;) now produces warning, not error, diagnostics. Warning diagnostics are suppressible via /WARNING=DISABLE, whereas error diagnostics are not.
- Functions declared volatile or const via a typedef:

```
typedef int (F) ();
const volatile F abs;
```

  now produce a CONSTFUNC or VOLATILEFUNC diagnostic.
- A new level of optimization, /OPT=LEVEL=5 . When selected, this includes all optimizations through level 4, plus software pipelining. See the User's Guide, table 1-9 for more information.

- Numerous other compiler optimizations, including:
  - small integer operations (char and short)
  - numerous code improvements accessing struct members
  - improved peephole optimizations

- builtins.h no longer includes ints.h .

5.16 Problems fixed in V7.3

A number of bugs have been fixed in this version. These include:

- A small number of compiler crashes and wrong optimized code problems were fixed.

- 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.

- The value of the predefined macro __STDC__ is now set to 2 by default or when /STANDARD=RELAXED is specified. The HP C V7.1 release used to set the macro to a value of 1 in these cases. This was an error because the C Standard states that the value of 1 should indicate a conforming implementation.

  When /STANDARD=RELAXED is specified, the compiler accepts extensions that prevent it from being a strictly conforming implementation. When /STANDARD is used with a value other than RELAXED, the value of __STDC__ is the same as that for the V7.1 release.

- Certain invalid structure declarations could cause the compiler to crash when /STANDARD=VAXC was specified. This problem has been corrected, and the compiler now emits a warning.

- If a compilation changed certain diagnostics to -E- level and then later disabled them, the compiler could crash. This problem has been corrected.

- If a nested structure element was passed to a function whose linkage was modified with a #pragma linkage directive, the compiler could crash. This problem has been corrected.

5.17 Problems fixed in V7.1

A number of bugs have been fixed in this version. These include:

- Correction to <time.h>

  This kit supplies an updated version of DECC$RTLDEF.TLB that addresses various issues in an upward-compatible fashion. However, there are changes to the conditionalization of the layout and member names for the type "struct tm" (from <time.h>), as well as conditionalization of the implementation of the gmtime( ) and localtime( ) functions [and their reentrant variants gemtime_r( ) and localtime_r( )] to be used, that might affect some programs. As a general precaution to minimize possible problems, it is recommended that if any source module that #includes <time.h> is recompiled, all modules in the application that #include <time.h> should also be recompiled.

  The purpose of the changes is to address an interrelated set of C++ functionality issues, C library user-namespace, and binary compatibility issues in a way that will be consistent between C and C++ going forward with new versions of the compilers and run-time libraries.

- C++ functionality:

  The C++ standard library's "time_put" time formatting facet relies on the ability to access timezone information from values of type "struct tm". Changes were made to ensure that C++ compilations always select the implementations of localtime( ) and gmtime( ) [and their reentrant _ r variants] that support timezone information, and that the declaration of type "struct tm" always has space for the timezone-related members.

- User-namespace:

  The members of struct tm that support timezone information are not part of the C standard. They are actually a common extension from BSD Unix, with conventional names of tm_gmtoff and tm_zone. Although the C standard allows an implementation to define additional members in struct tm, it does not reserve the prefix "tm_" for that purpose. So a strictly-conforming C program could #define tm_zone as a macro expanding into something that would cause a syntax error when encountering tm_zone as the name of a member in a struct type. Or conversely, a program compiled with the strict C namespace in effect could refer to the members by their BSD names, without getting a diagnostic, and infer that the names were defined under the C standard.

  Previous versions of <time.h> handled this issue by suppressing the declaration of these two members entirely under the strict C namespace (_ANSI_C_SOURCE). This new version has an option to declare the timezone-related members with names reserved to the implementation when compiled with the strict C standard namespace in effect: instead of tm_gmtoff and tm_zone it declares the members as __tm_gmtoff and __tm_zone (which is the convention on Tru64 and Linux systems). Using this feature, the size of type struct tm is the same regardless of the choice of the strict standard C namespace or the enabling of extensions.

- Binary compatibility:

  Previous versions of <time.h> that unconditionally suppressed the two non-standard members in struct tm under the strict C namespace resulted in potential binary incompatibility between modules compiled with/without the strict C namespace. The option in the new <time.h> of just changing the names of the members to meet the requirements

of the strict namespace makes modules compiled with either setting binary compatible, but it introduces potential for binary incompatibility with pre-existing object modules that cannot be recompiled from source.

Note that modules compiled with different sizes for struct tm only have potential for encountering a problem; in typical usage no problem will occur because struct tm is not often embedded within a struct or used as an element of an array, with that struct or array then shared beteen compilation units. Typical code using "struct tm" only accesses it locally through the pointer returned by the library functions, or contructs a local instance that is then passed by address to library functions. Those kinds of uses are not affected by a size difference that occurs between different compilations.

But to help deal with possible situations where a real binary compatibility problem is encountered, and it is not feasible to recompile all of the modules involved, the "struct tm" declaration in `<time.h>` can be forced either to the short version (without the timezone members) or to the long version. Defining the macro "_TM_SHORT" before the header is included will give the short version, and defining "_TM_ LONG" before the header is included will give the long version. If neither macro is defined when the header is included, the behavior remains unchanged when using current versions of the CRTL: the strict namespace will declare the short version of the struct. However, it is expected that a future CRTL version will change this behavior always to declare the long version of the struct unless "_TM_SHORT" is defined.

As described above, the use of implementation-reserved identifiers to make the size of struct tm uniform regardless of whether or not the strict C namespace is in effect was necessary to fix a bug in C++. The DECC$RTLDEF.TLB headers are shared between the C and C++ compilers, but that bug did not affect the C compiler, and it was felt that the risk of encountering a binary incompatibility problem was slightly greater for C than for C++. In part, this is because unlike C++, the C compiler enables the strict C namespace under its /stand=ansi89 or /stand=c99 qualifiers as well as when any of the _XOPEN, _POSIX, or _ANSI_C standard-conformance macros are specified (see section 1.5 Feature-Test Macros for Header-File Control in the C Run-Time Library Reference Manual). Therefore the change to the struct layout is not enabled for the C compiler under current versions of the CRTL. The change will take effect for the C compiler under a future upgrade to the CRTL.

However, note that the _TM_LONG and _TM_SHORT macros affect both compilers, regardless of CRTL version. If your application contains C modules that #include <time.h> as well as C++ modules that do, it is advisable to recompile those modules with /define=_TM_LONG to avoid possible binary incompatibility between the C and C++ modules. And you can prepare now for the change in C coming with a future version of the CRTL by testing with _TM_LONG defined.

- __MEMxxx builtin functions treated length as signed.

  The builtin functions __MEMMOVE, __MEMCPY, and __MEMSET were erroneously generating code to sign-extend the length parameter before passing it to an OTS$ routine that interprets the length as a signed 64-bit value. The length parameter for the standard C library routines is of type size_t, which is unsigned int on OpenVMS. But the sign extension done by these builtins made a length greater than or equal to 2147483648 be seen as negative by the underlying OTS$ routines, and those routines treat a negative length as a no-op. Although lengths so large are unusual, they are possible and need to be supported.

  It is possible that some source code exists that actually computes negative length values, and relies on these values being treated as no-ops. That behavior is not supported by the standard, although it is a convention used in similar VMS library routines. With this bug fixed, such code will most likely ACCVIO at run-time. Such code needs to be changed to test the length value explicitly to determine if it is in a range that should be ignored, and either bypass the call or use a length of zero - that is the only way to assure correct operation.

  Note that the <string.h> header actually defines macros to replace invocations of the standard C functions memmove, memcpy, and memset by invocations of these builtins. And these functions are also recognized as intrinsics, so even without the macros the compiler by default would optimize them into builtins. Therefore, just recompiling a module will most likely introduce the effect of this fix, even if linking against the same version of the CRTL. It would be possible to #undef these macros (or change the invocations to enclose the function name in parenthesis to avoid macro substitution), and also use #pragma function or optimization controls to avoid use of the builtins and force a call to the CRTL implementation of these functions. However, the CRTL implementation will also be changed in a future version to correct this problem. The only reliable way to ensure

that a negative length value will be treated as a no-op instead of a large positive value when using these standard C functions is to modify the source to test the value explicitly - that's what the standard-conforming behavior requires.

Finally, note that this bug is not present in either the builtin or CRTL implementions of these functions on OpenVMS I64, so this fix makes the Alpha behavior match the I64 behavior.

- The compiler could generate incorrect code in some situations where a struct consisting of a single longword member is modified, and then the value of the member is accessed. If the struct was allocated to a register, the code could fail to account for the unused longword in the register. The following example produces code where the value of rec.p used in the comparison is not properly sign-extended when compiled without optimization:

```
#include <string.h>
int f () {
    struct s {void *p; } rec;
    memset (&rec, 0, sizeof (rec));
    return rec.p == 0;
}
```

- The /NAMES=SHORTENED command line qualifier was not causing module names (either when specified by #pragma module or when derived from the source file specification) to be shortened, but simply truncated. This was an oversight in the original implementation of the qualifier, since distinct module names are significant to the librarian and when linking.

- The processing of C99 initializers with designators, in cases where later designators needed to override earlier designators, could cause the compiler to crash.

- The left operand of a comma operator was not checked for volatile accesses, and could be optimized away even if it contained accesses made through volatile lvalues.

- Certain unusual forms of function-call syntax, when used to invoke a function with non-standard linkage specified by #pragma use_linkage, could result in generated code that failed to use the specified linkage. And using a function invocation as the expression in a return statement could cause a compiler crash if the return type was a struct type, and the function used in the return expression was defined with non-standard linkage.

- Some cases of incompatibily between old-style functions and prototype-style declarations were not being diagnosed. Also, in C99 mode the return type on a function declaration could be omitted and default to int without a diagnostic, which is not permitted by the C99 standard.

- The __PAL_BUGCHK builtin function for Alpha should always have taken a parameter, but was incorrectly declared and implemented as taking no parameters. As a result, any use of this builtin (which obviously cannot be used in user code) would produce a bugcheck code that was arbitrary (whatever happened to be in R16 at the time). This has been corrected in <builtins.h> and the compiler to take a single parameter of type unsigned __int64.

- If the /FIRST_INCLUDE qualifier specified an empty file, the compiler would crash.

- The output generated for the /MMS_DEPENDENCIES command line qualifier no longer contains a form-feed character every 60 lines.

- An attempt to initialize an object of the C99 type _Bool having static storage duration, using an initializer containing an address constant, caused the compiler to crash. This was because the compiler attempted to put a relocation on the byte-sized _Bool object for the linker to fill in, and there is no such relocation.

  Such an initialization is permitted by the C99 standard. But there is an implementation issue in that the conversion from pointer to _ Bool is specified as the result of comparing the pointer to a null pointer constant (pointer != 0); and there is no relocation available to compute that expression. On the other hand, the only way that an address constant could compare equal to a null pointer constant would be if it were the address of an external identifier that remained undefined. Under the standard, when a translation unit refers to an identifier with external linkage, then somewhere among all the translation units that make up the entire program there must be exactly one definition of that identifier, or else the behavior is undefined. So under the standard, a conforming implementation is permitted to get the "wrong" result in the event that the external reference remains undefined.

  One way to get the "right" result would involve adding a new relocation for this special case. Another would be to have the C compiler generate a LIB$INITIALIZE routine to compute the result at program startup. The latter is what C++ compilers, and many C compilers that have a shared implementation with C++, usually do. We have chosen to do neither of these, and instead this construct gets flagged with a new informational message, "STATICBOOLADDR" by default in all modes that support type _Bool. The message is self-explanatory, especially if you compile with /WARN-VERBOSE, which includes suggested workarounds.

- Support for variadic function-like macros (the use of "..." in the parameter list and __VA_ARGS__ in the replacement list) was introduced in the C99 standard. This feature was mistakenly enabled in the ANSI89 and VAXC modes of the compiler. The feature has been disabled in those dialects in the new compiler.

- The compiler failed to warn about data loss when a case label constant was too large to fit in the promoted type of the switch expression.

- The compiler was incorrectly setting the NOPIC attribute on the $READONLY_ADDR$ psect it generates.

5.18 Problems fixed in V6.5

A number of bugs have been fixed in this version. These include:

- some very rare optimization problems that were still present in V6.4A where struct member assignments could be incorrectly optimized away
- some assertion failures in GEM when generating debugging symbols with optimization turned on
- a number of longstanding problems with character constants and wide character constants, including inability to use wide character constants in #if expressions
- #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.

5.19 Problems fixed in V6.4A

The V6.4 kit was found to have a number of significant problems that are fixed by this update. The first two of these problems involving incorrect optimization were actually regressions introduced in V6.2, but were not discovered until after V6.4 shipped.

- incorrect optimization of array element accesses

  In certain cases, optimization of accesses to array elements could produce incorrect code. The bug can manifest under the following conditions: an extern array declared with incomplete type has two assignments to the same array element, other than element zero, and one of the assignments uses a constant for the index while the other does not. Optimization incorrectly assumes that the assignments affect different array elements. If the same array element expressions are then used to fetch values, the compiler may replace one or more fetches with the value previously assigned, rather than taking into account the fact that the other assignment may have changed the stored value.

  This bug is a regression relative to V6.0. It was introduced in V6.2, and is present in the ECO kits for V6.2 and V6.2A, and in V6.4-005. The workaround is to compile /nooptimize. It is first fixed in V6.4A.

  An example:

```
extern int table[];    // Incomplete extern array
#define IDX 9          // Constant non-zero index
int bug(int i) {
    table[i] = 2;      // Assign using variable index
    table[IDX] = 1;    // Assign using constant index
```

```
    return table[i]    // Fetch using variable index
         + table[IDX]; // Fetch using constant index
    // Bug: Optimization returns constant 3, although
    //      correct result is 2 if i == IDX.
}
```

- incorrect optimization of struct member accesses

  In certain cases, optimization of struct member accesses could produce incorrect code. The exact circumstances required to manifest the bug vary considerably, and are sensitive to small changes in the source code and declarations. The bug is most likely to manifest when there are structs containing members that are arrays of structs that have bitfield members, and accesses are made to bitfield members and/or other members that share the same storage unit through the same array element - but some elements are specified with an index that is a compile-time constant and others use an index that is a run-time value. This has some similarity in symptoms to the previously-cited bug, but it is a separate problem.

  This bug is a regression relative to V6.0. It first appeared in the second ECO kit for V6.2 (V6.2-006), and is present through subsequent ECO kits to V6.2 and V6.2A, and in V6.4-005. The workaround is to compile /nooptimize. It is fixed in V6.4A.

  An example (Note: this particular example does not show the bug under the V6.4-005 compiler, but a longer version of this example does show it):

```
#include <stdio.h>
#include <string.h>

typedef struct STR1 {
  char               static_s[28];
  struct {
    unsigned int       ls_z;
    unsigned           ms_z     : 11;
    unsigned           nd_z     :  4;
    unsigned           pad0     :  1;
    unsigned short     pad1;
    unsigned int       pad2;
  }                  bits;
} STRUCTURE1;
```

```
typedef struct STR2 {
  int                 sub_recnum_d;
  char                sub_s[32];
  int                 sim_records_d;
  struct {
    int                 sim_recnum_d;
    STRUCTURE1          s1;
    int                 sim_pending_b;
  }                   ar[4];
} STRUCTURE2;

void dummy_stub(int*  ignored)
{
  return;
}
void main(void)
{
  int                    idx;
  STRUCTURE2             s2;

  memset<(&s2, 0xAA, sizeof(s2));

  idx = 0;
  printf("%X\n", s2.ar[0].s1.bits.ls_z);

  dummy_stub (&idx);

  /* Code for this assignment gets removed! */
  s2.ar[idx].s1.bits.ls_z =  0x12345678;

  s2.ar[idx].s1.bits.nd_z = 0;

  printf("%X\n", s2.ar[idx].s1.bits.ls_z);
}
```

- parameters of type float _Complex

  Compiling a function definition for a function with a parameter of type float _Complex, the V6.4-005 compiler will incorrectly issue a REGCONFLICT error in some common cases (e.g. if the float _Complex parameter is first, and the second parameter has a floating point type).

- incorrect external names under /FLOAT=D_FLOAT

  V6.4 introduced a new table in the compiler for the purpose of reserving and prefixing the names of certain run-time library functions that were added to the C standard in C99. Errors in this table cause the V6.4-005 compiler to generate incorrect external references in the object module for calls to the following math routines when /FLOAT=D_FLOAT is in effect: acosh asinh atan cbrt copysign erf erfc expm1 nextafter lgamma log1p log2 logb rint trunc.

- names not prefixed under /prefix=ansi_c89_entries

Errors in the same table mentioned in the previous item also cause the V6.4-005 compiler not to put the DECC$ prefix on calls to the following four routines when /prefix=ansi_c89_entries is in effect: wcscat wcscpy wcsncat wcsncmp.

- signal.h supports compiling under /pointer_size=long

  In previous kits, the signal.h header file did not accommodate handlers that were declared with 64-bit pointers. This could cause warnings in compilations that used the /pointer_size=long command line qualifier. The header in this kit resolves this problem.

- multiple version support issues

  The V6.4 kit introduced support for installing more than one version of the compiler on the same system. This kit addresses some issues with that support, including consistent upper/lowercase conventions in the scripts to avoid problems on ODS-5 volumes, and providing the CLD file that matches each installed compiler version.

- Stack probing bug

  There was a short time span where the register allocator could incorrectly use R26 for stack probing before the return address had been saved. This bug has been present since early versions of Compaq C, but no C code has been able to trigger it to date. Nevertheless, this window of opportunity for generating the incorrect code needed to be closed. Closing that window of time meant that an additional register had to be made available for stack probing. This means that there is one less register available for a pragma linkage preserved register list.

5.20 Problems fixed in V6.4

- token-pasting hex constants

  There was a problem using the preprocessor's token-pasting operator with operands consisting of a pp-number containing the letter E or e. In some case the operand would be treated as two separate tokens in modes other than VAXC or COMMON. E.g. in the following example, the initializer for "a" was correct, while the initializer for "b" was expanded to "0x0E a" instead of "0x0Ea".

```
#define XXX(xxx) 0x##xxx
int a = XXX(0Aa);
int b = XXX(0Ea);
```

- backslash-newline in COMMON mode

Under /MODE=COMMON/PREPROCESS_ONLY, the compiler would sometimes recognize a backslash-newline sequence outside of the context of a string literal, a character constant, or a preprocessing directive, and produce an output file reflecting logical line-splicing instead of leaving the backslash-newline sequence intact in the output. This problem has been corrected as shown:

```
Input:
    char str1[10] = "A"\
                    "Z";
    char str2[10] = 'A'\
                    'Z';
Incorrect output for /stand=common/preprocess_only:
    char str1[10] = "A"
                    "Z";
    char str2[10] = 'A'                    'Z';
Corrected output for /stand=common/preprocess_only:
    char str1[10] = "A"\
                    "Z";
    char str2[10] = 'A'\
                    'Z';
```

5.21 Problems fixed in V6.2A ECO kit 4

- /OPT problem with va_args

  In circumstances that cannot easily be characterized at the source code level, optimization of the code generated for the va_arg macro (for traversing variable-length argument lists) could produce an incorrect result - in some cases accessing the argument value following the one that should have been accessed. First corrected in V6.2A ECO kit 4, compiler ident V6.2-009.

- /OPT problem with tail recursion

  When the return statement of an old-style function definition contained a recursive call, the compiler would sometimes try to replace the recursive call with a loop construct that did not correctly intialize all the loop control elements. First corrected in V6.2A ECO kit 4, compiler ident V6.2-009.

5.22 Problems fixed in V6.2A

- /OPT bug with loops

  The compiler could incorrectly optimize certain loops in which computation of the trip count depended on loop bounds and/or increments that were run-time expressions that were not made into CSEs. This could manifest either as incorrect code or an internal error. First corrected in V6.2A, compiler ident V6.2-008.

- bltinimplret no longer emitted by default

  The compiler used to emit the message bltinimplret by default when certain runtime library functions that do not return an int were called without including the appropriate header. This message was confusing to many users when it appeared in existing well-debugged applications.

  The compiler has been changed so that bltinimplret is no longer emitted by default, but only when enabled explicitly (or as part of the performance or level5 message groups). First corrected in ECO kit 3, compiler ident V6.2-007.

- Compiler hang with /DEBUG/OPT

  The V6.2 compiler could hang when compiling /DEBUG/OPT when there are many instances of misaligned data. This was also a problem in the V6.0 compiler. First corrected in ECO kit 2, compiler ident V6.2-006.

- /OPT bug with structs

  The V6.2 compiler could incorrectly optimize away certain "store" instructions that performed an assignment to a struct member. The only reliable workaround is to compile /NOOPTIMIZE. First corrected in ECO kit 2, compiler ident V6.2-006.

- __CMP_STORE_QUAD built-in function problem

  Using the __CMP_STORE_QUAD built-in function with a 64-bit value as the third parameter of the built-in will cause the compiler to generate code that truncates the third parameter to 32-bits. First corrected in ECO kit 1, compiler ident V6.2-005.

5.23 Problems fixed in V6.2

- excessive compile time for long initializer lists

  Extremely long initializer lists (~100K initializer elements) caused apparent looping in the compiler, which was actually an n**2 algorithm introduced in V6.0 along with initializers with designators. This has been corrected.

- compiler failure with /SHOW=EXPANSION

  When compiling with /LISTING/SHOW=EXPANSION or -source_listing -show expansion the compiler could accvio if there was a macro which expandanded to a string of 32737 characters or more. The compiler will now truncate the expanded macro text.

5.24 Problems fixed in V6.0

The following problems are also fixed in an ECO update for V5.7, V5.7-006

- In order to address a time-critical CLD, a compiler developer option is temporarily being enabled in the C compiler. This command line option allows the user to request that code generated for assignments to objects that are specified as volatile and smaller than 32 bits not use the load-locked/store-conditional sequences that in general can be required to assure volatile data integrity. The option is needed to assure that code will not hang when accessing certain kinds of memory-mapped I/O devices. The option is specified on the cc command line using the syntax /switch=weak_ volatile. Because this is a compiler-developers option, if a compilation using it fails to produce an object module (e.g. due to an error in the source code), the compiler will issue a spurious warning message suggesting that the name of the option might be misspelled. This option should very seldom be used - it is needed to address certain hardware device-access situations and should not otherwise be used.

- Passing a 64-bit pointer to builtin functions such as __STRCPY resulted in a MAYLOSEDATA message. And in general, there were a number of spurious messages and problems using intrinsic and builtin functions in programs using 64-bit pointers. The V5.7 compiler's newly-added feature to detect intrinsic functions automatically had not been adequately tested with mixed pointer sizes. All of these problems have now been addressed.

- The pow( ) intrinsic function incorrectly produced a zero result when the first argument was negative and it was compiled with /assume=nomath_ errno.

- The preprocessor output of wide-string literals explicitly present in the source failed to include the leading L character, turning them into ordinary string literals when the preprocessed output was compiled.
- Compiling the following code with /noopt triggered an Assertion failure in the compiler.

```
struct x {int y;};
int foo()
{       /* note: expansion of the offsetof macro: */
    if (((unsigned int)(&((struct x *)0)->y)) == 0) {
        return 0;
    }
}
```

- The compiler sometimes generated procedure prologues that violated the OpenVMS calling standard by moving the updated SP register into the FP register before storing the address of the procedure descriptor. This could cause certain debugger operations to report stack corruption, and could cause runtime stack corruption in routines running under threads. E.g. a machine code sequence such as the following:

```
    MOV     SP, FP
    STQ     R27, (SP)
```

- The compiler generated incorrect code when a function pointer was cast to a pointer to a procedure descriptor in order to access members of the procedure descriptor other than the code address. The generated code always fetched the first longword from the procedure descriptor (the code address), regardless of the location of the requested struct member within the procedure descriptor. E.g. in the following example the value proc->pdsc$l_size was incorrectly computed as the code address of hstExecuteThread:

```
#include <pdscdef.h>
unsigned int foo(void)
{
    extern int*     hstExecuteThread();
    unsigned int        offset;
    struct pdscdef      *proc;

    proc = (struct pdscdef*) hstExecuteThread;
    offset = proc->pdsc$l_size - 24;

    return offset;
}
```

- Negative values in a shareable image produced using the OpenVMS message compiler could lose sign extension when cast to type int. An example:

```
$ create tst1.c
#include <stdio.h>
extern int*     TST_K_TYPE_I;
int gf11_msglist_extract_i4( int*       arg_i_value ) {
    int         i_type = (int)&TST_K_TYPE_I;
    *arg_i_value = i_type >> 2;
    return (-1) >> 2;
}
main() {
  int x;
  printf("%d  ",gf11_msglist_extract_i4(&x));
  printf("%d\n",x);
}
$ cc tst1.c
$ create tst.msg
        .FACILITY/PREFIX=TST__ TST,   ^X200

.LITERAL        TST_K_TYPE_I = -1
.LITERAL        TST_K_SIZE_I4 = 8

        .SEVERITY ERROR
NO_I4   <>
ENDMSG  <>

        .END
$ message tst
$ create tst2.opt
SYMBOL_VECTOR=(-
TST_K_TYPE_I = DATA,-
TST_K_SIZE_I4 = DATA,-
TST__NO_I4 = DATA,-
TST__ENDMSG = DATA )
TST
$ link tst2.opt/opt/share
$ create tst1.opt
tst1
tst2/shareable
$ link tst1/opt
$ define/user tst2 sys$disk:[]tst2.exe
$ run tst1
Correct output is:
-1  -1
V5.7-004 produced:
-1  1073741823
```

5.25 Problems fixed in V5.7

- Case EVT102462 is fixed: SCA files for large compilations generated under the /ANALYSIS_DATA qualifier no longer causes SCA-F-NOTERMINATE from the SCA LOAD command when there are more than 65535 SCA events.

- A bug in V5.6 caused informational and warning messages to be suppressed when the questionable constructs appeared within the actual arguments to a builtin or intrinsic function call. This is now fixed.

- DEC C now correctly handles array data members of function return results (for functions returning struct or union values). E.g. the following code no longer generates %CC-E-NEEDPOINTER:

```
typedef struct {
    struct {
        int field4;
        int field5[10];
    } field3;
} struct_t;

struct_t return_struct(void);

void foo (struct_t y) {
    int x, z;
    z = y.field3.field5[0];
       /* generate NEEDPOINTER diagnostic below */
    x = return_struct().field3.field5[0];
}
```

- If /ROUNDING_MODE is specified as other than NEAREST, and /FLOAT=IEEE is not specified, then instead of emitting an error, the compiler now emits an informational and changes the /FLOAT mode to IEEE.

- A day-one problem in the code generated for /IEEE_MODE=UNDERFLOW_TO_ZERO was exposed only under OpenVMS V7.1. Programs compiled with this qualifier behave as expected under earlier versions of the operating system, but receive floating point exceptions under OpenVMS V7.1. The compiler problem is now fixed. Note that this particular command line qualifier only affects the code generated for a C main program. In particular, this option causes the generated main program startup function, __main, to call SYS$IEEE_SET_FP_CONTROL. Earlier compilers passed a "set_mask" value of 0x4010 as the second parameter (R17 loaded with the address of this value). The correct value should always have been 0x4000, but the extraneous bit did not cause a runtime problem under pre-V7.1 versions of OpenVMS.

- Valid C code containing a file-scope const declaration initialized with an address constant, and with the file-scope declaration referenced by an extern declaration within a block, could cause a GEM-F-ASSERTION internal compiler error and traceback containing routine GEM_ST_UPDATE_IV_LISTS in the call chain. E.g., the following code triggers this problem:

```
static char * const x = "hello";
void foo (void)
{
      extern char * const x;
}
```

- In previous compilers, enumeration constants with values beyond the range of type int were silently truncated to int. These now cause a warning to be issued.

- Certain invalid preprocessing directives could cause an infinite loop in the compiler. E.g. #module 0badident, #include directives with mismatched delimiters.

- The REMQxxQ builtin functions have been fixed to return 64-bit pointers.

- Certain uses of the token-pasting operator to form valid hexadecimal constants were incorrectly treated as errors. E.g., the following produced an error but is now handled correctly:

```
#define SVAL(a,b) 0x ## a ## b
int i = SVAL(23a2,0bae);
```

- The compiler no longer issues a pointer mismatch warning when a pointer to an array of unqualified element type is passed to a function declared with a parameter declared as a multiply-dimensioned array type with a type qualifier. The warning was erroneous, as the types are compatible under ANSI C. E.g., the following no longer produces a warning:

```
extern void func1(const double dbl[][3]);
void func2(void) {
  double  adPnts[8][3];

  func1 (adPnts);
}
```

- The compiler now warns when two different names with external linkage in the same compilation are mapped to the same name in the object module (e.g. due to truncation or forcing to monocase).

- The size of conditional operator (i.e. "?:") expressions was not being computed correctly when the values were character string constants. They were being treated as arrays instead of pointers. E.g. the following produced an error instead of evaluating to the size of a pointer-to-char:

```
int is_what;
int b;
void f(void)
{
   b = sizeof(is_what ? "one" : "testing");
}
```

- Left shift of signed ints did not perform final sign extension of the 32-bit result.

5.26 Problems fixed in V5.6

- A compilation containing an ANSI function prototype for a variable argument list using ellipsis syntax, with a mismatched K&R style definition for that function, provoked the warning "%CC-W-FUNCREDECL, In this declaration, function types differ because one has no argument information and the other has ellipsis". But the compiler then access violated after issuing the warning.

- Compiler crash and/or memory exhausted system hang using /debug qualifier with #include <exc_handling.h>, or processing any other struct or union type containing a member that is a pointer to a qualified version of the same struct or union type.

- When compiling a comma-separated list of source files with /list, the listing section containing the values of predefined macros would not be present in the listing files for the second and subsequent compilations. In some cases, the compiler would crash.

- Compiler would sometimes crash after emitting %CC-W-NOLINKAGE.

- Under /STAND=VAXC, inform/warn about use of "signed" keyword, which was not supported by VAX C.

- Function parameters of type "pointer-to-qualified-type" in some cases could not be examined in the debugger, provoking the messagee %DEBUG-W-UNIMPLENT from the debugger.

- The /VERSION qualifier previously required that a source file name be provided, and the compiler would then create a 0-length .OBJ file corresponding to that source file, even though the action of the /VERSION qualifier does not involve any source or object files. Now the compiler version is obtained by running just "cc/version" without specifying a source file name or any other qualifiers, and no object file is created.

- Fixes to VAXC and MS compatibility modes.
- Fixes to /ANA processing.

5.27 Problems fixed in V5.5

- In V5.3-006, if a source file to be compiled resided on a remote filesystem that was mounted by NFS client services (either from Digital's UCX product or from other products providing NFS client services on OpenVMS Alpha), or in some cases if the file was remote-mounted using DFS, it was possible to receive the following fatal error message from the compiler:

  ```
  %CC-F-OPENIN, error opening 'input-file' as input
  -RMS-F-COD, invalid or unsupported type field in XAB at 'addr'
  ```

  The only workaround was to move the source(s) to a local disk.

- Under /optimize=level=5, the V5.3-006 compiler could encounter an assertion failure attempting to compile certain kinds of loops with conditional exits. An example is:

  ```
  $ type foo.c
  main()
  {
      int len;
      char *cp;

      while ((len > 1) && (*cp == '0'))
          ++cp, --len;
  }
  $ cc/optimize=level=5 foo.c
  Assertion failure:  Compiler internal error - please submit...
  %GEM-F-ASSERTION, Compiler internal error - please submit...
  %GEM-F-ASSERTION, Compiler internal error - please submit...
  %TRACE-F-TRACEBACK, symbolic stack dump follows
    image         module    routine              line
   DECC$COMPILER  GEM_DB  GEM__DB_ASSERT_END      720
   DECC$COMPILER  GEM_DB  GEM_DB_ABORT            587
   DECC$COMPILER  GEM_DB  GEM_DB_ABORT_FAST       610
   DECC$COMPILER  GEM_LU  VECTORIZE_LOOP         7606
   DECC$COMPILER  GEM_LU  GEM_LU_MAIN            2126
   DECC$COMPILER  GEM_CO  GEM_CO_COMPILE_ROUTINE 2078
   DECC$COMPILER  GEM_CO  GEM_CO_COMPILE_MODULE  1432
   DECC$COMPILER                                   0
   DECC$COMPILER                                   0
   DECC$COMPILER  GEM_CP_VMS  GEM_CP_MAIN        2509
  ```

- The globalvalue extern_model could produce a compiler traceback when an external variable of a struct or other type not supported by globalvalue was encountered. The compiler now produces an appropriate warning message and uses the strict_refdef model for such declarations.

- SCA typing information for incomplete arrays declared with a typedef declaration were sometimes incorrect and complex declarations caused compile time access violations. Known instances of these problems have been corrected.

- Placement of certain kinds of pragmas within declarations could cause compiler failures or erroneous .ANA files to be produced uner /ANALYSIS_DATA.

- Forward declarations involving structure members no longer cause invalid .ANA files to be produced when compiling with /ANALYSIS_DATA

- Invalid .ANA files are no longer produced when compiling with /ANALYSIS_DATA for source files with code beyond column 256.

- Better SCA reference information is output for struct declarations and references.

- Functions with an excessive number of parameters (around 100) will no longer cause the compiler to crash when outputting SCA information. Instead, a warning message is emitted and further parameters will not generate SCA information.

- Debugger support for pointers to functions has been improved.

- Improved initialization for redeclared variables. Previously the compiler would not initialize a in the following code

```
int a;
static int a = 1;
```

- Delete the object file for certain backend-detected errors.

- Issue a warning for a cast expression used as an lvalue in strict ANSI mode.

- Warning messages issued by the compiler backend are now subject to message control qualifiers and pragmas. Previously, message idents such as ASMRAWREG (issued by the backend for inline assembly code that uses a hardware register that was not initialized by anything visible to the compiler) were not known to the message control facility. An attempt to suppress the ident would produce the message '%CC-W-UNKMSGID, Unknown message id or group "ASMRAWREG" is ignored.'

- A #undef or #define preprocessing directive appearing within the arguments to a function-like macro invocation could cause the compiler to crash (e.g. if it undefined or redefined the function-like macro being invoked). This now produces a warning message.

- Source files containing very large numbers of #line preprocessing directives could cause a seemingly infinite loop in the compiler due to an n**2 algorithm. The algorithm is now linear.

- A number of changes have been made to improve compatibility with VAX C, Microsoft C, and UNIX compilers. New messages have been introduced and old ones altered to mimic more accurately the behavior of these other compilers when using the /STANDARD switch. Some of the these compatibility changes are:

  - Do not accept an array of incomplete types.

  - Do not allow constructs like "(z ? x : y).a = 1;"

  - Issue warning for pointer to int assignment.

  - Do not allow pointer types as case constants.

- Disable processing of type qualifiers after a comma.

- Handle redeclaration of items with different storage classes.

- Implement VAX C style unnamed struct/union members in VAXC mode, where the struct/union type of the unnamed member is treated as if it occurred outside of any containing struct/union. In MS mode, the same construct is treated like a C++ unnamed member of union type in C++: the members of the nested struct/union are promoted to the containing struct/union, similar to the behavior of VAXC variant_struct and variant_union constructs.

- Allow "x->y" under /STAND=VAXC where x is an integer constant or variable.

- Under /STAND=VAXC, "sizeof(&array)" now returns "sizeof(array)" (i.e. the size of the array object) instead of the size of a pointer (which is what ANSI C requires in this case).

- Emit meaningful error message for a[b] when both a and b are pointers.

- Emit informational in VAXC mode for declarations with an explicit "extern" storage class which are also initialized.

- Issue a warning message for cases like a->b where a is a pointer to an incomplete struct/union.

- In MS mode, allow an incomplete array type only as the last member in a struct or union. Previously, other incomplete types were accepted, and they were not restricted to the last member.

- Allow globalvalues to be used in initializers Alpha as they are on VAX.

- Allow ellipsis in old-style function definitions under /STAND=VAXC.

- Emit language extension messages for __restrict and __unaligned in /STAND=PORT

- Improve handling of VAX C storage class modifiers (noshare, readonly, _align) to more closely match the VAX C compiler's behavior.

- In modes other than VAXC, emit additional diagnostics when a function is redeclared with a different type.

5.28 Problems fixed in V5.3

- V5.2 introduced a bug which caused logical name translation to be applied to the names of text library modules.

- V5.2 changed the behavior of /NESTED_INCLUDE=PRIMARY_FILE on Alpha to match that of DEC C on VAX, which was unintentionally incompatible with the original VAX C behavior. In V5.3, both VAX and Alpha interpret this qualifier to produce behavior that is compatible with VAX C.

- Use of the /CHECK=POINTER_SIZE qualifier could cause a compiler failure if a cast of a constant to a long pointer type was assigned to a short pointer variable.

- Calling LIB$SIGNAL from within a signal handler failed to produce a traceback.

- Problems involving the use of variable argument lists (<stdarg.h> or <varargs.h>) were fixed. These involved old-style function parameters of types requiring default argument promotion (now get an E-level diagnostic), and the use of a va_list variable that was not a simple locally-declared variable (now handled correctly).

- Diagnostic messages containing character constants using a hex escape sequence were being printed with the "x" character omitted.

- The preprocessor now allows the "defined" operator to be produced as a result of macro expansion. ANSI C does not specify whether or not this should be allowed. Under /standard=ansi89, a warning message is produced when the preprocessor encounters such a construct. In relaxed ANSI an informational is issued instead.

- Under /STAND=VAXC, the type of the sizeof operator is now signed int (matching the behavior of VAX C) rather than unsigned int (which is required by ANSI C).

- A block-scope extern declaration with incomplete type, with linkage to a file-scope declaration with complete type, "hid" the completed type information from being available within the scope of that block. This could produce an E-level diagnostic for a correct program such as the following:

```
int i[] = {1, 2, 3};
unsigned int f()
{
    extern int i[];
    return sizeof(i);
}
```

- An initialized globalvalue may now be used as a compile-time constant, just as an enumeration value can be.

- For a compilation with multiple source files in a comma-list, the module names in the .ana files produced under the /ana qualifier were incorrect for all but the first compilation.

- Under /standard=vaxc, match the VAX C behavior and accept a sequence of conflicting type specifiers with a warning and use the last one, instead of issuing an E-level diagnostic.

- SPR HPAQA9E70: Statically initialized data declared with the "const" qualifier was being placed in the $READONLY$ psect, even if it contained address constants requiring fixups. This rendered the entire psect non-sharable. Constant addresses requiring fixups are now placed in a different psect ($READONLY_ADDR$), allowing the $READONLY$ psect to remain sharable.

- SPR HPAQ628CF: Static initialization using a globalvalue as an initializer produced incorrect initialization if the type of object being initialized was not int (e.g. if it was a char or short type).

- The compiler now allows an actual argument to a function call to pass the address of a globalvalue; formerly this produced an E-level diagnostic even in VAX C mode. Also changed the mechanism for passing the address of a constant such that if the callee modifies the value it will not affect subsequent calls. This makes the behavior compatible with VAX C.

- The compiler failed to diagnose an ANSI constraint violation for an old-style function definition with a parameter identifier list, where one of the parameter names matches a visible typedef name.

- The compiler failed to diagnose use of the name of a variant_struct or variant_union member when constructing the name of a contained member. This is a VAX C feature, and VAX C compiler produces an E-level diagnostic if the name is used in this way.

- The compiler was incorrectly producing an E-level "invalid declarator" diagnostic when a label was defined with the same name as a typedef. Labels have a separate namespace and should not conflict with other declarations.

- The compiler would fail if it attempted to output a diagnostic message referring to an unnamed bit field in a struct or union declaration.

- Under /stand=common, hexadecimal escape sequences (\xnnnn) were being recognized within string literals and character constants. This feature was part of VAX C and ANSI C, but it was not present in "pcc" compilers, and recognizing it under /stand=common produced results that differed from pcc compilers.

- SPR HPXQ11CEF: Compiler failure when globaldef storage class is applied to a function. The compiler now gives an appropriate E-level diagnostic when globaldef, globalref, or globalvalue storage class is applied to a function.

- Compiling a function definition that attempts to use a typedef for its function header could either cause a compiler failure or produce an inappropriate diagnostic, e.g.:

```
typedef void VFV();
VFV f;
VFV f {}
```

  This now produces the correct diagnostic: %CC-E-TYPEDEFFUNC, In this function definition, "f" acquires its type from a typedef.

- If a globaldef declaration specified a psect name, the psect specification would be ignored if the same variable was previously declared in a globalref declaration.

- A block-scope declaration with the extern storage class would cause a compiler failure if the identifier being declared matched a visible typedef name declared at file scope.

- Compiler failure when attempting to output a fatal error message. The following example produced a compiler failure trying to output the fatal error message for include file not found. The root cause involved the specific placement of the pragmas in relation to the declaration:

```
struct my_s {
#pragma message save
#pragma message disable portable
   int one;
   int two;

#pragma message restore
};
#include <i_dont_exist.h>
```

- When the VAX C "main_program" keyword was used to identify a main program procedure with a name other than "main", the compiler still generated a global procedure named "__MAIN". This made it difficult to put more than one such object module into a single object library. The compiler now makes the "__MAIN" symbol it generates in this case a local symbol.

- Under /stand=vaxc, an incompatibility between a function prototype and the corresponding function definition produced only a W-level diagnostic, but calls to such a function were silently ignored by the compiler, causing incorrect behavior at runtime. The diagnostic remains at W-level, but now the generated code is correct.

- Compiler failure when processing a long comma-list of source files (more than 20 sources).

- SPR UVO104030: Compiler failure under /preprocess_only when processing a pragma with incorrect syntax.

5.29 Problems fixed in V5.2

- DEC C will now Skip parameter checks if we are evaluating an operand to sizeof in common mode as in the example below:

```
j(int a)
{
p(sizeof(j()));
}
```

The severity of the messages concerning too many and too few parameters is now a warning in VAXC mode (/STAND=VAXC) rather than an error. This behavior is compatible with VAX C.

- Error messages have been relaxed for conflicting extern defs as follows:

  For /STAND=COMMON there is no change in behavior.
  All extern declarations are promoted to file scope. When the compiler encounters a conflicting declaration, it will issue a error as it has always done.

  For /STAND=VAXC the severity of the error message has been reduced to a warning.
  VAX C issues a warning whenever it finds a conflicting extern declaration. It does not matter if the declarations are in the same name scopes or not. In addition to issuing a warning, VAX C replaces the prior declaration with the new declaration from the point of the new declaration onward. DEC C now matches this behavior in VAXC mode.

  For /STAND=ANSI, /STAND=RELAXED
  Errors will be generated, (as they always were) if there is a conflict between declarations that are in the same or inner name scopes.

  DEC C will issue a warning if there is a conflict between names that are in disjoint scopes. This will no longer be an E-level message. The standard says that such a case is in error, but that a diagnostic does not have to be issued. We felt that it was better to issue a diagnostic than to silently accept the program.

  For example, in the program shown below:

  /STANDARD=COMMON will result in no diagnostic messages.
  /STANDARD=VAXC will result in a warning message about incompatibile declarations for the second and third declaration of init_color
  /STANDARD=RELAXED, /STANDARD=ANSI will result in an informational diagnostic on the second declaration of init_color because it is implicitly declared and will result in a warning on the third declaration because it is incompatible with the declaration on line 3, even though it is in a different scope.

```
main()
{
  extern void init_color();
}
```

```
Raise_Maps()
{
 init_color();
}

Title_Page()
{
  extern void init_color();
}
```

- A problem has been corrected which could cause an ACCVIO at compile time when compiling with the qualifier /ANALYSIS_DATA.

- In all modes, functions declared at block scope will now have their storage class set to extern. A warning is issued if the storage class is register. A warning is also issued if the storage class is auto (except in VAXC mode). If the storage is static, in common and vaxc mode then no warning is issued for the declaration. But a warning will issued later if the function is referrenced and not defined anywhere in the module.

- SPR EVT101335
  Whenever a call causes more than 255 items (longwords on VAX/quadwords on Alpha) to be used in constructing the arg list a warning will be issued. On Both VAX and Alpha an informational will be output warning that the argument list length exceeds maximum specified by the calling standard.

```
/* This program used to ACCVIO on VAX/VMS, now
   it gets a compile-time diagnostic */
struct {
        int i;
        char longer_than_1020[1021];
       } public_domain_sloppy_programmer;
void nothing();
main ()
{
 nothing (public_domain_sloppy_programmer);
}
```

- SPR UVO102632 Formerly the compiler sometimes failed to issue a diagnostic when an assignment was made to a constant array as in the example below:

```
void fred (void)
{
   typedef int A[2][3];
   const A a = {{4, 5, 6}, {7, 8, 9}};

   a[0][0] = 42;
}
```

- The compiler will now accept unnamed structures as members of a struct in VAXC mode.

- The compiler will now issue a warning instead of an error when pointers and ints are compared in common mode.

- The compiler will now issue a warning when preprocessing directives are used in the argument list for a macro

- The compiler will now Allow more than just integers in switch and case expressions in vaxc and common modes. We now issue a new warning when a float or pointer is used in a switch or case expression.

- A problem has been corrected involving the #dictionary directive when it was nested within a structure declaration. The compiler now correctly generates a member name for the extracted CDD record nested within a struct, not a tag name.

- The Alpha VMS V5.0 Help (and the User's Guide) are incorrect in their description of /IEEE_MODE=FAST.

```
The V5.0 documentation reads:

  /IEEE_MODE
        /IEEE_MODE=option
        /IEEE_MODE=FAST (D)

     Selects the IEEE floating-point mode to be used if
     /FLOAT=IEEE_FLOAT is specified.

  Options:
```

```
      FAST  During program execution, no exceptions are
            raised and only finite values (no infinities,
            NaNs, or denorms) are created.  Your program
            must examine errno for any error indication.

It should, however, read:

  Options:

      FAST  During program execution, only finite values
            (no infinities, NaNs, or denorms) are created.
            Exceptional conditions, such as floating point
            overflow and divide by zero, are fatal.
```

- The compiler will no longer remove code that accesses a location under the "volatile" qualifier, even if the value is unused. E.g. the statement "x;" will now generate code to fetch x if x was declared volatile.
- The macro definitions within a /define=(name[=value],...) list are now processed left to right. Thus /DEFINE=(A=1,A=2) now leaves A defined as 2 instead of 1.
- Some cases of right-shifting the result of an int left-shift operation could produce incorrect code, e.g. ((i32 << 24) >> 16) and ((i64 << 32) >> 48) produced incorrect results.
- Problems with the /NESTED= qualifier have been fixed.
- The severity of the NONMULTALIGN message has been reduced to a warning.
-
- Several problems in computing the value of an integer constant constructed through token-pasting in the preprocessor have been fixed. E.g. the following code formerly resulted in an incorrect message "%CC-W-INVALTOKEN, Invalid token discarded".

```
  #define concat(a,b) a ## b
  return concat(0x0,1AL) ;
```

  It now is handled correctly.

5.30 Problems fixed in V5.0

- DEC C used to issue messages for lexical "errors" appearing within the bodies of macro definitions for macros that were never used. In some cases these should not have been issued according to ANSI C (e.g. warnings for octal constant containing digits 8 or 9), and generally such potential problems do not require an ANSI diagnostic. Common practice is to defer such reports until a macro is used, which is what DEC C now does.

- The result of compiling the output of the /PREPROCESS_ ONLY qualifier was not always the same as the result of compiling the original program. Consider the program below.

```
#define A(x) -x
main() {
    int i = 1;
    printf("%d\n", -A(i));
}
```

  The output from /PREPROCESS_ONLY used to place the '-' of the body of macro A next to the '-' before the macro invocation, producing:

```
printf("%d\n", --i);
```

  Now the output has a space to separate the two '-' characters to prevent this accidental token-pasting unless the compiler is in common or vaxc modes, where this kind of token-pasting is done when compiling the original source directly.

- When doing macro substitution inside string constants in VAX C mode, DEC C did not always substitute when VAX C would. Given the macros:

```
#define str1(arg) "arg"
#define str2(arg) "arg-ber"
#define str3(arg) "go to the arg"
```

  Formerly DEC C did not do a replacement in str2 where VAX C does. Now this replacement is done.

- ICA-48945: mixing of old-style and new style function prototypes:

The compiler now allows mixing of new-style function prototypes and old style function definitions where the prototype parameters are not fully promoted integer types (according to default argument promotion rules). With this modification, all integer type combinations are allowed (including signed/unsigned mixing). A warning is issued where we were issuing an E level error in the past (no message is issued if in VAXC mode and the integer types in the old style parameter definition match those in the prototype, as in the code fragment provided in the SPR).

```
void f (char);
void f (p1)
char p1;
{}

$ cc/stand=vaxc foo.c
$ cc foo.c
char p1;
.....^
%CC-W-PROMOTMATCHW, In the definition of the function
"f", the promoted type of p1 is incompatible with
the type of the corresponding parameter in a prior
declaration.
at line number 3 in file DISK:[dir]FOO.C;1
```

In addition, the following will now correctly compile:

```
extern in (*f1())(int (*f2)());
int (*f1(f2))()
int (*f2)();
{ return 0;}
```

- HPXQ7084C, CDD datatype text size 1, can now be converted be converted to an array of char or to a char using the new #dictionary keywords, text1_ to_array, text1_to_char.

- HPXQ74784 DEC C will no longer ACCVIO when an include file can not be found.

- UVO101931 The will now generate correct code for the following program /NOOPT:

```
#include <stdio.h>
typedef unsigned __int32 uns32 ;
main ()
{
  uns32 y1, y2 ;
  char str1[255] ;
  y1 = 23 ;
  y2 = 1 ;
  str1[24 - y1 + y2] = '\0' ;  // <- Used to Crash here
}
```

- DEC C no longer gives an erroneous INCOMPNOLINK diagnostic for the following code:

```
static int a[] = {0, -1, 2, -3, 4, -5, 6, -7, 8, -9};
int main() {
    extern int a[];
}
```

  This used to occur only if the extern declaration for a appeared inside a block.

- The compiler will now correctly give a diagnostic if C++ style comments are used with /STANDARD=PORT or /STANDARD=ANSI89.

- Full support for 64bit integer constants is now available, e.g.

```
/* Value of constant cannot be represented as
 * unsigned long, so it is unsigned __int64.
 */
unsigned __int64 foo = 0x7fffffffffffffffu;
```

5.31 Problems fixed in V4.1

- HPXQ97CFE: The maximum length of a source line has now been relaxed to the maximum allowed by RMS.

- Fixed a problem which caused V4.0 to miss breakpoints on an if statement when the OpenVMS/Alpha V6.1 was being used.

- Fixed a DST nesting error when a sequence of two or more typedefs occurred in a recursive type declaration.

- HPAQ92A0F: Formally, the debugger referenced bitfields as longwords, which caused problems when depositing values into those bitfields. This problem has been fixed.

- An ECO kit is available to fix problems with printf and fcvt IEEE floating point.

- Fixed a compile-time ACCVIO when compiling volatile structs with /NOMEMBER_ALIGN

- builtins.h

  The storage for __xxxQUE_MAP_ALPHA_TO_VAX and __REMQxI_MAP_ALPHA_TO_VAX has been moved from builtins.h to the DEC C RTL in order to reduce the number of bytes of storage required by the header file.

- curses.h

  Changes have been made to improve the functionality of the default curses package.

- float.h

  On OpenVMS Alpha the D_FLOAT definitions of DBL_MAX and LDBL_MAX were corrected.

- fp.h

  The new header file <fp.h> implements some of the features defined by the Numerical C Extensions Group of the ANSI X3J11 committee. Applications making extensive use of floating point functions may find this useful.

  Some of the double precision DEC C RTL functions return the value HUGE_VAL (defined in either math.h or <fp.h>) if the result is out of range. The float versions of those functions return the value HUGE_VALF (defined only in <fp.h>) for the same conditions. The long double versions return the value HUGE_VALL (also defined in <fp.h>). Note that for programs compiled to enable IEEE infinity and NaN values, the values HUGE_VAL, HUGE_VALF and HUGE_VALL are expressions, not compile-time constants. Initializations such as the following cause a compile-time error:

```
$ CREATE IEEE_INFINITY.C
#include <fp.h>
<P>
double my_huge_val = HUGE_VAL
^Z
$ CC /FLOAT=IEEE/IEEE=DENORM IEEE_INFINITY

double my_huge_val = HUGE_VAL;
.....................^
%CC-E-NEEDCONSTEXPR, In the initializer for my_huge_val,
"decc$gt_dbl_infinity" is not constant, but occurs in
a context that requires a constant expression.
at line number 3 in file WKD$:[LCRTL]IEEE_INFINITY.C;1
$
```

When using both math.h and <fp.h> be aware that math.h defines a function isnan( ) and <fp.h> defines a macro by the same name. Whichever header is included first in the application will resolve a reference to isnan( ). To force references to use the function instead of the macro, enclose the name of the function in parentheses, e.g. (isnan)(arg) instead of isnan(arg).

- math.h

  The D_FLOAT definition of HUGE_VAL was corrected on both OpenVMS VAX and OpenVMS Alpha.

- ints.h

  Definitions for (u)int16 and (u)int32 were added for use by DEC C++ programs on OpenVMS VAX. This will allow DEC C programs using (u)int16 or (u)int32 to be portable to DEC C++ on OpenVMS VAX.

- perror.h

  Definitions for decc$ga_sys_errlist and decc$gl_sys_nerr were added for use by DEC C and DEC C++ programs. These are provided for compatibility with VAX C programs that made use of sys_errlist and sys_nerr.

- setjmp.h

  A prototype for decc$setjmp was added.

- stat.h

  Macros defining constants for group and other protection masks were added to match the ones for 'owner'.

- stdarg.h

  A definition for va_count was added.

- stdio.h

Modifications were made to the definitions of clearerr, feof, ferror such that proper usage of these macros does not give warnings when compiling /WARNING=ENABLE=CHECK.

- unixlib.h

  Prototypes were provided for the following routines on OpenVMS VAX: decc$to_vms, decc$from_vms, decc$match_wild, decc$fix_time, decc$translate_vms.

5.32 Problems fixed since V1.3A

- Fixed several problems with SCA support.

- Fixed a problem in include file lookup behavior whereby the compiler would search the wrong directory under /NESTED_INCLUDE=INCLUDE.

- Fixed compiler assertions generated by certain references to the PAL_ REMQxxx functions.

- Fixed a problem in the static initialization of bitfield structs of greater than 32 bits.

- Fixed a compile-time ACCVIO when compiling a structure containing an array of incomplete structures.

- The compiler now correctly detects attempts to pass a pointer to an array of pointers to const chars to a function expecting a pointer to a pointer to an array of pointers to non-const chars:

```
static void f(char *argv[]) {}
static void g(const char *argv[])
{
  f(argv);
}
> cc foo.c
 f(argv);
..^
%CC-W-PTRMISMATCH, In this statement, the referenced
type of the pointer value "argv" is "Pointer to const
char", which is not compatible with "Pointer to char".
```

- A problem in which carriage returns immediately following comments caused the compiler to crash, has been fixed.

- All known restrictions regarding initialization of variant_unions have been lifted.

- A problem in which _align( ) did not result in correct alignment has been fixed.

- In any of the /STANDARD={VAXC,RELAXED_ANSI89,COMMON} modes, a redeclaration of a function with an empty argument list is now compatible with previous declarations containing ellipses. The following declarations are now compatible:

```
#include <stdio.h>
fopen();
```

- Problems with implicit conversions from unsigned base integer types to their same sized signed counterparts have been fixed.

- A problem with bad code generated for uint64 switch statement values has been fixed. Using int64 switch statement values is still problematic. See the restrictions section below.

- An optimizer error involving left shifts in doubly nested loops has been fixed.

- An internal compiler assertion involving conversions from int to address has been fixed.

- A problem in which #pragma [no]standard could at times cause subsequent code to be ignored has been fixed.

- A problem with /NESTED_INCLUDE causing an infinite loop has been fixed.

- A problem involving incorrect assignment of psect RD/WRT attributes has been fixed.

- A problem with aliasing of extern variables has been fixed.

- An /OPT=LEVEL=5 problem involving comma-listed post-increment operators as the increment expression of a for loop has been fixed.

6 Support for STDARG.H and VARARGS.H

The standard header files STDARG.H and VARARGS.H which are provided with the DEC C for OpenVMS Alpha have special builtin support to walk the argument list. To walk the argument list of a routine, you must use the standard macros in one of the above header files.

Programs that take the address of a parameter and, through pointer arithmetic, independently walk the argument list to obtain the value of other parameters, make the assumption that all arguments reside on the stack and that arguments appear in increasing order. These assumptions are not valid when using the DEC C for OpenVMS Alpha. To ensure correct results the macros provided in the header files must be used.

7 Debugger support

OPTIMIZATION LEVEL: For satisfactory use during debugging, modules should be compiled using /NOOPTIMIZE. Compilation with normal (full) optimization will have these noticeable effects:

1. Stepping by line will generally seem to bounce forward and back, forward and back, etc, due to the effects of code scheduling. The general drift will definitely be forward, but initial experience indicates that the effect will be very close to stepping by instruction!
2. Variables that are "split" so that they are allocated in more than one location during different parts of their life times are not described at all.

FORMAL PARAMETERS: Formal parameters that are passed in registers, while not handled quite like normal split variables, do share many of the same problems as split variables. Even with /NOOPTIMIZE, such a parameter will often be immediately copied to a "permanent home" (either on the stack or in some other register) during the routine prolog. The DST description of such parameters encodes this permanent home location and NOT the physical register in which the parameter is passed. The end-of-prolog location is recorded in the DSTs and will be used as the preferred breakpoint location when a breakpoint is set in the context of an appropriately set module (so that DST information is available to the debugger).

PLUS_LIST_OPTIMIZE: /PLUS_LIST_OPTIMIZE executables are fully debuggable. If two or more variables in different files share the same name and are statically declared, however, the debugger cannot discriminate between them. Users should in this case do an "EVAL/ADDR FOO" for variable FOO, and subsequently reference the address of the variable.

8 64 bit support

The compiler has builtin types for signed and unsigned 16, 32, and 64 bit integer data types. They are intended for applications that must have integer data types of a specific size across platforms that can provide the data type. Note that some data types, for example 64 bit integer types, are available on OpenVMS Alpha but not available on OpenVMS VAX.

The header file ints.h contains typedefs for the integer data types. For sake of portability, we encourage the use of ints.h typedefs and highly discourage the use of the builtin data types directly.

The content of ints.h is included below:

```
/*
 *  <ints.h> - Definitions for platform specific integer types
 *
 *
 *  Copyright (c) 1993 by Digital Equipment Corporation.
 *  All rights reserved.
 *
 *  DEC C for OpenVMS VAX and OpenVMS Alpha
 *  DEC C++ for OpenVMS VAX and OpenVMS Alpha
 */

#if defined(__DECC) || defined(__DECCXX)
/* This file uses non-ANSI-Standard features */
#pragma __nostandard
#else
#pragma nostandard
#endif

#ifdef __cplusplus
    extern "C" {
#endif

typedef signed char             int8;
typedef unsigned char           uint8;

#if defined(__DECC) || (defined(__DECCXX) && defined(__ALPHA))
typedef signed __int16          int16;
typedef unsigned __int16        uint16;
typedef signed __int32          int32;
typedef unsigned __int32        uint32;

#if defined(__ALPHA)
    typedef signed __int64      int64;
    typedef unsigned __int64    uint64;
#endif
#endif

#ifdef __cplusplus
    }
#endif

#if defined(__DECC) || defined(__DECCXX)
/* This file uses non-ANSI-Standard features */
#pragma __standard
#else
#pragma standard
#endif
```

64bit integer constants are now supported.

9 Restrictions and known bugs

This is a list of known compiler restrictions and bugs.

- Compiler might emit erroneous BADANSIALIASn message

  In some situations, the compiler's loop unrolling optimization might generate memory accesses in the code stream that never actually execute at run-time, but which 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's 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).

- The complex data types are not available when using the /float=d_float command line option. This is a permanent restriction.

- On versions of OpenVMS prior to V7.3, the long double complex type cannot be used because the run-time support for it is not present.

- In V6.4 of Compaq C, the C99 functions cabs, cabsf, and cabsl cannot be used. This is a temporary restriction.

- The __typeof__ operator, useful for gcc compatibility, causes a compiler failure if compiled with the /ANALYSIS_DATA qualifier.

- Source code that uses CDD (either #pragma dictionary or the equivalent #dictionary directive) and is compiled with the /ANALYSIS_DATA qualifier may produce incorrect information for SCA.

- Function definitions compiled with #pragma use_linkage that used to compile with the V6.2 compiler may encounter the following error when compiled by a V6.4 or later compiler: "%CC-E-NOREGAVAIL, Unable to satisfy program register allocation requirements". This is caused by a linkage that attempts to preserve too many registers, leaving an insufficient number of scratch registers for the compiler to generate the function prologue. The solution is to change the linkage to preserve fewer registers. If the linkage worked with an earlier compiler, it should only be necessary to remove one preserved register to meet the requirements of V6.4 and later compilers. This is a permanent restriction.

- The changed syntax for the /VERSION qualifier, which no longer requires a source file to be specified, now issues a warning if other qualifiers are used. To workaround the previous problem of the /version qualifier creating a 0-length object file, some users may have specified cc/noobject/version, or cc/version/noobject. The former syntax now provokes "%DCL-I-IGNQUAL, qualifiers appearing before this item were ignored" followed by the normal /version output. The latter syntax, where other qualifiers appear after /version, now produces "%DCL-W-IVQUAL, unrecognized qualifier - check validity, spelling, and placement", and the /version output is not produced. This will likely be a permanent aspect of the /version qualifier.

- Under /NESTED_INCLUDE_DIRECTORY=PRIMARY_FILE, the sticky-default processing for included files ignores filespec information from top-level source files that are empty.
- There is a permanent restriction that header file searches involving DECnet file specifications may not correctly interpret an access string with password. This is because an expanded filespec is actually used to open the files, and expanded filespecs are stored without the password information for security reasons. Thus an attempt to open the file using the expanded name will generally fail. The DECnet access should be made without the need for a password (e.g. through the default account or through a proxy).
- Currently, the compiler allows you to take the address of any predefined built-in function. This should be illegal and a compile time error should be generated.
- Function argument lists longer than 255 longwords are not consistently diagnosed.
- The globaldef keyword when used with an enum type definition does not make all of the enumeration constants globalvalues in VAX C mode. If you wish to make the values of enum constants available to another module, put the enum type definition in a header file and include it.
- When data is placed into the LIB$INITIALIZE psect using the extern_ model psect naming controls, by default DEC C creates the psect with quadword alignment like any other data psect, rather than the longword alignment required for this special VMS psect. This is easily addressed by specifying the alignment explicitly on the extern_model pragma. This is a permanent restriction.
- The following VAX C pragmas are not supported. This is a permanent restriction.

  - #pragma ignore_dependency
  - #pragma safe_call
  - #pragma sequential_loop
- There is no support for the PDF (/DESIGN) and DWCI tools.
- The functions LIB$ESTABLISH, LIB$REVERT, and VAXC$ESTABLISH are recognized by the compiler as builtin functions. There is currently a restriction that the builtins do not return the previous condition handler as their function result.

- If your code includes assert.h multiple times and uses the text library form of inclusion, `#include assert`, the first include will work correctly. But, the second include causes the compiler to issue an error message about an invalid include file name. This is because assert have been defined as a macro within assert.h, so the compiler is looking at the macro expanded file name, which does not exist. The assert macro may be used elsewhere without any problems.

  Digital recommends that you avoid the text library form of inclusion for assert.h. Use #include <assert.h> instead.

- DEC C may give redundant diagnostics for FLOATOVERFL, FUNCREDECL, and PROTOSCOPE3. Two diagnostics will point to the same location. Fixing the problem makes both diagnostics go away.
- The diagnostic for array declarations with more than INT_MAX elements is misleading. It should state that there are too many elements in the array.