VSI BLISS V1.14-144 for OpenVMS x86-64

Release Notes

Publication Date: November 2024

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

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

VSI BLISS V1.14-144 for OpenVMS x86-64 Release Notes

Legal Notice

Table of Contents

1. OpenVMS X86-64 BLISS Bugfixes, Features, and Differences 5
1.1. The Family of BLISS Compilers 5
1.2. File Extensions and Output Locations 5
1.3. BLISS Differences Between VAX BLISS-32 and Alpha BLISS 7
1.3.1. VAX Hardware Registers 7
1.3.2. QUAD Allocation Unit 8
1.3.3. Attributes 8
1.3.4. Linkages 10
1.3.5. Machine Specific Features 10
1.4. BLISS Differences Between Alpha BLISS and IA-64 BLISS 13
1.4.1. Machine-Specific Built-ins 13
1.4.2. IA-64 Registers 13
1.4.3. PALcode Built-in Functions 14
1.4.4. INTERRUPT and EXCEPTION Linkages 14
1.4.5. "BUILTIN *rn*" 14
1.4.6. Built-ins 14
1.4.7. BLI$CALLG 17
1.4.8. IA-64 Registers 17
1.4.9. ALPHA_REGISTER_MAPPING Switch 17
1.4.10. /ANNOTATIONS Qualifier 18
1.4.11. /ALPHA_REGISTER_MAPPING Qualifier 19
1.4.12. /ALPHA_REGISTER_MAPPING Informationals 19
1.4.13. ADD, AND, Built-in Functions for Atomic Operations 19
1.4.14. TESTBITxxI and TESTBITxx Built-in Functions for Atomic Operations 20
1.4.15. Granularity of Byte, Longword and Quadword Writes 20
1.4.16. Shift Built-in Functions 20
1.4.17. Compare and Swap Built-in Functions 21
1.4.18. IA-64-Specific Multimedia Instructions 21
1.4.19. Linkages 21
1.4.20. /[NO]TIE Qualifier 21
1.4.21. /ENVIRONMENT=([NO]FP) and ENVIRONMENT([NO]FP) 22
1.4.22. Floating Point Support 22
1.4.23. New and Expanded Lexical Functions 22
1.4.24. OpenVMS IA-64 BLISS Support for IPF Short Data Sections 23
1.5. BLISS Differences Between IA-64 BLISS and x86-64 BLISS 25
1.5.1. Floating Point Register Names 25
1.5.2. RETURNADDRESS Built-in with BLISS-32 25
1.5.3. BLI$CALLG Removed 25
1.5.4. ALPHA_REGISTER_MAPPING SWITCH and Qualifier 25
1.5.5. Unsupported and Ignored DCL Qualifiers 26
1.5.6. Built-ins Supported from OpenVMS Alpha and OpenVMS IA-64 Systems 26
1.5.7. New and Expanded Lexical Functions 28
1.6. Floating Point Support 28
1.6.1. Floating Point Built-in Functions 28
1.6.2. Floating Point Literals 28
1.6.3. Floating Point Registers 29
1.6.4. Calling Non-BLISS Routines with Floating Point Parameters 29
1.7. Documentation 29
1.8. Debugging 29
1.9. Building the STARLET and LIB .L32 and .L64 Libraries 29

2. Maintenance Corrections for OpenVMS x86-64 BLISS 30
3. Known Bugs and Deficiencies 31

1. OpenVMS X86-64 BLISS Bugfixes, Features, and Differences

This section describes the bugfixes in this release of OpenVMS x86-64 BLISS, new features, and the differences between the BLISS compilers on OpenVMS IA-64 and OpenVMS Alpha.

BLISS V1.14-144 requires OpenVMS x86-64 V9.2-1 or higher. V9.2-2 or higher is strongly recommended.

The list of corrections since the last release is found in Section 2, “Maintenance Corrections for OpenVMS x86-64 BLISS”.

1.1. The Family of BLISS Compilers

BLISS-32EN and BLISS-64EN are native compilers running on, and generating code for, OpenVMS for Alpha systems.

BLISS-32IN and BLISS-64IN are native compilers running on, and generating code for, OpenVMS for IA-64 systems.

BLISS-32XN and BLISS-64XN are native compilers running on, and generating code for, OpenVMS for x86-64 systems.

The BLISS-32xx compilers do operations 32 bits wide (i.e. BLISS values are longwords). The default width is 32 bits. In this document, they are collectively referred to as "the 32-bit compilers."

The BLISS-64xx compilers do operations 64 bits wide (i.e. BLISS values are quadwords). The default width is 64 bits. In this document, they are collectively referred to as "the 64-bit compilers."

The compilers are invoked as follows:

Compiler	**Command**
BLISS-32EN	BLISS/A32 or BLISS (On OpenVMS Alpha systems)
BLISS-64EN	BLISS/A64
BLISS-32IN	BLISS/I32 or BLISS (On OpenVMS IA-64 systems)
BLISS-64IN	BLISS/I64
BLISS-32XN	BLISS/X32 or BLISS (On OpenVMS x86-64 systems)
BLISS-64XN	BLISS/X64

1.2. File Extensions and Output Locations

The default file type for object files is .OBJ.

The default output file type for library files is .L32 for BLISS-32EN, BLISS-32IN, and BLISS-32XN; and .L64 for BLISS-64EN, BLISS-64IN, and BLISS-64XN.

Library files are NOT compatible between dialects.

The search list for BLISS-32EN is:

For source code:	.B32E, .B32, .BLI
For require files:	.R32E, .R32, .REQ
For library files:	.L32E, .L32, .LIB

The search list for BLISS-64EN is:

For source code:	.B64E, .B64, .BLI
For require files:	.R64E, .R64, .REQ
For library files:	.L64E, .L64, .LIB

The search list for BLISS-32IN is:

For source code:	.B32I, .B32, .BLI
For require files:	.R32I, .R32, .REQ
For library files:	.L32I, .L32, .LIB

The search list for BLISS-64IN is:

For source code:	.B64I, .B64, .BLI
For require files:	.R64I, .R64, .REQ
For library files:	.L64I, .L64, .LIB

The search list for BLISS-32XN is:

For source code:	.B32X, .B32, .BLI
For require files:	.R32X, .R32, .REQ
For library files:	.L32X, .L32, .LIB

The search list for BLISS-64XN is:

For source code:	.B64X, .B64, .BLI
For require files:	.R64X, .R64, .REQ
For library files:	.L64X, .L64, .LIB

The location of the output files depends on where in the command line the output qualifier was found.

If an output file qualifier such as **/OBJECT**, **/LIST**, or **/LIBRARY** is used after an input file specification and does not include an output file specification, the output file specification defaults to the device, directory, and file name of the immediately preceding input file.

For example:

```
BLISS [FOO]BAR/OBJ     -- Puts BAR.OBJ in directory FOO
BLISS /OBJ [FOO]BAR    -- Puts BAR.OBJ in default directory
BLISS [FOO]BAR/OBJ=[]  -- Puts BAR.OBJ in default directory
```

1.3. BLISS Differences Between VAX BLISS-32 and Alpha BLISS

1.3.1. VAX Hardware Registers

1.3.1.1. AP Register

The parameter-passing mechanism is different on Alpha, and there is no equivalent to the AP register.

References to AP which are used to access the parameter list can be replaced by COMMON BLISS linkage functions such as **ACTUALPARAMETER** and **ACTUALCOUNT**. The COMMON BLISS function **ARGPTR** is supported on Alpha (but requires the compiler to "fake" an argument block, and so impacts performance adversely, particularly for the 32-bit compilers). **ARGPTR** returns a pointer to a VAX-format argument list (i.e. starting with a count, and structured as a vector). Vector elements are BLISS values, 32 bits long in the 32-bit compilers and 64 bits long in the 64-bit compilers.

It is important to note that this means:

- The values in the **ARGPTR** vector are copies of the actuals, not the actuals themselves.
- There is no easy way for 32-bit code to look at the upper 32 bits of an actual parameter.

One common VAX idiom is "CALLG(.AP, ...)", which passes a routine's own input parameter list on to another routine. For fixed-length parameter lists, this is easy for users to imitate in COMMON BLISS, but for variable-length ones it is not. To support this idiom in Alpha BLISS, we provide an assembly routine BLI$CALLG on Alpha and IA-64, but replaced this with the standard LIB$CALLG on x86-64.

BLI$CALLG takes two arguments: the first is a pointer to the in-memory argument list built by the **ARGPTR** built-in, and the second is the address of a routine to call. BLI$CALLG is designed to handle a valid argument list. If anything else is passed to it, there will be unpredictable results.

We anticipate that users will replace CALLG(.AP,.RTN) by using BLI$CALLG(ARGPTR(),.RTN).

As the compiler has to "home" register arguments for **ARGPTR**, it will be more efficient to use normal calls than to use BLI$CALLG(ARGPTR(),...). This contrasts with CALLG(.AP,...), which is more efficient than a normal call.

Use of the AP as a scratch register holding a local variable is not portable, but the larger number of registers offered on Alpha will allow another register to be used in its place.

The name "AP" is not recognized by the compilers, and will cause an error.

1.3.1.2. FP Register

The Alpha calling standards do not directly support dynamic condition handling, so constructs of the type .FP = handler are invalid.

COMMON BLISS has the **ENABLE** mechanism to establish static condition handlers.

The Alpha BLISS compilers provide two new built-ins named **ESTABLISH** and **REVERT**. Their semantics parallel the VAX idiom of assignments to the location pointed to by the FP register.

- **ESTABLISH(rtn)**

  Establishes RTN as the current handler. It has no value.

- **REVERT()**

  Disestablishes any established handler routine. It also has no value.

These built-ins cannot be used in the same routine as **ENABLE**. No enable vector is passed to a handler that is established using **ESTABLISH**.

When a routine which has established a handler returns, that handler is disestablished automatically.

Programs that inspect the frame will have to be re-written, as the frame layout has changed for Alpha.

As all references to the VAX FP must be changed, use of the name "FP" in a **BUILTIN** declaration is an error. Access to the Alpha FP may be achieved by use of register 29, the Alpha frame pointer. However, this is almost certain to cause a register conflict and prevent the generation of code; we recommend that all manipulation of the real FP take place in Alpha assembler.

1.3.1.3. SP Register

As all references to the VAX SP must be changed, use of the name "SP" in a **BUILTIN** declaration is an error. Access to the Alpha SP may be achieved by use of register 30, the Alpha stack pointer. However, this is almost certain to cause a register conflict and prevent the generation of code; we recommend that all manipulation of the real SP take place in Alpha assembler.

1.3.1.4. PC Register

Use of the name "PC" is an error, as there is no equivalent to the VAX PC on Alpha.

1.3.2. QUAD Allocation Unit

The 64-bit compilers recognize QUAD as specifying a data-segment of eight bytes (64 bits), in the same way that LONG is recognized as specifying 32 bits. QUAD is not recognized by the 32-bit compilers, as the 32-bit version of the BLISS language provides no method for manipulating data-segments larger than 32 bits in size.

QUAD may be used in the 64-bit compilers wherever BYTE, WORD, and LONG may be used in BLISS-32.

The default size for data-segments in the 64-bit compilers is LONG or QUAD, depending on the presence or absence of the compile-time switch **/ASSUME=LONG_DEFAULT** or **/ASSUME=SIGNED_LONG**.

1.3.3. Attributes

1.3.3.1. ALIGN Attribute

The ALIGN attribute is allowed on **EXTERNAL**, **BIND**, and **GLOBAL BIND** declarations, in addition to the **OWN**, **GLOBAL**, **LOCAL**, and **STACKLOCAL** declarations on which BLISS-32 allows it. In the Alpha compilers, in addition to telling the compiler on what boundaries to allocate data, it tells the compiler what assumptions it can make regarding the alignment of data that the compiler does not allocate in this compilation unit.

1.3.3.2. ALIAS Attribute

The ALIAS attribute indicates that a variable may be changed by routine calls and indirect assignments. ALIAS may be used on any of the following declarations: **EXTERNAL**, **GLOBAL**, **OWN**, **FORWARD**, **MAP**, **BIND**, **GLOBAL BIND**, **LOCAL**, and **STACKLOCAL**.

The BLISS language assumes that named storage segments will in general be changed only by using the name of the variable. Also, BLISS expects that two different named pointer variables do not point to the same block of storage.

If it is necessary to have more than one pointer to the same named variable or block of storage, these pointers should be declared with the ALIAS attribute.

Any variable passed using an OpenVMS item-list must be treated as ALIAS.

The Alpha BLISS compilers automatically mark as ALIAS any variable whose address is used in a context other than fetch or store. These are the variables that got the "consider VOLATILE" diagnostic when **/CHECK=ADDRESS_TAKEN** was used. This diagnostic now reads: "assuming ALIAS".

In most cases, Alpha-only BLISS source code can rely on the automatic ALIAS analysis by the Alpha BLISS compiler. Since VAX BLISS-32 does not do this, code that is common between Alpha and VAX needs to use explicit ALIAS attributes. This requires VAX BLISS V4.7; older versions do not support the ALIAS attribute.

1.3.3.3. VOLATILE Attribute

The VOLATILE attribute is stronger in Alpha BLISS than in VAX BLISS. On VAX, VOLATILE means that the variable can change at any time. On Alpha, it has the following properties:

- Always causes one memory access for long/quad read/write and byte/word reads
- Strict source ordering (no reordering of reads, for example)
- Proper size (for long/quad)
- Only supported for aligned accesses
- Yields byte granularity

As a result, VOLATILE is sufficient for I/O registers.

This may require source code changes in existing Alpha BLISS programs. Specifically, writing to an unaligned 16-bit (word) field in a VOLATILE structure causes an unrecoverable alignment (ROPRAND) fault.

The compiler has added a diagnostic to issue a warning for unaligned VOLATILE references:

```
%BLS32-W-UNAVOLACC, volatile access appears unaligned, but must be
aligned at run-time
```

BLISS users should modify their code to completely eliminate all UNAVOLACC warnings.

The following *must* be done to ensure correct operation:

- Any references to VOLATILE unaligned 16-bit words must be changed to eliminate unrecoverable faults.
- All uses of structures that contain 16-bit words should be examined to ensure that either:
  - The 16-bit field is aligned, or
  - The structure is not accessed using **REF VOLATILE**

Additionally, the following *should* be done to get good performance:

- Unaligned access of other types are recoverable but take extra time. In almost all cases, it is desirable to change the VOLATILE attribute to ALIAS to avoid unnecessary alignment faults.

1.3.4. Linkages

1.3.4.1. CALL linkage

The **CALL** linkage is the standard linkage on Alpha. See the *VSI OpenVMS Calling Standard* [https://docs.vmssoftware.com/vsi-openvms-calling-standard/] for details.

Routines compiled with a 32-bit compiler can call routines compiled with a 64-bit compiler and vice versa. Parameters are truncated when shortened, and sign-extended when lengthened.

By default, **CALL** linkages pass an argument count. This can be overridden using the NOCOUNT linkage option described below.

Although the arguments are passed in quadwords, the 32-bit compilers can only "see" the lower 32 bits.

1.3.4.2. JSB Linkage

The OpenVMS compilers have a JSB linkage type. Routines declared with the JSB linkage type are frameless routines, i.e. they do not modify the FP register.

1.3.4.3. Global Registers

Routines with linkages with GLOBAL REGISTERS which did not declare those registers as EXTERNAL REGISTERS follow the BLISS-32 rules:

- If the global register is not declared at all, its value will be preserved.
- If the register is specifically declared (e.g. as a GLOBAL register), the value of the register will be preserved and a new, nested lifetime started for the register.

1.3.4.4. INTERRUPT and EXCEPTION Linkages

The OpenVMS compilers have **INTERRUPT** and **EXCEPTION** linkage types. See the *Alpha Architecture Reference Manual* for details. Routines with these linkages cannot be called from BLISS.

1.3.4.5. COUNT and NOCOUNT Linkage Attributes

The linkage attributes COUNT and NOCOUNT allow the user to specify whether the argument count should be passed from the caller to the callee. These attributes are legal only for the **CALL** linkage type. The default is COUNT. The default can be controlled on a module-wide basis by using either the command line qualifier **/[NO]COUNT** or the module head switch [NO]COUNT. As usual, a switch setting given in a module head overrides the command line qualifier. The linkage functions **ACTUALCOUNT**, **ACTUALPARAMETER**, **NULLPARAMETER**, and **ARGPTR** may not be used in routines whose linkages are NOCOUNT.

1.3.5. Machine Specific Features

1.3.5.1. Alpha Registers

Alpha integer registers are available, using the same syntax used on the VAX for access to VAX registers. Alpha floating registers are not available.

Many of the registers are not available for use by user code, as they are reserved for special uses. A register conflict message will be issued when a user request for a particular register cannot be satisfied.

1.3.5.2. PALcode Built-in Functions

The following PALcode instructions are available as built-in functions in the OpenVMS compilers:

```
BPT       INSQHIL     REMQHIL     BUGCHK     MTPR_ASTSR    MFPR_MCES
CHME      INSQTIL     REMQTIL     IMB        MTPR_FEN      MFPR_SSP
CHMK      INSQUEL     REMQUEL     PROBER     MTPR_PRBR     MFPR_WHAMI
CHMS      INSQHIQ     REMQHIQ     PROBEW     MTPR_SIRR     MFPR_PTBR
CHMU      INSQTIQ     REMQTIQ     RD_PS      MTPR_TBIAP    MFPR_SISR
HALT      INSQUEQ     REMQUEQ     SWASTEN    MTPR_IPIR     MFPR_IPL
          INSQUEL_D   REMQUEL_D   WR_PS_SW   MTPR_MCES     MFPR_PCBB
          INSQUEQ_D   REMQUEQ_D   CFLUSH     MTPR_TBIS     MFPR_SCBB
          INSQHILR    REMQHILR    DRAINA     MTPR_SSP      MFPR_TBCHK
          INSQHIQR    REMQHIQR    LDQP       MTPR_ASTEN    MFPR_ESP
          INSQTILR    REMQTILR    STQP       MTPR_IPL      MFPR_USP
          INSQTIQR    REMQTIQR    SWPCTX     MTPR_SCBB     MFPR_FEN
                                  GENTRAP    MTPR_TBIA     MFPR_PRBR
                                  RSCC       MTPR_ESP      MFPR_ASTEN
                                  READ_UNQ   MTPR_USP      MFPR_ASTSR
                                  WRITE_UNQ  MTPR_VPTB     MFPR_VPTB
                                             MTPR_TBISD
                                             MTPR_TBISI
                                             MTPR_DATFX
                                             MTPR_ASN
                                             MTPR_PERFMON
```

The following PALcode instructions are available as built-in functions in the other compilers:

```
BPT       IMB         HALT      GENTRAP
```

Refer to the *Alpha Architecture Reference Manual* for information regarding inputs and outputs. Please note that all of above names are preceded by a **PAL_** to distinguish them from similar VAX built-ins.

CALL_PAL is a generic PALcode built-in. The first parameter, which must be a compile-time constant expression, is the function field of the **CALL_PAL** instruction. The remaining parameters are the inputs to the **CALL_PAL** instruction.

1.3.5.3. New Built-in Functions for Atomic Operations

The following new built-in functions allow atomic updating of memory:

```
ADD_ATOMIC_LONG, ADD_ATOMIC_QUAD, AND_ATOMIC_LONG, AND_ATOMIC_QUAD,
OR_ATOMIC_LONG, OR_ATOMIC_QUAD
```

The operations have the form:

```
<op>_ATOMIC_<size>( ptr, expr

        [ , retry_count ]       ! Optional input
        [; old_value    ] )     ! Optional output

   Value:       1       Operation succeeded
                0       Operation failed

   <op> is one of AND, ADD, OR
   <size> is one of LONG or QUAD
```

The operation is addition (or ANDing or ORing) of the expression *expr* to the data-segment pointed to by *ptr* within a load-locked/store-conditional code sequence. The operation will be tried *retry_count* times. If the operation cannot be performed successfully in the specified number of trials, the built-in's value is zero.

The value of *ptr* must be a naturally-aligned address.

The optional output parameter *old_value* is set to the previous value of the data-segment pointed to by *ptr*.

For the 32-bit compilers, the *expr* parameter will be sign-extended and the *old_value* parameter will be truncated for the QUAD operations.

1.3.5.4. Compatible Built-in Functions for Atomic Operations

TESTBITSSI, **TESTBITCCI**, **TESTBITSS**, **TESTBITSC**, **TESTBITCS**, **TESTBITCC**, and **ADAWI** have been implemented in an upward-compatible manner.

The **TESTBITxx** built-ins are AST-atomic. This is a weaker form of atomicity than the **TESTBITxxI** built-ins have. The operations have the form:

```
 TESTBITxxx( field
        [ , retry_count ]       ! Optional input
        [; success-flag ] )     ! Optional output

   Value:       1       Bit was set (TESTBITSSI) or clear (TESTBITCCI)
                0       Otherwise
```

BLISS-32's **ADAWI** returns the contents of the PSL. Since the PSL doesn't exist on Alpha, the return value of **ADAWI** is a simulated partial VAX PSL, where only the condition codes are significant:

```
ADAWI( address, addend)

   Value:       PSL 0:3 (Simulated partial VAX PSL)
```

1.3.5.5. New Shift Built-in Functions

Built-in functions for shifts in a known direction have been added. They are only valid for shift amounts in the range 0..%BPVAL-1.

```
result = SLL(value, amount)         Shift left logical
    result = SRL(value, amount)         Shift right logical
    result = SRA(value, amount)         Shift right arithmetic
```

1.3.5.6. Other Machine-Specific Built-in Functions

Other machine specific built-in functions are:

ROT(***value***, ***shift***)

Rotates *value* by *shift* bits, returning the rotated value.

TRAPB()

Generates the TRAPB instruction.

RPCC()

Generates the RPCC instruction.

WRITE_MBX(*dest-address*, *value-to-store*)

Generates the STQ_C instruction required for writing to mailboxes. Refer to the *Alpha Architecture Reference Manual* for further details. This function returns 1 for success and 0 for failure.

UMULH
CMPBGE
ZAP
ZAPNOT

UMULH, **CMPBGE**, **ZAP**, and **ZAPNOT** each have two input parameters and a value. They correspond to the Alpha instructions with the same names.

CMP_STORE_LONG(*addr*, *comparand*, *value*, *destination*)
CMP_STORE_LONG(*addr*, *comparand*, *value*, *destination*)

CMP_STORE_LONG and **CMP_STORE_QUAD** do the following interlocked operations: compare the longword or quadword at *addr* with *comparand*, and if they are equal, store *value* at *destination*. They return an indicator of success (1) or failure (0).

1.4. BLISS Differences Between Alpha BLISS and IA-64 BLISS

This section describes those Alpha BLISS features that are not be supported by OpenVMS IA-64 BLISS.

1.4.1. Machine-Specific Built-ins

The following Alpha BLISS machine-specific built-ins are not supported:

```
RPCC          ZAP
TRAPB         ZAPNOT
DRAINT        CMP_STORE_LONG
WRITE_MBX     CMP_STORE_QUAD
CMPBGE
```

CMP_STORE_LONG and **CMP_STORE_QUAD** are replaced by **CMP_SWAP_LONG** and **CMP_SWAP_QUAD**.

1.4.2. IA-64 Registers

The following IA-64 registers are not supported for naming in REGISTER, GLOBAL REGISTER, or EXTERNAL REGISTER, or as parameters to LINKAGE declarations:

R0	zero register
R1	global pointer
R2	volatile and GEM scratch register
R12	stack pointer
R13	thread pointer
R14-R16	volatile and GEM scratch registers
R17-R18	volatile scratch registers

1.4.3. PALcode Built-in Functions

The following Alpha BLISS PALcode built-ins are not supported:

```
CALL_PAL          PAL_MFPR_PCBB      PAL_MTPR_SIRR
PAL_BPT           PAL_MFPR_PRBR      PAL_MTPR_SSP
PAL_BUGCHK        PAL_MFPR_PTBR      PAL_MTPR_TBIA
PAL_CFLUSH        PAL_MFPR_SCBB      PAL_MTPR_TBIAP
PAL_CHME          PAL_MFPR_SISR      PAL_MTPR_TBIS
PAL_CHMK          PAL_MFPR_SSP       PAL_MTPR_TBISD
PAL_CHMS          PAL_MFPR_TBCHK     PAL_MTPR_TBISI
PAL_CHMU          PAL_MFPR_USP       PAL_MTPR_USP
PAL_DRAINA        PAL_MFPR_VPTB      PAL_MTPR_VPTB
PAL_HALT          PAL_MFPR_WHAMI     PAL_PROBER
PAL_GENTRAP       PAL_MTPR_ASTEN     PAL_PROBEW
PAL_IMB           PAL_MTPR_ASTSR     PAL_RD_PS
PAL_LDQP          PAL_MTPR_DATFX     PAL_READ_UNQ
PAL_MFPR_ASN      PAL_MTPR_ESP       PAL_RSCC
PAL_MFPR_ASTEN    PAL_MTPR_FEN       PAL_STQP
PAL_MFPR_ASTSR    PAL_MTPR_IPIR      PAL_SWPCTX
PAL_MFPR_ESP      PAL_MTPR_IPL       PAL_SWASTEN
PAL_MFPR_FEN      PAL_MTPR_MCES      PAL_WRITE_UNQ
PAL_MFPR_IPL      PAL_MTPR_PRBR      PAL_WR_PS_SW
PAL_MFPR_MCES     PAL_MTPR_SCBB      PAL_MTPR_PERFMON
```

Macros are provided in STARLET.REQ for PALCALL built-ins. The privileged **CALL_PAL**s call exec routines and the unprivileged **CALL_PAL**s execute system services.

1.4.4. INTERRUPT and EXCEPTION Linkages

The Alpha **INTERRUPT** and **EXCEPTION** linkages are not supported.

1.4.5. "BUILTIN *rn*"

The ability to specify an IA-64 register name to the BUILTIN keyword is not supported.

1.4.6. Built-ins

1.4.6.1. Common BLISS Built-ins

The following existing Common BLISS built-ins are supported:

```
ABS               CH$FIND_NOT_CH  CH$WCHAR
ACTUALCOUNT       CH$FIND_SUB     CH$WCHAR_A
ACTUALPARAMETER   CH$GEQ          MAX
ARGPTR            CH$GTR          MAXA
BARRIER           CH$LEQ          MAXU
CH$ALLOCATION     CH$LSS          MIN
CH$A_RCHAR        CH$MOVE         MINA
CH$A_WCHAR        CH$NEQ          MINU
CH$COMPARE        CH$PLUS         NULLPARAMETER
CH$COPY           CH$PTR          REF
CH$DIFF           CH$RCHAR        SETUNWIND
CH$EQL            CH$RCHAR_A      SIGN
CH$FAIL           CH$SIZE         SIGNAL
CH$FILL           CH$TRANSLATE    SIGNAL_STOP
CH$FIND_CH        CH$TRANSTABLE
```

1.4.6.1.1. RETURNADDRESS Built-in

A new built-in function **RETURNADDRESS** returns the PC of the caller's caller.

This built-in takes no arguments and the format is:

```
RETURNADDRESS()
```

1.4.6.2. Machine-Specific Built-ins

The following Alpha BLISS machine-specific built-ins are supported:

```
BARRIER
ESTABLISH
REVERT

ROT
SLL
SRA
SRL
UMULH

ADAWI

ADD_ATOMIC_LONG   AND_ATOMIC_LONG  OR_ATOMIC_LONG
ADD_ATOMIC_QUAD   AND_ATOMIC_QUAD  OR_ATOMIC_QUAD
```

The **XXX_ATOMIC_XXX** built-ins no longer support the optional *retry-count* input argument.

```
TESTBITSSI TESTBITCC  TESTBITCS
TESTBITCCI TESTBITSS  TESTBITSC
```

The **TESTBITxx** instructions no longer support the optional *retry-count* input argument or the optional *success-flag* output argument.

```
ADDD    DIVD    MULD    SUBD    CMPD
ADDF    DIVF    MULF    SUBF    CMPF
ADDG    DIVG    MULG    SUBG    CMPG
ADDS    DIVS    MULS    SUBS    CMPS
ADDT    DIVT    MULT    SUBT    CMPT

CVTDF   CVTFD   CVTGD   CVTSF   CVTTD
CVTDG   CVTFG   CVTGF   CVTSI   CVTTG
CVTDI   CVTFI   CVTGI   CVTSL   CVTTI
CVTDL   CVTFL   CVTGL   CVTSQ   CVTTL
CVTDQ   CVTFQ   CVTGQ   CVTST   CVTTQ
CVTDT   CVTFS   CVTGT           CVTTS

CVTID   CVTLD   CVTQD
CVTIF   CVTLF   CVTQF
CVTIG   CVTLG   CVTQG
CVTIS   CVTLS   CVTQS
CVTIT   CVTLT   CVTQT

CVTRDL  CVTRDQ
CVTRFL  CVTRFQ
CVTRGL  CVTRGQ
CVTRSL  CVTRSQ
CVTRTL  CVTRTQ
```

1.4.6.3. New Machine-Specific Built-ins

A number of new built-ins have been added that provide access to single IA-64 instructions which may be used by the operating system.

1.4.6.3.1. Built-ins for Single Instructions

Each name capitalized below is a new built-in function which may be specified. The lower-case name in parentheses is the actual IA-64 instruction executed. The arguments to these instructions (and therefore their associated BLISS built-in names) are detailed in the *Intel IA-64 Architecture Software Developer's Manual*.

```
BREAK   (break)     LOADRS (loadrs)    RUM   (rum)     HINT   (hint)
BREAK2  (break)     PROBER (probe.r)   SRLZD (srlz.d)
FC      (fc)        PROBEW (probe.w)   SRLZI (srlz.i)
FLUSHRS (flushrs)   PCTE   (ptc.e)     SSM   (ssm)
FWB     (fwb)       PCTG   (ptc.g)     SUM   (sum)
INVALAT (invala)    PCTGA  (ptc.ga)    SYNCI (sync.i)
ITCD    (itc.d)     PTCL   (ptc.l)     TAK   (tak)
ITCI    (itc.i)     PTRD   (ptr.d)     THASH  (thash)
ITRD    (itr.d)     PTRI   (ptr.i)     TPA    (tpa)
ITRI    (itr.i)     RSM    (rsm)       TTAG   (ttag)
```

Note

The **BREAK2** built-in requires two parameters. The first parameter, which must be a compiletime literal, specifies the 21-bit immediate value of the **BREAK** instruction. The second parameter may be any expression whose value is moved into register R17 just prior to executing the **BREAK** instruction.

1.4.6.3.2. Access to Processor Registers

The OpenVMS IA-64 BLISS compiler provides built-in functions for read and write access to the many and varied processor registers in the IA-64 implementations. They are:

```
GETREG  SETREG  GETREGIND  SETREGIND
```

These built-ins execute the mov.i instruction, which is detailed in the *Intel IA-64 Architecture Software Developer's Manual*.

The two **GET** built-ins return the value of the register specified.

To specify the register, a specially encoded integer constant is used, which is defined in an Intel C header file.

1.4.6.4. PALcode Built-ins

The following Alpha BLISS PALcode built-ins are supported:

```
PAL_INSQHIL     PAL_REMQHIL
PAL_INSQHILR    PAL_REMQHILR
PAL_INSQHIQ     PAL_REMQHIQ
PAL_INSQHIQR    PAL_REMQHIQR
PAL_INSQTIL     PAL_REMQTIL
PAL_INSQTILR    PAL_REMQTILR
PAL_INSQTIQ     PAL_REMQTIQ
PAL_INSQTIQR    PAL_REMQTIQR
PAL_INSQUEL     PAL_REMQUEL
```

```
PAL_INSQUEL_D   PAL_REMQUEL_D
PAL_INSQUEQ     PAL_REMQUEQ
PAL_INSQUEQ_D   PAL_REMQUEQ_D
```

The 24 queue-manipulation PALcalls are implemented by BLISS as a call to a OpenVMS-provided SYS$PAL_xxxx run-time routine.

1.4.7. BLI$CALLG

The VAX idiom CALLG(.AP, ...) was replaced by an assembly routine BLI$CALLG(ARGPTR(), .RTN) for OpenVMS Alpha BLISS. This routine as defined for OpenVMS Alpha BLISS will be re-written for the IA-64 architecture and supported for OpenVMS IA-64 BLISS.

1.4.8. IA-64 Registers

The IA-64 general registers which may be named in REGISTER, GLOBAL REGISTER, and EXTERNAL REGISTER, and as parameters to LINKAGE declarations, are as follows:

- R3 to R11
- R19 to R31

In addition, eight parameter registers will be able to be named for parameters in LINKAGE declarations only. They are R32 to R39.

There is no support for accessing the IA-64 general registers R40 to R127.

Naming of any of the IA-64 Floating Point, Predicate, Branch, and Application registers via the REGISTER, GLOBAL REGISTER, EXTERNAL REGISTER, and LINKAGE declarations is not supported.

A register conflict message is issued when a user request for a particular register cannot be satisfied.

1.4.9. ALPHA_REGISTER_MAPPING Switch

A new module level switch ALPHA_REGISTER_MAPPING is being provided for OpenVMS IA-64 BLISS.

This switch may be specified either in the **MODULE** declaration or a **SWITCHES** declaration. Use of this switch will cause a re-mapping of Alpha register numbers to IA-64 register numbers as described below.

Any register number specified as part of a REGISTER, GLOBAL REGISTER, EXTERNAL REGISTER, or as parameters to **GLOBAL**, **PRESERVE**, **NOPRESERVE**, or **NOT USED** in linkage declarations in the range of 0-31 will be remapped according to the IMACRO mapping table as follows:

```
0  =    GEM_TS_REG_K_R8      16 =    GEM_TS_REG_K_R14
1  =    GEM_TS_REG_K_R9      17 =    GEM_TS_REG_K_R15
2  =    GEM_TS_REG_K_R28     18 =    GEM_TS_REG_K_R16
3  =    GEM_TS_REG_K_R3      19 =    GEM_TS_REG_K_R17
4  =    GEM_TS_REG_K_R4      20 =    GEM_TS_REG_K_R18
5  =    GEM_TS_REG_K_R5      21 =    GEM_TS_REG_K_R19
6  =    GEM_TS_REG_K_R6      22 =    GEM_TS_REG_K_R22
7  =    GEM_TS_REG_K_R7      23 =    GEM_TS_REG_K_R23
8  =    GEM_TS_REG_K_R26     24 =    GEM_TS_REG_K_R24
9  =    GEM_TS_REG_K_R27     25 =    GEM_TS_REG_K_R25
10 =    GEM_TS_REG_K_R10     26 =    GEM_TS_REG_K_R0
11 =    GEM_TS_REG_K_R11     27 =    GEM_TS_REG_K_R0
```

```
12 =    GEM_TS_REG_K_R30     28 =    GEM_TS_REG_K_R0
13 =    GEM_TS_REG_K_R31     29 =    GEM_TS_REG_K_R29
14 =    GEM_TS_REG_K_R20     30 =    GEM_TS_REG_K_R12
15 =    GEM_TS_REG_K_R21     31 =    GEM_TS_REG_K_R0
```

The mappings for register numbers:

16-20

26-28

30-31

Translate into registers which are considered invalid specifications for OpenVMS IA-64 BLISS. Declarations including any these registers when ALPHA_REGISTER_MAPPING is specified generate an error similar to the following:

```
         r30 = 30,
.........^
%BLS64-W-TEXT, Alpha register 30 cannot be declared, invalid mapping to
IPF register 12 at line number 9 in file ddd:[xxx]TESTALPHAREGMAP.BLI
```

Note that the source line names register number `30`, but the error text indicates register `12` is the problem. It is the translated register for `30`, register `12`, which is illegal to specify.

1.4.9.1. ALPHA_REGISTER_MAPPING and Linkage Declarations

There is a special set of mappings for Alpha registers R16 to R21, if those registers are specified as linkage I/O parameters.

For linkage I/O parameters *only*, the mappings for R16 to R21 are as follows:

```
16 =    GEM_TS_REG_K_R32     19 =    GEM_TS_REG_K_R35
17 =    GEM_TS_REG_K_R33     20 =    GEM_TS_REG_K_R36
18 =    GEM_TS_REG_K_R34     21 =    GEM_TS_REG_K_R37
```

1.4.9.1.1. ALPHA_REGISTER_MAPPING and "NOTUSED"

When ALPHA_REGISTER_MAPPING is specified, any Alpha register that maps to an IA-64 scratch register and is specified as NOTUSED in a linkage declaration will be placed in the PRESERVE set.

This will cause the register to be saved on entry to the routine declaring it NOTUSED and restored on exit.

1.4.10. /ANNOTATIONS Qualifier

The OpenVMS IA-64 BLISS compiler will support a new compilation qualifier **`/ANNOTATIONS`**. This qualifier provides information in the source listing regarding optimizations that the compiler is (or is not) making during compilation.

The qualifier accepts a number of keywords that reflect the different listing annotations available. They are:

ALL	
NONE	
CODE	Used for annotations of machine code listing. Only NOP instructions are currently annotated.

DETAIL	Provides greater detail. Used in conjunction with other keywords.

The remaining keywords reflect GEM optimizations:

```
INLINING
LOOP_TRANSFORMS
LOOP_UNROLLING
PREFETCHING
SHRINKWRAPPING
SOFTWARE_PIPELINING
TAIL_CALLS
TAIL_RECURSION
LINKAGES
```

All keywords with the exception of ALL and NONE are negatable. The qualifier itself is also negatable. By default it is not present in the command line.

If the **/ANNOTATIONS** qualifier is specified without any parameters, the default is ALL.

1.4.11. /ALPHA_REGISTER_MAPPING Qualifier

The OpenVMS IA-64 BLISS compiler supports the new compilation qualifier **/ALPHA_REGISTER_MAPPING**, which enables ALPHA_REGISTER_MAPPING without having to modify the source.

This is a positional qualifier.

Specifying this qualifier on the compilation line for a module is equivalent to setting the ALPHA_REGISTER_MAPPING switch in the module header.

1.4.12. /ALPHA_REGISTER_MAPPING Informationals

For OpenVMS IA-64 BLISS, new informational messages have been added to show the usage of the ALPHA_REGISTER_MAPPING feature.

If the switch ALPHA_REGISTER_MAPPING is specified in the module header or as an argument to the **SWITCHES** declaration, the following will be displayed:

```
MODULE SIMPLE (MAIN=TEST, ALPHA_REGISTER_MAPPING)=
.........................^
%BLS64-I-TEXT, Alpha Register Mapping enabled
```

If the switch NOALPHA_REGISTER_MAPPING is specified in the module header or as an argument to the **SWITCH** declaration, the following will be displayed:

```
MODULE SIMPLE (MAIN=TEST, NOALPHA_REGISTER_MAPPING)=
.........................^
%BLS64-I-TEXT, Alpha Register Mapping disabled
```

1.4.13. ADD, AND, Built-in Functions for Atomic Operations

The **ADD_ATOMIC_XXXX**, **AND_ATOMIC_XXXX**, and **OR_ATOMIC_XXXX** built-in functions for atomic updating of memory are supported by OpenVMS IA-64 BLISS.

As the IA-64 instructions to support these built-ins will wait until the operation succeeds, the optional *retry-count* input parameter has been eliminated. These built-ins now have the form:

```
<op>_ATOMIC_<size>(ptr, expr
                     [;old_value] ) !Optional output

   Value:   1 Operation succeeded
            0 Operation failed

  <op> is one of AND, ADD OR
  <size> is one of LONG or QUAD
```

The operation is addition (or ANDing or ORing) of the expression *expr* to the data-segment pointed to by *ptr* in an atomic fashion.

The value of *ptr* must be a naturally-aligned address.

The optional output parameter *old_value* is set to the previous value of the data-segment pointed to by PTR.

Any attempt to use the OpenVMS Alpha BLISS optional *retry_count* argument results in a syntax error.

1.4.14. TESTBITxxI and TESTBITxx Built-in Functions for Atomic Operations

The **TESTBITxxI** and **TESTBITxx** built-in functions for atomic operations are supported by OpenVMS IA-64 BLISS.

Because the IA-64 instruction to support these built-ins will wait until the operation succeeds, the optional input parameter *retry_count* and the optional output parameter *success_flag* have been eliminated.

These built-ins now have the form:

```
TESTBITxxx( field )
```

Any attempt to use the OpenVMS Alpha BLISS optional *retry_count* or *success_flag* arguments results in a syntax error.

1.4.15. Granularity of Byte, Longword and Quadword Writes

OpenVMS IA-64 BLISS will support the **/GRANULARITY=*keyword*** qualifier, the DEFAULT_GRANULARITY=*n* switch, and the GRANUALRITY(*n*) data attribute as described below.

Users can control the granularity of stores and fetches by using the **/GRANULARITY=*keyword*** command line qualifier, the DEFAULT_GRANULARITY=*n* switch, and the GRANUALRITY(*n*) data attribute .

The keyword in the command line qualifier must be either BYTE, LONGWORD, or QUADWORD. The parameter *n* must be either 0(byte), 2(longword), or 3(quadword).

When these are used together, the data attribute has the highest priority. The switch, when used in a **SWITCHES** declaration, sets the granularity of data declared after it within the same scope. The switch may also be used in the module header. The command line qualifier has the lowest priority.

1.4.16. Shift Built-in Functions

The Alpha built-in functions for shifts in a known direction are supported for OpenVMS IA-64 BLISS.

They are only valid for shift amounts in the range 0..%BPVAL-1.

1.4.17. Compare and Swap Built-in Functions

OpenVMS IA-64 provides support for the following new compare and swap built-in functions:

CMP_SWAP_LONG(*addr*, *comparand*, *value*)
CMP_SWAP_QUAD(*addr*, *comparand*, *value*)

These functions do the following interlocked operations: compare the longword or quadword at *addr* with *comparand*, and if they are equal, store *value* at *addr*. They return an indicator of success (1) or failure (0).

Note

These new built-in functions are provided for OpenVMS Alpha BLISS as well.

1.4.18. IA-64-Specific Multimedia Instructions

There are no plans to support access to the IA-64-specific multimedia-type instructions.

1.4.19. Linkages

1.4.19.1. CALL Linkage

The **CALL** linkage, as described below for OpenVMS Alpha Bliss, is supported by OpenVMS IA-64 BLISS.

Routines compiled with a 32-bit compiler can call routines compiled with a 64-bit compiler, and vice versa. Parameters are truncated when shortened and sign-extended when lengthened.

By default, **CALL** linkages pass an argument count. This can be overridden using the NOCOUNT linkage option.

Although the arguments are passed in quadwords, the 32-bit compilers can only "see" the lower 32 bits.

1.4.19.2. JSB Linkage

The OpenVMS IA-64 BLISS compilers have a JSB linkage type. Routines declared with the JSB linkage will fit in with the JSB rules as defined by the OpenVMS Calling Standard.

1.4.20. /[NO]TIE Qualifier

This qualifier is supported for OpenVMS IA-64.

/TIE is used to enable the compiled code to be used in combination with translated images, because the code might call into, or be called from, a translated image.

In particular, **/TIE**:

- Causes the inclusion of procedure signature information in the compiled program. This may increase the size and possibly also the number of relocations processed during linking and image activation, but does not otherwise affect performance.
- Causes calls to procedure values (sometimes called indirect or computed calls) to be compiled using a service routine (OTS$CALL_PROC); this routine determines whether the target procedure is native IPF code, or in a translated image, and proceeds accordingly.

/NOTIE is the default.

1.4.21. /ENVIRONMENT=([NO]FP) and ENVIRONMENT([NO]FP)

The **/ENVIRONMENT=([NO]FP)** qualifier and the ENVIRONMENT([NO]FP) switch were provided for OpenVMS Alpha BLISS to cause the compiler to disable the use of floating point registers for certain integer division operations.

For OpenVMS IA-64 BLISS, the **/ENVIRONMENT=NOFP** command qualifier or ENVIRONMENT(NOFP) switch does not totally disable floating point due to the architectural features of IA-64. Instead, source code is still restricted to not have floating point operations, but the generated code for certain operations (in particular, integer multiplication and division and the constructs that imply them) are restricted to use a small subset of the floating point registers. Specifically, if this option is specified, the compiler is restricted to using f6-f11, and will set the ELF EF_IA_64_REDUCEFP option as described in Section 4.1.1.6 "Processor-specific Flags" in the *Intel Itanium Processor-specific Application Binary Interface*.

The **/ENVIRONMENT=FP** command qualifier and ENVIRONMENT(FP) switch are unaffected.

1.4.22. Floating Point Support

1.4.22.1. Floating Point Built-in Functions

BLISS does not have a high level of support for floating-point numbers. The extent of the support involves the ability to create floating-point literals, and there are machine-specific built-ins for floating-point arithmetic and conversion operations.

None of the floating point built-in functions detect overflow, so they do not return a value.

1.4.22.2. Floating Point Literals

The floating point literals supported by OpenVMS IA-64 BLISS is the same set supported by OpenVMS Alpha BLISS: %E, %D, %G, %S, and %T.

1.4.22.3. Floating Point Registers

Direct use of the IA-64 floating-point registers is not supported.

1.4.22.4. Calling Non-BLISS Routines with Floating Point Parameters

It is possible to call standard non-BLISS routines that expect floating-point parameters passed by value, and that return a floating-point or complex value.

The standard functions %FFLOAT, %DFLOAT, %GFLOAT, %SFLOAT, and %TFLOAT are supported by OpenVMS IA-64 BLISS.

1.4.23. New and Expanded Lexical Functions

BLISS will add new compiler-state lexical functions to support the OpenVMS IA-64 compilers: BLISS32I and BLISS64I.

- %BLISS will now recognize BLISS32E, BLISS64E, BLISS32V, BLISS32I, and BLISS64I.

  %BLISS(BLISS32) is true for all 32-bit BLISS compilers.

%BLISS(BLISS32V) is true only for VAX BLISS (BLISS-32).

%BLISS(BLISS32E) is true for all 32-bit Alpha compilers.

%BLISS(BLISS64E) is true for all 64-bit Alpha compilers.

%BLISS(BLISS32I) is true for all 32-bit IA-64 compilers.

%BLISS(BLISS64I) is true for all 64-bit IA-64 compilers.

- The lexical functions %BLISS32I and %BLISS64I have been added. Their behavior matches that of the new parameters to %BLISS.
- Support for the IA-64 architecture (with the I64 keyword) has been added to the %HOST and %TARGET lexical functions for OpenVMS IA-64 BLISS.

1.4.24. OpenVMS IA-64 BLISS Support for IPF Short Data Sections

The OpenVMS Calling Standard requires that all global data objects with a size of 8 bytes or smaller be allocated in short data sections.

Short data sections can be addressed with an efficient code sequence that involves adding a 22-bit literal to the contents of the GP base register. This code sequence limits the combined size of all the short data sections. A linker error will occur if the total amount of data allocated to short data sections exceeds a size of 2**22 bytes.

Compilers on IA-64 can use GP relative addressing when accessing short globals and short externals.

OpenVMS IA-64 BLISS has two new PSECT attributes, GP_RELATIVE and SHORT, to support allocating short data sections.

Specifying the GP_RELATIVE keyword as a PSECT attribute causes that PSECT to be labeled as containing short data so that the linker will allocate the PSECT close to the GP base address.

The syntax of the SHORT attribute is as follows:

```
"SHORT" "(" psect-name ")"
```

The following rules apply to the SHORT attribute:

- If the *psect-name* in a SHORT attribute is not yet declared, then its appearance in a SHORT attribute constitutes a declaration. The attributes of the PSECT containing the SHORT attribute become the attributes of the PSECT named in the SHORT attribute. An exception is when the PSECT name declared in the SHORT attribute does not have the SHORT attribute, and the PSECT name declared in the SHORT attribute does have the GP_RELATIVE attribute.
- If the *psect-name* in a SHORT attribute has previously been declared, then its attributes are not changed. A warning message is generated if the PSECT named in a SHORT attribute does not have the GP_RELATIVE attribute.
- If a data object with storage class OWN, GLOBAL, or PLIT has a size of 8 or fewer bytes, and the data object is specified to be allocated to a PSECT that includes the SHORT attribute, then that object is allocated to the PSECT named in the SHORT attribute. Note that this is a one-step process that is not recursive. If a short data object has its allocation PSECT renamed by the SHORT attribute, then the SHORT attribute of the renamed PSECT is not considered for any further renaming.

- Data objects with sizes larger then 8 bytes ignore the SHORT attribute.
- Data objects in the CODE, INITIAL, and LINKAGE storage classes ignore the SHORT attribute, regardless of their size.
- For the purposes of PSECT renaming by means of the SHORT attribute, the size of a PLIT object does not include the size of the count word that precedes the PLIT data.

Example:

```
PSECT

  NODEFAULT = $GLOBAL_SHORT$
    (READ,WRITE,NOEXECUTE,NOSHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3),
    GP_RELATIVE),

! The above declaration of $GLOBAL_SHORT$ is not needed.  If the above
! declaration were deleted, then the SHORT($GLOBAL_SHORT$) attribute in
! the following declaration would implicitly make an identical
! declaration of $GLOBAL_SHORT$.

  GLOBAL = $GLOBAL$
    (READ,WRITE,NOEXECUTE,NOSHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3),
    SHORT($GLOBAL_SHORT$)),

  NODEFAULT = MY_GLOBAL
    (READ,WRITE,NOEXECUTE,SHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3)),

  PLIT = $PLIT$
    (READ,NOWRITE,NOEXECUTE,SHARE,NOPIC,CONCATENATE,GLOBAL,ALIGN(3),
    SHORT($PLIT_SHORT$));

GLOBAL

        X1,                     ! allocated in $GLOBAL_SHORT$
        Y1 : VECTOR[2,LONG],    ! allocated in $GLOBAL_SHORT$
        Z1 : VECTOR[3,LONG],    ! allocated in $GLOBAL$
        A1 : PSECT(MY_GLOBAL),  ! allocated in MY_GLOBAL
        B1 : VECTOR[3,LONG] PSECT(MY_GLOBAL), ! allocated in MY_GLOBAL
        C1 : VECTOR[3,LONG]
                PSECT($GLOBAL_SHORT$); ! allocated in $GLOBAL_SHORT$

PSECT GLOBAL = MY_GLOBAL;
! use MY_GLOBAL as default for both noshort/short

GLOBAL
        X2,                     ! allocated in MY_GLOBAL
        Y2 : VECTOR[2,LONG],    ! allocated in MY_GLOBAL
        Z2 : VECTOR[3,LONG],    ! allocated in MY_GLOBAL
        A2 : PSECT($GLOBAL$),   ! allocated in $GLOBAL_SHORT$
        B2 : VECTOR[3,LONG] PSECT($GLOBAL$); ! allocated in $GLOBAL$;

! Note that the allocations of A1, X2, and Y2 violate the calling
! standard rules.  These variables cannot be shared with other
! languages, such as C or C++.

PSECT GLOBAL = $GLOBAL$;
! back to using $GLOBAL$/$GLOBAL_SHORT$ as default noshort/short
```

```
GLOBAL BIND
        P1 = UPLIT("abcdefghi"),        ! allocated in $PLIT$
        P2 = PLIT("abcdefgh"),          ! allocated in $PLIT_SHORT$
        P3 = PSECT(GLOBAL) PLIT("AB"),  ! allocated in $GLOBAL_SHORT$
        p4 = PSECT($PLIT_SHORT$)
                PLIT("abcdefghijklmn"), ! allocated in $PLIT_SHORT$
        P5 = PSECT(MY_GLOBAL) PLIT("AB"); ! allocated in MY_GLOBAL

! Note that the allocations of A1, X2, Y2, and P5 violate the calling
! standard rules.  These variables cannot be shared with other
! languages, such as C or C++.  They can be shared with modules
! written in BLISS and MACRO.
```

Note

OpenVMS IA-64 BLISS does not support using GP_RELATIVE addressing mode on EXTERNAL variable references. However, the usual GENERAL addressing mode used by EXTERNAL variables will correctly reference a GP_RELATIVE section. There are no plans to add an ADDRESSING_MODE(GP_RELATIVE) attribute to BLISS.

1.5. BLISS Differences Between IA-64 BLISS and x86-64 BLISS

This section describes those BLISS features that will not be supported by OpenVMS x86-64 BLISS.

1.5.1. Floating Point Register Names

The Alpha/IA-64 floating point register names are not supported for naming in REGISTER, GLOBAL REGISTER, or EXTERNAL REGISTER, or as parameters to LINKAGE declarations.

1.5.2. RETURNADDRESS Built-in with BLISS-32

The **RETURNADDRESS** built-in function returns the PC of the caller's caller.

The linker on OpenVMS x86-64 will place code in 64-bit address space by default. BLISS-32 will only return the bottom 32-bits of the PC of the caller's caller. There is no ability to obtain the full 64-bit return address from BLISS-32.

1.5.3. BLI$CALLG Removed

The VAX idiom CALLG(.AP, ...) was replaced by an assembly routine BLI$CALLG(ARGPTR(), .RTN) for OpenVMS Alpha and OpenVMS IA-64.

The BLI$CALLG routine that was present on OpenVMS Alpha and OpenVMS IA-64 has been obsoleted for OpenVMS x86-64. Programs will just use the LIB$CALLG Run-Time Library function, which performs an identical function.

1.5.4. ALPHA_REGISTER_MAPPING SWITCH and Qualifier

The module level switch ALPHA_REGISTER_MAPPING as well as the **/ALPHA_REGISTERMAPPING** DCL qualifier are essentially ignored. All register names correspond to their Alpha counterparts at all times.

1.5.5. Unsupported and Ignored DCL Qualifiers

The **/ANNOTATIONS** qualifier, which provides information about optimizations applied to the program on Alpha and IA-64, is ignored on x86-64.

The **/GRANULARITY** qualifier is ignored for x86-64. The x86-64 architecture is a byte-granular architecture, and the code generator will automatically adjust as needed for better performance.

The **/MACHINE_CODE** qualifier is ignored on x86-64. Use **ANALYZE/OBJECT/DISASSEMBLE** as an alternative. An improved machine code listing will be provided in a future release.

The **/TIE** qualifier is ignored for x86-64, since there is no binary translator available on OpenVMS x86-64.

1.5.6. Built-ins Supported from OpenVMS Alpha and OpenVMS IA-64 Systems

1.5.6.1. SETREG and GETREG Built-ins

The OpenVMS IA-64 BLISS compiler provided built-in functions (**GETREG**, **SETREG**, **GETREGIND**, and **SETREGIND**) for read and write access to the many and varied processor registers. The built-ins are supported on OpenVMS x86-64 for a limited number of x86-64 general registers and processor registers. See STARLET.REQ (specifically symbols prefixed with X86REG$) for the short list of registers.

1.5.6.2. Built-ins from OpenVMS IA-64

A small number of built-ins that were added for OpenVMS IA-64 BLISS are supported on OpenVMS x86-64 BLISS:

```
BREAK
BREAK2
```

1.5.6.3. Built-ins from OpenVMS Alpha

The following built-ins from OpenVMS Alpha BLISS are supported on OpenVMS x86-64 BLISS:

```
BARRIER
ESTABLISH
REVERT

ROT
SLL
SRA
SRL
UMULH

ADAWI

ADD_ATOMIC_LONG   AND_ATOMIC_LONG  OR_ATOMIC_LONG
ADD_ATOMIC_QUAD   AND_ATOMIC_QUAD  OR_ATOMIC_QUAD

CMP_SWAP_LONG CMP_SWAP_QUAD

TESTBITSSI TESTBITCC  TESTBITCS
TESTBITCCI TESTBITSS  TESTBITSC
```

```
ADDD    DIVD    MULD    SUBD    CMPD
ADDF    DIVF    MULF    SUBF    CMPF
ADDG    DIVG    MULG    SUBG    CMPG
ADDS    DIVS    MULS    SUBS    CMPS
ADDT    DIVT    MULT    SUBT    CMPT

CVTDF   CVTFD   CVTGD   CVTSF   CVTTD
CVTDG   CVTFG   CVTGF   CVTSI   CVTTG
CVTDI   CVTFI   CVTGI   CVTSL   CVTTI
CVTDL   CVTFL   CVTGL   CVTSQ   CVTTL
CVTDQ   CVTFQ   CVTGQ   CVTST   CVTTQ
CVTDT   CVTFS   CVTGT           CVTTS

CVTID   CVTLD   CVTQD
CVTIF   CVTLF   CVTQF
CVTIG   CVTLG   CVTQG
CVTIS   CVTLS   CVTQS
CVTIT   CVTLT   CVTQT

CVTRDL  CVTRDQ
CVTRFL  CVTRFQ
CVTRGL  CVTRGQ
CVTRSL  CVTRSQ
CVTRTL  CVTRTQ

PAL_INSQHIL     PAL_REMQHIL
PAL_INSQHILR    PAL_REMQHILR
PAL_INSQHIQ     PAL_REMQHIQ
PAL_INSQHIQR    PAL_REMQHIQR
PAL_INSQTIL     PAL_REMQTIL
PAL_INSQTILR    PAL_REMQTILR
PAL_INSQTIQ     PAL_REMQTIQ
PAL_INSQTIQR    PAL_REMQTIQR
PAL_INSQUEL     PAL_REMQUEL
PAL_INSQUEL_D   PAL_REMQUEL_D
PAL_INSQUEQ     PAL_REMQUEQ
PAL_INSQUEQ_D   PAL_REMQUEQ_D
```

The **TESTBITxx** instructions do not support the optional *retry-count* input argument or the optional *success-flag* output argument from Alpha.

The **XXX_ATOMIC_XXX** do not support the optional *retry-count* input argument from Alpha. These built-ins now have the form:

```
<op>_ATOMIC_<size>(ptr, expr
                     [;old_value] ) !Optional output

   Value:   1 Operation succeeded
            0 Operation failed

  <op> is one of AND, ADD OR
  <size> is one of LONG or QUAD
```

The operation is addition (or ANDing or ORing) of the expression *expr* to the data-segment pointed to by *ptr* in an atomic fashion.

The value of *ptr* must be a naturally-aligned address.

The optional output parameter *old_value* is set to the previous value of the data-segment pointed to by PTR.

Any attempt to use the OpenVMS Alpha BLISS optional *retry_count* argument will result in a syntax error.

The **CMP_SWAP** built-ins have the form:

```
CMP_SWAP_<op>(addr, comparand, value)
```

These functions do the following interlocked operations: compare the longword or quadword at *addr* with *comparand*, and if they are equal, store *value* at *addr*. They return an indicator of success (1) or failure (0).

1.5.7. New and Expanded Lexical Functions

BLISS will add new compiler-state lexical functions to support the OpenVMS x86-64 compilers: BLISS32X and BLISS64X.

- %BLISS will now recognize BLISS32E, BLISS64E, BLISS32V, BLISS32I, BLISS64I, BLISS32X, and BLISS64X.

  %BLISS(BLISS32) is true for all 32-bit BLISS compilers.

  %BLISS(BLISS32V) is true only for VAX BLISS (BLISS-32).

  %BLISS(BLISS32E) is true for all 32-bit Alpha compilers.

  %BLISS(BLISS64E) is true for all 64-bit Alpha compilers.

  %BLISS(BLISS32I) is true for all 32-bit IA-64 compilers.

  %BLISS(BLISS64I) is true for all 64-bit IA-64 compilers.

  %BLISS(BLISS32X) is true for all 32-bit x86-64 compilers.

  %BLISS(BLISS64X) is true for all 64-bit x86-64 compilers.

- The lexical functions %BLISS32X and %BLISS64X have been added. Their behavior will parallel that of the new parameters to %BLISS.

- Support for the x86-64 architecture (with the keywords x86_64 and x86) has been added to the %HOST and %TARGET lexical functions for OpenVMS x86-64 BLISS.

1.6. Floating Point Support

1.6.1. Floating Point Built-in Functions

BLISS does not have a high level of support for floating-point numbers. The extent of the support involves the ability to create floating-point literals, and there are machine-specific built-ins for floating-point arithmetic and conversion operations.

1.6.2. Floating Point Literals

The floating point literals supported by OpenVMS x86-64 BLISS is the same set supported by OpenVMS Alpha and OpenVMS IA-64 BLISS: %E, %D, %G, %S and %T.

1.6.3. Floating Point Registers

Direct use of the x86-64 floating-point registers is not supported.

1.6.4. Calling Non-BLISS Routines with Floating Point Parameters

It is possible to call standard non-BLISS routines that expect floating-point parameters passed by value, and that return a floating-point or complex value.

The standard functions %FFLOAT, %DFLOAT, %GFLOAT, %SFLOAT, and %TFLOAT will be supported by OpenVMS x86-64 BLISS.

1.7. Documentation

Documentation available for OpenVMS x86-64 BLISS consists of the following:

- VAX BLISS-32 Language Reference Manual (SYS$HELP:BLISSREFMANUAL.PDF)
- VAX BLISS-32 Language User Manual (SYS$HELP:BLISSUSERMANUAL.PDF)
- These release notes.

We are considering a future update/addendum to the manual to incorporate all the information from these release notes.

1.8. Debugging

All programs that will be used with a debugger should be compiled with the **/NOOPTIMIZE/DEBUG** qualifiers. Debugging optimized code is very difficult. Compilation with normal (full) optimization will have these noticeable effects:

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

Neither of these problems will occur in modules compiled with **/NOOPTIMIZE**.

Debugging on OpenVMS x86-64 is currently limited due to missing support in the LLVM backend for BLISS-specific DWARF records. This will be improved in future versions of BLISS.

1.9. Building the STARLET and LIB .L32 and .L64 Libraries

The STARLET and LIB pre-compiled header files are shipped as part of the OpenVMS x86-64 kit. If you need to rebuild them yourself, below are the commands required to do so:

```
$ BLISS/X32/TERMINAL=NOERRORS/LIB=SYS$COMMON:[SYSLIB]STARLET.L32 SYS$LIBRARY:STARLET.REQ
$ BLISS/X32/TERMINAL=NOERRORS -
  /LIB=SYS$COMMON:[SYSLIB]LIB.L32 SYS$LIBRARY:STARLET.REQ+SYS$LIBRARY:LIB.REQ
$ BLISS/X64/TERMINAL=NOERRORS/LIB=SYS$COMMON:[SYSLIB]STARLET.L64 SYS$LIBRARY:STARLET.R64
$ BLISS/X64/TERMINAL=NOERRORS/ASSUME=NOQUAD_LITERAL -
```

```
/LIB=SYS$COMMON:[SYSLIB]LIB.L64 SYS$LIBRARY:STARLET.R64+SYS$LIBRARY:LIB.R64
```

You will need to be logged into an account with system privileges to successfully write the files to SYS$COMMON:[SYSLIB].

2. Maintenance Corrections for OpenVMS x86-64 BLISS

The following bugs have been fixed since the V1.13-136 release.

- Initializing a bitvector would incorrectly overwrite adjacent variables. For example:

  ```
  global b;
  local a : initial(.b) bitvector[8];
  ```

  This would incorrectly write 32-bits to variable A, which is only 8-bits big.

- A field-selected store with size of zero would incorrectly write into the destination. For example:

  ```
  own data : initial(-1);

  routine write(dptr, offset, s) : novalue =
    begin (.dptr)<.offset,.s,0> = 0; end;

  write(data,0,0);
  ```

  This would incorrectly write the bottom byte of 'data'.

- The alignment of EXTERNAL LITERALs was incorrectly computed, and the optimizer would incorrectly believe that different literals were the same value.

- The WEAK attribute did not work with EXTERNAL ROUTINE. It would generate a non-WEAK reference instead.

- LOCAL variables in nested scopes in a routine would get incorrect debug information and sometimes cause a compiler assertion.

- The compiler would get a G2L assertion if a routine and variable had the same name.

- The **ACTUALCOUNT** built-in would sometimes return the wrong number of arguments when the optimizer was enabled.

- The compiler would generate incorrect calling sequences when calling a routine inside the same module with a different number of arguments.

- The optimizer would perform a "tail-call" optimization where it would replace the final "callq" instruction with a "jmp" instruction. This transformation is not allowed by the OpenVMS Calling Standard and results in the exception handling stack-walking code to return prematurely.

- Overlay PSECTs would be generated with extra padding at the end and be the wrong size.

- The compiler would ACCVIO if **/DEBUG** was used with some LINKAGE declarations that references Alpha registers.

- Improved code for various VAX floating built-ins. The prior compiler would sometimes convert the VAX floating to IEEE floating then immediately convert it back to VAX floating.

- The compiler now uses 64-bit heap for much of the optimizer and code-generator. This allows much larger programs to be compiled.
- The BLISS-64 compiler would generate incorrect code for storing a 64-bit value to a 32-bit LOCAL variable using the INITIAL clause.
- BLISS would incorrectly write data to memory for a field-selector with a size of 0. For example:

```
routine write(dptr, offset, s) : novalue =
begin
(.dptr)<.offset,.s,0> = 0;
end;

write(data,0,0);
```

  This would incorrectly overwrite bits in 'data'.
- Fixed various BIND and PRESET issues with stores of different sizes than the source.
- Improved compile-time performance for writing large static variables that are initialized.

3. Known Bugs and Deficiencies

This section describes known bugs and deficiencies in the x86-64 BLISS compiler.

1. Due to a design flaw in structure definitions, compiler errors can occur when the first occurrence of a structure formal is within a conditional branch.

   Example:

   ```
   STRUCTURE BAD[I,P,S]= [%UPVAL]
       (IF .I THEN BAD ELSE BAD + .BAD<16,16>)<P,S>;
   ```

   BLISS semantics guarantee that a structure actual-parameter is evaluated only once. This is implemented by treating the first occurrence of a structure formal as if it were a **BIND** declaration. The other occurrences of the structure formal are then treated as if they were uses of the "imaginary" bind-name. This choice of implementation fails when the first occurrence of the structure formal is in conditional flow. The problem can be avoided by ensuring that the first occurrence of each formal is outside of conditional flow. The example structure should be written as:

   ```
   STRUCTURE GOOD[I,P,S]= [%UPVAL]
       (GOOD; IF .I THEN GOOD ELSE GOOD + .GOOD<16,16>)<P,S>;
   ```

   Note that the "structure-name" is the zeroth structure formal parameter. The formals "I", "P", and "S" are already outside of conditional flow, so they are processed correctly. This change will cause the compiler to use slightly more memory, but the resulting code will be correct. There should also be no reduction in optimization.

   No problems will occur when the conditional flow is constant folded at compile time, or when there is no conditional flow in the structure body.