



Migrating Existing C and C++ Applications to Monterey IA-64

May 1999



Migrating Existing C and C++ Applications to Monterey IA-64

**Ahmed C. Chibib, IBM STSM
RS/6000 Partners in Development
IBM Server Group
Austin, Texas**

May 1999

Special Notices

This publication/presentation was produced in the United States. IBM may not offer the products, programs, services or features discussed herein in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the products, programs, services, and features available in your area. Any reference to an IBM product, program, service or feature is not intended to state or imply that only IBM's product, program, service or feature may be used. Any functionally equivalent product, program, service or feature that does not infringe on any of IBM's intellectual property rights may be used instead of the IBM product, program, service or feature.

Information in this presentation concerning non-IBM products was obtained from the suppliers of these products, published announcement material or other publicly available sources. Sources for non-IBM list prices and performance numbers are taken from publicly available information including D.H. Brown, vendor announcements, vendor WWW Home Pages, SPEC Home Page, GPC (Graphics Processing Council) Home Page and TPC (Transaction Processing Performance Council) Home Page. IBM has not tested these products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM may have patents or pending patent applications covering subject matter in this presentation. The furnishing of this presentation does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of a specific Statement of General Direction.

The information contained in this presentation has not been submitted to any formal IBM test and is distributed "AS IS". While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. The use of this information or the implementation of any techniques described herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The information contained in this document represents the current views of IBM on the issues discussed as of the date of publication. IBM cannot guarantee the accuracy of any information presented after the date of publication. All prices shown are IBM's suggested list prices; dealer prices may vary. IBM products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Any performance data contained in this document was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements quoted in this presentation may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this presentation may have been estimated through extrapolation. Actual results may vary. Users of this presentation should verify the applicable data for their specific environment.

The following terms are registered trademarks of International Business Machines Corporation in the United States and/or other countries: AIX, AIX/6000, C Set++, CICS, CICS/6000, DB2, ESCON, IBM, LANStream, LoadLeveler, Magstar, MediaStream, Micro Channel, MQSeries, Netfinity, Parallel Sysplex, RS/6000, S/390, Service Director, ThinkPad, TURBOWAYS, VisualAge. The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries: AIX PVMe, DB2 Universal Database, Deep Blue, e-business (logo), HACMP/6000, Intelligent Miner, Intellistation, Network Station, POWER2 Architecture, PowerPC (logo), PowerPC 604, SP. A full list of U.S. trademarks owned by IBM may be found at www.ibm.com/legal/copy/trade.html.

Microsoft, Windows, Windows NT and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries or both. UNIX is a registered trademark of The Open Group. Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and other countries. Lotus, Lotus Domino and Lotus Notes are trademarks or registered trademarks of Lotus Development Corporation. Tivoli, TME, TME 10 and TME 10 Global Enterprise Manager are trademarks or registered trademarks of Tivoli Systems, Inc. Other company, product and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

1.0 Introduction	2
1.1 Acknowledgment	2
1.2 Definitions	3
2.0 Migrating AIX Applications to Monterey IA-64	4
2.1 Missing Functionality	4
2.2 Module Format	5
2.3 Alignment Differences	5
2.4 Default for char	5
2.5 Floating Point Support	6
2.6 Libraries and Archives	6
2.5 Linking and ld flags	7
3.0 Porting AIX 32-bit Device Drivers to Monterey	10
3.1 IFDEFS	10
3.2 Config Methods	10
3.3 Memory Mapped I/O	11
4.0 Converting 32-bit Applications to 64-bit	27
4.1 Overview	27
4.2 C and C++ Data Type Size and Alignment Issues	28
4.3 <code>sizeof(int) != sizeof(long)</code>	29
4.4 Truncation of 64 bit value when assigned to a smaller type	29
4.5 Explicit cast improperly applied	30
4.6 A pointer to an int is not compatible with a pointer to a long	31
4.7 Lack of prototyped function declarations in scope of call	32
4.8 Integer expression with potential overflow is converted to a long	35
4.9 Untyped integral constants are int by default	36
4.10 <code>sizeof(void *) != sizeof(int)</code>	39
4.11 Truncation of a 64 bit pointer value when converted to a smaller	39
4.12 Assumption that pointers and int are same size in arithmetic	40
4.13 Pointer return type or argument types in the absence of a prototyped function declaration	41
4.14 Objects Change Size	44
4.15 Integer Constants	45
4.16 Stack layout changes due to larger data elements	46
5.0 Byte Ordering	48
6.0 Dealing with Endianness in Porting	52
6.1 Data Type Mismatch	52
6.2 Overlaid Data (with Bit Fields)	54
6.3 Endianness in Exchanging or Sharing Data	55
7.0 References	59

1.0 Introduction

It is the intent of the Monterey partners to facilitate an easy migration of applications from AIX V4.3 on PowerPC and Unixware 7 on x86 to Monterey IA-64. In most cases, migration will require nothing more than a simple recompile; however, there are some exceptions. This document covers the various migration scenarios and those instances that require changes to the application source and/or to the way the application is built.

In Section 2 of this document, potential migration issues are highlighted by comparing the LE and BE data storage models. The significance of endian dependency in referencing data is exemplified with data structure and code instance examples. In Section 3, we show you how to look for endian-dependent code and how to modify the code to make it endian-neutral or endian-aware. In Section 4, we offer guidelines and recommendations for dealing with endianness in porting applications.

Note: AIX was designed for Power/PowerPC architecture, which supports the Big-Endian (BE) data storage model. And, while IA-64 and Merced support both little- and big-endian, the legacy IA applications and platforms/environments support only little-endian (LE). To support legacy IA applications and be compatible with other IA platforms, Monterey IA-64 supports LE.

1.1 Acknowledgment

I would like to thank the following persons for taking the time to review the paper and provide valuable feedback.

Casey Cannon
Thomas Cheng
Bill Carter
Mike Day
Mike Lyons
Hye-Young McCreary
David Mehaffy
Paul Milot
Scott Porter
Randy Swanberg
Don Wood

1.2 Definitions

IA-64: 64-bit Intel Architecture

ILP32: A 32-bit application source that has been compiled to run natively on IA-64. In this model, the size of **int**, **long**, and **pointer** for C and C++ is 32-bit.

LP64: A 64-bit application source that has been compiled to run natively on IA-64. In this model, the size of **int** for C and C++ is 32 bit, while the size of **long**, and **pointer** is 64-bit.

IA-32: A 32-bit application source that has been compiled to run on 32-bit Intel architecture such as a Pentium.

EM-32: The native 32 bit mode of the IA-64 architecture.

EM_64: The native 64-bit mode of the IA-64 architecture.

Merced: First Intel implementation of the IA-64 Intel architecture. This processor supports IA-32, EM-32 (ILP32) and EM-64 (LP64) binaries

Monterey IA-64: The IBM/SCO/Sequent UNIX operating system that will run on Merced

Little Endian (LE): The right to left assignment of addresses to bytes in a data word.

Big Endian (BE): The left to right assignment of addresses to bytes in a data word.

Endian Sensitive Application: An application source that depends on the byte ordering

Endian Neutral Application: An application source that has no dependency on byte ordering

2.0 Migrating AIX Applications to Monterey IA-64

AIX on the PowerPC addresses data in Big Endian (BE) order, while traditional Intel processors address data in Little Endian (LE) order. In porting Monterey IA-64 to Merced, we chose to maintain the Intel LE byte ordering. As a result, application source that is dependent on byte ordering and runs successfully on AIX/PowerPC may not run successfully on Monterey IA-64 without modifications to eliminate that dependency.

Note that, in general, software developers have a single source of their application that runs on AIX and other LE systems, such as NT, ensuring that their source is Endian-neutral. But when this is not the case, changes to the source may be required. For dealing with Endian issues, see Section 3.0.

Potential migration issues to consider include the following.

2.1 Missing Functionality

Some functionality available on AIX on the PowerPC may not be supported on Monterey in its first release. An exhaustive list is not available at this time but, the following list is known to be missing from the first release:

2.1.1 3D Graphics Support

Monterey IA-64 based systems will initially be targeted for commercial use only. None of the 3D libraries (graPHIGS, OpenGL, ..) will be ported, therefore, there will be no 3D application support.

2.1.2 Fortran Support

The first release of Monterey will not have a Fortran compiler, as a result, applications that include Fortran source will not be able to port to Monterey.

2.1.3 High Availability Cluster Multi Processing (HACMP)

There is no HACMP support in the first release on Monterey.

2.1.4 DCE

There will be no support for DCE in the first release of Monterey. Applications that depend on the availability of DCE will not be supported.

2.1.5 SOM/DSOM

SOM and DSOM will not be supported on Monterey.

2.2 Module Format

- Monterey IA-64 supports the UNIX System V.4 ABI for shared libraries and linking and loading of applications. This differs from what is supported on AIX/PowerPC in the following way:

	AIX/PowerPC	Monterey IA-64
Module Format	XCOFF	ELF
Debugger Format	stab strings	DWARF2

The linking and loading, dynamic loading, shared library creation on Monterey IA-64 will be the same as it is on UNIX System V.4. Import/Export files will no longer be needed and many of the AIX linker flags will become obsolete. This may impact “makefiles.”

2.3 Alignment Differences

- “long long” is used to define an 8-byte integer in 32-bit applications. “long long” on AIX is aligned at an 8-byte boundary; however, “long long” is aligned at a 4-byte boundary on ILP32. This causes a structure containing “long long” to have different size and mapping characteristics on AIX/PowerPC than on Monterey IA-64. For example:

	AIX/PowerPC	ILP-32
struct {		
int i;	# size 4; alignment 4	size 4; alignment 4
long long j;	# size 8; alignment 8	size 8; alignment 4
long k;	# size 4; alignment 4	size 4; alignment 4
long double m;	# size 16; alignment 16	size 16; alignment 4
}		

AIX/PowerPC requires an 8-byte alignment for “long long”, and 16 byte alignment for “long double”, therefore, on the PowerPC, 4 bytes of padding is introduced between the “int i” and the “long long j” and 12 of bytes padding before the “long double m” to force the proper alignment. This is not required for ILP32. As a result, the size of the structure and its mapping will differ depending on the platform.

2.4 Default for *char*

- The default compilation mode for *char* on AIX/PPC is *unsigned*, while on Monterey it is *signed*. AIX applications that need the default behavior for *char* on Merced to be *unsigned* can use the ***-qchar=unsigned*** compile-time flag to accomplish the same desired effect.

2.5 Floating Point Support

Some floating point APIs which are supported by the AIX kernel will not be supported on Monterey. These are:

fp_trapstate (which was/is the replacement for fp_cpusync)
fp_fpscrx (which modifies the in-memory extension to the fpscr)

2.6 Libraries and Archives

- On AIX/PowerPC, an archive may contain different types of objects. For example, *libc.a* contains *.o* files, 32-bit shared libraries and 64-bit shared libraries. Applications (both 32-bit and 64-bit) can link dynamically and/or statically against this single archive (*libc.a*). This behavior is not duplicated on Monterey IA-64. The System V.4 semantics for shared library creation, static and dynamic linking doesn't allow for a single archive to be used in the same manner as on AIX/PowerPC. For Monterey IA-64, library variants will exist in separate directories as follows:

/usr/lib/ia32	(IA32 little-endian libraries)
/usr/lib/ia64l32	(IA64 ilp32 little-endian libraries)
/usr/lib/ia64l64	(IA64 lp64 little-endian libraries)

Mixed mode linking is disallowed. How the linker determines its target mode (ia32, ia64l32, ia64l64) is left to the implementation.

When a directory specifier is provided, such as 'ld ... -Ldir ...', the linker will add that directory to its search list before the default directories to be searched (as it always has).

When the run-time linker and link-editor are searching for libraries it is not a fatal error if a ELF file of the wrong type is encountered in the search. Instead the link-editors should exhaust the search of all paths before determining a matching object could not be found. This will permit having a common search path (LD_LIBRARY_PATH) which contains a mix of directories containing differing process models.

2.5 Linking and *ld* flags

The AIX linker differs from the SVR4 linker (used in Monterey IA64) in features and functionality. The flags used as well as their default behavior is often different. The Monterey linker **will not support relinking of an executable** and will ignore the following AIX flags since they either have no meaning in an ELF and IA64 environment or they have no equivalent:

-k -z -basis -bautoexp -bbigtoc -bbindcmds:File -bbinder:File -bbindopts:File
-bcomprld -bcrlld -bcror15 -bcror31 -bD:Number -bdbg:Option -bdebugopt:Option
-bdelcsect -berrmsg -bex1:File -bex2:File -bex3:File -bex4:File -bex5:File -bgc
-bgcbypass:Number -bglink:File -bh:Number -bhalt:Number -bI:File -nimport:File
-binitfini:Init:Term:Priority -bipath -bkeepfile:File -blazy -bL:File -bloadmap:File
-bM:ModuleType -bmodtype:ModuleType -bmaxdata:Number -bmaxstack:Number
-bS:Number -bnl -bnoloadmap -bnoautoimp -bnobigtoc -bnobind -bnocomprld -bnocrlld
-bnodelcsect -bnoentry -bnoerrmsg -bnogc -bnoglink -bnoipath -bnolibpath -bnom
-bnoobjreorder -bnop:Nop -bnoquite -bnoreorder -bnortl -bnortllib -bnostrip -bnosymbolic
-bnotextro -bnro -bnotypchk -bnov -bnox -bquiet -br -breorder -brename:Old -brtl -brtllib
-bS:number -bsmap -bstabcmpct -bsxref:File -btypchk -bx -bX:File -bxref:File -m -M
-SNumber -uName -v -zString

There are other AIX linker flags for which there are corresponding Monterey linker flags that provide either identical or similar functionality. Following are the AIX linker flags and their semantics with the corresponding Monterey linker flags and semantics:

AIX Linker Flag	Monterey Equivalent Flag
-DNumber locates the initialized data	Possible with the Map File (-M option)
-eLabel sets the entry point to Label	-e <i>epsym</i> sets the entry point to epsym
-G produces a shared object	-G produces a shared object
-HNumber aligns the text, data, and loader sections at a multiple of number	Possible with the Map File (-M option)
-IName processes libName.a in static mode	-l x
-LDir adds Dir to the list of search directories	-L <i>path</i>
All -L options are processed first before any of the -l options, therefore, each library specified with the -l option will be searched in all directories specified	Each directory specified with -l option is searched in the directory specified with -L only if -L precedes the -l option
LDPATH = dirlist Libraries are searched in the following order: 1) directories specified by -L or LIBPATH if there is no -L option 2) Standard directories (/usr/lib and /lib)	LD_LIBRARY_PATH = dir1:dir2; dir3:dir4 Libraries are searched in the following order: 1) dir1 and dir2 2) directories specified by -L option 3) dir3 dir4 4) standard directories (/usr/lib & /usr/ccs/lib)
-oName names the output file	-o outfile names the output file
-r produces non executable output file suitable for another linking	-r produces a relocatable file (partially linked file)
-s strips symbolic debugging information	-s strips symbolic debugging information
-TNumber sets the starting address of the text section. Doesn't have any effect on run-time addresses	Possible with the Map File (-M option)
-bautoimp or -bso imports symbols from any shared object specified as input. This option is valid for all shared objects	-Bdynamic link with shared object version of a library (when available). This option is valid only for the shared objects following it and until the next -Bstatic
-bC:File or -bcalls:File writes an address map to a file	-m produces a memory map of input/output sections
-bdynamic or -bshared processes subsequent shared objects in dynamic mode	-Bdynamic link with shared object version of a library (when available) until next -Bstatic
-bstatic processes subsequent shared objects in static mode	-Bstatic link with archive version of a library, until next -Bdynamic
-bE:File or -bexport:File exports the external symbols listed in File	-Bexport:File exports all global and weak symbols listed in File
-bernotok or -bf reports an error if there are	-z <i>defs</i> do not allow undefined symbols

any unresolved external references	(default for executables)
-berok allows unresolved external references in the output file	-z nodefs allows undefined symbols (default for shared objects)
-bexpall export all global symbols	-Bexport
-blibpath:Path uses Path when writing the loader section of the output file	-R <i>path</i> records path for run-time library search (this may not be equivalent)
-bnoexpall does not export any symbol not listed in the export file (default)	-Bexport:File hides all symbols except those listed in File
-bpD:Origin specifies origin as the address of the first byte of file page containing the beginning of the data section	Possible with the Map File (-M option)
-bpT:Origin specifies origin as the address of the first byte of the file page containing the beginning of the next section	Possible with the Map File (-M option)
-bro or -btextro ensures that there are no load-time relocations for the text section	-z <i>text</i> in dynamic mode only. Do not allow relocations against non-writable allocatable segment
-bsymbolic assigns the symbolic attribute to most symbols exported without an explicit attribute	-Bsymbolic=[list :File] bind all references to the named symbol to its definition
LIBPATH environment variable	-YP, dirlist changes default library search directories
-bnso -bI:/lib/syscalss.exp	-a produces a statically linked executable file (must not be used with -r and -dy options)

3.0 Porting AIX 32-bit Device Drivers to Monterey

3.1 IFDEFS

64-bit compiles of kernel extensions and device drivers will define `__64BIT_KERNEL` and `__64BIT__`. Therefore, for single source device drivers that generate a 32-bit device driver and a 64-bit device driver the differences can be controlled with these ifdefs.

3.2 Config Methods

Currently, the plan is to continue to have the existing 32-bit application configuration methods work as is and capable of loading the 32-bit or 64-bit version of the device driver. In general this means that device config methods will not be impacted by the 64-bit porting work. The only known issue is that with `dev_t` for passing devno's in the device's DDS. Ideally, since the device fileset will be impacted and shipping due to the porting of the device driver, it would be a good time to go ahead and make the config method change to handle 64-bit and 32-bit kernel devnos. At a minimum, the device's DDS structure will need to define a `dev32_t` type devno which will remain a 32-bit integer type.

The preferred solution will be for the method to define the devno in the device's DDS structure as a `dev64_t` which will handle either a 32-bit or 64-bit devno. The method would then have a run time check to determine whether the 32-bit or 64-bit kernel was active and use the appropriate `libcfg.a` services to generate the major and minor numbers. For example, the config method code might look like:

```
struct dds {
    .
    dev64_t devno;
    .
} dds;

if (__KERNEL_32()) {
    major = genmajor();
    minor = genminor(major);
    dds.devno = (dev64_t) makedev(major, minor);
} else {
    major = genmajor64();
    minor = genminor64(major);
    dds.devno = makedev64(major, minor);
}
```

(for more info, see the section on Devno).

All ODM information will stay the same, and likewise is not impacted. This is also true for the pathname of the device driver, as the 32-bit and 64-bit drivers will be packaged as a single archived object that contains both the 32-bit and 64-bit executables in a single file image with the singular device driver name (see Filesets/Packaging section).

3.3 Memory Mapped I/O

3.3.1 Address Space Management

The current 32-bit device driver model includes the **iomem_att()** and **iomem_det()** services for getting and releasing temporary addressability to I/O space. These interfaces will not be supported in the 64-bit kernel.

A new I/O mapping model is defined for the 64-bit kernel. This new model is designed to allow a device driver to map multiple discontinuous I/O regions into a single virtually attached and addressable segment. It also is structured to minimize the exposure of addressable I/O space to other errant kernel mode accesses and to allow "priming" of the translation hardware. Following is a description of the new model:

3.3.2 Initialize an I/O Mapping

```
io_handle_t io_map_init(struct io_map *, vpn_t, io_handle_t);
```

This service will create a segment to establish a cache-inhibited virtual to real translation for the bus address region defined by the contents of the **struct io_map**. The **flags** parameter of the **io_map** structure can be used to customize the mapping such as making the region read-only, using the **IOM_RDONLY** flag. This service can only be called from the process environment.

It returns a "handle" of an opaque type **io_handle_t** to be used on future map/unmap calls. If the segment can not be created or the mapping performed for any reason, NULL is returned.

The **vpn_t** type parameter represents the virtual page number offset to allow the caller to specify where in the virtual segment to map this region. If the offset conflicts with a previous mapping in the segment, NULL is returned. The caller would want to map the most frequently accessed and

performance critical I/O region at **vpn_t** offset 0 into the segment. This is due to the fact that the subsequent **io_map()** calls for this **io_handle_t** will return the effective address representing the start of the segment, i.e., page offset 0. The device driver is responsible for managing the various offsets into the segment. A single bus memory address page can be mapped multiple times at different **vpn_t** offsets within the segment if desired.

The **io_handle_t** parameter is passed when the caller wants to append a new mapping to an existing segment. For the initial creation of a new I/O segment, this parameter must be NULL. For appended mappings to the same segment, this parameter is the **io_handle_t** returned from

the last successfully `io_map_init()` call. If the mapping fails for any reason (offset conflicts with prior mapping, or no more room in the segment), `NULL` is returned. In this case, the previous `io_handle_t` as passed in is still valid. If successful, the `io_handle_t` returned should be used on all future calls.

This allows device drivers that must manage multiple address spaces on a single adapter to map them into a single virtual address region, requiring the driver to only do a single attach, `io_map()`, to gain addressability to all the mappings.

For example, the following creates an I/O segment and maps 3 distinct regions into it, one read-write Bus I/O space region, one read-write Bus Memory region, and one read-only Bus Memory region:

```
struct io_map iom;
io_handle_t ioh;
io_handle_t aioh;

/*
 * Initialize mapping for I/O region
 */
iom.bid = BID_ALTREG(adapter->bid, PCI_IOMEM);
iom.key = IO_MEM_MAP;
iom.flags = 0;
iom.busaddr = adapter->io_address;
iom.size =  PAGESIZE;

ioh = io_map_init(&iom, 0, NULL);
if (ioh == NULL)
    panic("Call Bruce Mealey....");

/*
 * Initialize mapping for Bus Memory region 1
 */
iom.bid = BID_ALTREG(adapter->bid, PCI_BUSMEM);
iom.flags = 0;
iom.busaddr = adapter->bus_address_1;
iom.size =  ONE_MB;

aioh = io_map_init(&iom, PAGESIZE/PAGESIZE, ioh);

if (aioh == NULL)
    panic("Call Bruce Mealey....");

/*
```

```

* Initialize mapping for Bus Memory region 2
*/

iom.busaddr = adapter->bus_address_2;
iom.size = SIXTEEN_MB;
iom.flags = IOM_RDONLY;

ioh = io_map_init(&iom, (PAGESIZE + ONE_MB)/PAGESIZE, aioh);

if (ioh == NULL)
    panic("Call Bruce Mealey....");

```

3.3.3 Attach to an I/O Mapping

```
void * io_map(io_handle_t);
```

This service sets up addressability to the I/O address space defined by the **io_handle_t** structure. It returns an effective address representing the start of the mapped region.

This is basically the replacement call for **iomem_att()**, however note that it might possibly replace multiple **iomem_att()** calls depending on the device and driver and whether multiple regions were mapped into this single virtual segment. Also like **iomem_att**, this service does not return any sort of failure, if something goes wrong the system will crash.

It is important to note a major difference from **iomem_att()**. **iomem_att()** took an **io_map** structure containing a bus address and returned a fully qualified effective address with any byte offset from the bus address preserved and computed into the returned effective address. **io_map()** will always return a segment aligned effective address representing the beginning of the I/O segment corresponding to **io_handle_t**. The manipulation of page and byte offsets within the segment are the responsibility of the device driver.

Consider the following comparison between the old model and the new:

```

iom.bid = BID_ALTREG(adapter->bid, PCI_BUSMEM);
iom.key = IO_MEM_MAP;
iom.flags = 0;
iom.busaddr = 0x1234888;
iom.size = PAGESIZE;

eaddr = iomem_att(&iom);

```

In the above example, the **eaddr** returned by **iomem_att()** would be something like **0x31234888**. Now consider the new model:

```

iom.bid = BID_ALTREG(adapter->bid, PCI_BUSMEM);
iom.key = IO_MEM_MAP;
iom.flags = 0;
iom.size = PAGESIZE;

iom.busaddr = 0x1234888;
ioh = io_map_init(&iom, 1, ioh);

eaddr = io_map(ioh);
eaddr += PAGESIZE * 1; /* add in virtual page offset */
eaddr += 0x888;      /* add in byte offset      */

```

In the above example, the `eaddr` returned by `io_map()` would be something like **0x3000000**. The driver had to add in the virtual page offset (0x1000) that it previously assigned this region on the `io_map_init()` call, and add in the byte offset of the actual target bus address.

This service is also subject to the same nesting rules regarding the number of attaches allowed. The total system number of active temporary attaches is 4. It is recommended that no more than 1 active attach be owned by a driver calling Interrupt or DMA kernel services. It is also recommended that no active attaches be owned by a driver when calling other kernel services.

This service is callable from both the process and interrupt environments.

3.3.4 Detach from an I/O Mapping

```
void      io_unmap(void * eaddr);
```

This service removes addressability to the I/O address space defined by the `eaddr` parameter. There must be a valid active mapping from a previous `io_map()` call for this effective address. Note that `eaddr` can be any valid effective address within the segment, and does not have to be exactly the same as the address returned by `io_map()`.

This is basically the replacement call for `iomem_det()`, however note that it might possibly replace multiple `iomem_det()` calls depending on the device and driver and whether multiple regions were mapped into this single virtual segment via `io_map_init()`.

This service is callable from both the process and interrupt environments.

3.3.5 Remove an I/O Mapping

```
void      io_map_clear(io_handle_t);
```

This service destroys all mappings defined by the `io_handle_t` parameter.

There should be no active mappings, i.e. outstanding `io_map()`'s, to this handle when `io_map_clear()` is called. The segment previously created by the `io_map_init()` call, or multiple `io_map_init()` calls, is deleted.

This service is only callable from the process environment.

Here are some example snippets of what a common source 32-bit and 64-bit driver might look like:

```
dd_open()

#ifdef __64BIT_KERNEL
    io_handle_t ioh;
#endif
    struct io_map iom;

    /*
     * Initialize an I/O MAP structure with adapters bus
     * memory region....this is works for 32-bit or 64-bit
     */
    iom.bid = BID_ALTREG(adapter->bid, PCI_BUSMEM);
    iom.key = IO_MEM_MAP;
    iom.flags = 0;
    iom.busaddr = ap->busmem_address_1;
    iom.size = ap->busmem_size_1;

#ifdef __64BIT_KERNEL
    /*
     * Create an I/O segment and map this bus address range
     * at Virtual page 0 into the segment.
     */
    ioh = io_map_init(&iom, 0, NULL);
    if (ioh == NULL)
        return(EINVAL);
#endif
}

dd_startio() and/or dd_intr() etc...
```

```

void * eaddr;

/*
 * Get addressability to busmem_address_1
 */
#ifdef __64BIT_KERNEL

/* Attach to my I/O segment and add in the byte offset to
 * my target bus address. Since I put this region at Virtual
 * Page 0 in my segment, I don't need to add in a page offset.
 */
eaddr = io_map(ioh) + PAGEOFFSET(ap->busmem_address_1);
#else
eaddr = iomem_att(&iom); /* Get addressability to busmem_address_1 */
#endif

/*
 * Remove addressability to busmem_address_1
 */

#ifdef __64BIT_KERNEL

io_unmap(eaddr);

#else
iomem_det(eaddr);

#endif

dd_close()

#ifdef __64BIT_KERNEL

/*
 * Remove my I/O segment
 */
io_map_clear(ioh);
#endif

```

3.3.6 REALMEM_BID

The **REALMEM_BID** will still be supported in the 64-bit environment, however, it will only be supported by **rmmmap_create()**. The new mapping service, **io_map_init()**, will not support **REALMEM_BID**. Specific functions that previously used **REALMEM_BID** to access system facility space not specifically associated with any I/O bus, such as the PAL or diagnostic tools, should convert to using the BID intended for that purpose, **SYSMEM_BID**.

3.3.7 Unaligned/String/Multiple Accesses to I/O

The **IA-64** architecture does not guarantee atomicity of access when performing unaligned, string, or multiple operations. There are 2 issues with this when dealing with these operations to I/O space. The first is that the CPU may break up unaligned accesses into a series of smaller operations (i.e., a non word aligned store may result in 4 separate byte stores). Depending on the device this could be a problem if it can not handle the broken up access or since the 4-byte data store is not atomic. The second issue deals with "stuttering", or re-issuing the same load or store operation. This could be caused by the CPU initiating a non-atomic operation (such as an unaligned, string, or multiple storage access), then being interrupted (by some non-maskable interrupt), and then restarting the entire non-atomic operation from the beginning. This becomes a problem for I/O space if the target of the load or store is not idempotent.

On our current systems, we are not subject to the latter of the two problems. If unaligned storage accesses are being performed to I/O space today, it most likely is being broken up into multiple smaller operations. However, with today's systems there isn't the concern for "stuttering" as there will be with future processors. Therefore, any device driver performing these types of accesses must be fixed.

Depending on the CPU, when it encounters an unaligned operation, it can either break up the operation itself or vector to the operating system's alignment fault handler to break up and perform the operation. The 64-bit Kernel alignment handler will recognize attempted unaligned accesses to I/O space and crash the system.

3.3.8 DMA

For the most part, for mainstream device drivers, there is no change with respect to DMA and the DMA services. The one noticeable exception is that the **busaddr** parameter to **D_MAP_PAGE()** and **D_UNMAP_PAGE()**, becomes a **ulong ***, instead of a **uint ***.

If there were any drivers using the **xmemdma()** service (which I don't think there are in PCI), they will come to find out that **xmemdma()** doesn't exist in the 64-bit kernel. **xmemdma64()** was the bridge replacement for **xmemdma()** in 4.3 and will now be the only one supported.

Finally, the interfaces designed for supporting DAC-capable PCI devices (i.e., 64-bit addressing), via the **DMA_ENABLE_64** flag to **D_MAP_INIT()** become defunct, since the general set of interfaces are now 64-bit capable. Note that the **DMA_ADDRESS_64_BIT** flag is still important and retains its meaning that the device can support generating a 64-bit Address.

3.3.9 Interrupts

There are no impacts or changes to the interrupt processing model. Note that in the 64-bit environment, any call to `i_init()` without the `INTR_MPSAFE` flag set will return a failure. (refer to the `MPSAFE` discussion under [Devswitch Interfaces](#))

3.3.10 EPOW

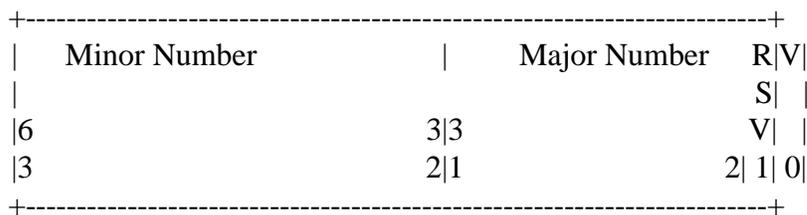
The EPOW registration and processing remains unchanged.

3.3.11 Devno

Currently, in existing systems, the `devno` of type `dev_t` is a 32-bit field. The low 16-bits represent the minor number and the high 16-bits represent the major number. As part of the 64-bit porting effort, we are laying the ground work for the kernel and kernel extensions to understand a 64-bit `devno`, and the `dev_t` type will become a long in the 64-bit environment. This will allow for future expansion to support more than 64K minor and 64K major numbers.

A problem with respect to this is there are existing application interfaces that understand data structures that contain a type `dev_t`. To a 32-bit application it is important that `dev_t` remain 32-bits. Therefore, in our first pass while enabling the 64-bit kernel and 64-bit kernel extensions to understand a 64-bit `dev_t`, interfaces that surface devno's as 32-bit quantities will need to convert from the 64-bit formatted devno to a 32-bit formatted devno. Macros will be provided for accomplishing this, although there is the fact that the most significant bits of the major and minor numbers will be lost. When we actually support major and minor numbers greater than 2^{16} , new API's that understand the larger `dev_t` format will be created to replace these interfaces and the old interfaces will "fail" appropriately.

This will be the format of the 64-bit devno under the 64-bit kernel:



V = Version bit

0 = Old Style 32-bit devno (Major:32-47; Minor 48-63)

1 = New Style (Major:2-31; Minor 32-63)

RSV = Reserved bit

Major = 30 bit major allowing up to 1 billion major numbers

Minor = 32 bit minor number allowing up to 4 billion minor numbers

The following macros will be provided to config methods and device drivers (available in 32-bit and 64-bit compiles) for extracting major and minor fields or creating 64-bit devnos.

```
/*
 * Extract major number from 64-bit devno
 */
#define major64(_devno) \
    ((int)((_devno & 0x3FFFFFFF00000000LL) >> 32))

/*
 * Extract minor number from 64-bit devno
 */
#define minor64(_devno) \
    ((int)(_devno & 0x00000000FFFFFFFFLL))

/*
 * Make a 64-bit devno
 */
#define makedev64(_major, _minor) \
    ((dev64_t)(((long long)_major << 32) | \
    ((long long)_minor & 0x00000000FFFFFFFFLL) | DEVNO64))
```

The following macros will be provided to 64-bit device drivers for testing and converting devnos from 32-bit to 64-bit and vice versa.

```
/*
 * Version bit for 64-bit devno
 */
#define DEVNO64 0x8000000000000000LL

/*
 * See if devno is 64 or 32
 */
#define ISDEVNO64(_devno) \
    (((ulong)_devno & DEVNO64) : TRUE ? FALSE)

/*
 * Convert to 64-bit devno.
 * Used by 64-bit drivers whose method passed in a 32-bit devno.
 */
#define DEV32TO64(_devno) \
    ((dev_t) (((ulong)_devno) & DEVNO64) : _devno ? \
```

```

        (((_devno & (ulong)0xFFFF0000) << 16) |
         (_devno & 0xFFFF) | DEVNO64)))

/*
 * Convert to 32-bit devno.
 * Used where an existing API only understands a 32-bit devno.
 * NOTE: major and minor bits greater than 2^16 are truncated (lost)
 */
#define DEV64TO32(_devno) \
    ((dev32_t) (!(((ulong)_devno) & DEVNO64) : _devno ? \

```

The following macros will be provided to device drivers to allow them to have common source between the 32-bit and 64-bit versions. Depending on the compile mode, these resolve to the appropriate version of the *major*, *minor*, and *vmakede* macros:

```

#ifdef __64BIT_KERNEL
/*
 * For DD source code simplicity, map the old to the new
 */
#define major_num(_devno) major64(_devno)
#define minor_num(_devno) minor64(_devno)
#define makedevno(_major, _minor) makedev64(_major, _minor)

#else /* __64BIT_KERNEL */

/*
 * For DD source code simplicity, map the old to the new
 */
#define major_num(_devno) major(_devno)
#define minor_num(_devno) minor(_devno)
#define makedevno(_major, _minor) makedev(_major, _minor)

#endif /* __64BIT_KERNEL */

```

64-bit Device drivers will need to understand 64-bit devnos and their new format under the 64-bit kernel. They will need to be in sync with their config method depending on what their method supports. For example, if the method was not ported to understand the new 64-bit devno, and continues to call **genmajor()/genminor()** and passes in a 32-bit devno to the driver, the driver will need to convert the devno to a 64-bit format, using the **DEV32TO64()** macro. Otherwise, the 64-bit driver should only need to worry about 64-bit devnos, i.e., on **open()**, **close()**, **strategy()**, etc. calls.

Application interfaces that must convert 64-bit devnos to 32-bit devnos for binary compatibility can use the **DEV64TO32()** macro. Initially, the maximum number of major numbers supported by the 64-bit kernel will be 2^{16} , which can still be represented within a 32-bit devno, so this conversion will suffice for this release and until there is a need to go beyond the 2^{16} th major or minor number limit.

3.3.12 Devswitch Interfaces

Some general items to be aware of with respect to the device switch table interfaces are the parameter types. As previously discussed, the **devno** parameter on each of the **config()**, **open()**, **close()**, **read()**, **write()**, etc... entry points should be declared as type **dev_t**. This ensures the field is the correct size for the active compilation mode. In addition, the **ext** parameter, or extended parameter, for interfaces such as **open()**, **read()**, **write()**, **ioctl()**, must grow to a **long** type in the 64-bit environment. A new typedef is provided in **m_types.h**, **ext_t**, for use in declaring the **ext** parameters. Another such parameter is the **arg** parameter to the **ioctl()** entry point. To the application programming interface, this parameter is defined as a **void ***. The exceptions to this are the **ioctl32()** and **ioctl32x()** application interfaces that were created so a 64-bit application could call the **ioctl()** entry point of a device driver even though the device driver was not 64-bit application safe. All 64-bit kernel extensions will be 64-bit application safe (see 64-bit Application Support), so this becomes a moot point. So, each driver should make sure to declare **arg** parameters as **void *** to get the natural growth to 64-bits in the 64-bit compilation environment.

There are currently a couple of flags used in the **d_opts** field of the **devsw** structure, **DEV_MPSAFE** and **DEV_64BIT**, which indicate that a device driver is Multi-Processor Safe and callable by 64-bit applications respectively. Both of these flags will remain and retain their current definition. However, in the 64-bit kernel environment all device drivers must be either MP safe or MP efficient. All device funneling support will be removed from the 64-bit kernel. Therefore, **devswadd()** will fail any calls to register a device switch entry that does not indicate **DEV_MPSAFE**. This also applies to the interrupt handler registration as indicated in the Interrupts section earlier, regarding the **INTR_MPSAFE** flag.

Also, in the 64-bit kernel environment all 64-bit kernel extensions will support being called by 64-bit applications and 32-bit applications. The issues surrounding a 32-bit kernel extension supporting a call from a 64-bit application dealt with the 32-bit kernel extension needing to reshape data structures passed in from the 64-bit application and depending on the interface also remap address spaces to gain 32-bit addressability to 64-bit process address spaces. Once ported to 64-bit, each kernel extension's default view of any private interface data structures match that of any 64-bit application, so the support is natural. Also there is no address space remapping required as the 64-bit kernel/kernel extension has full natural addressability to the 64-bit application address space. Instead, each 64-bit kernel extension will need to understand the 32-bit application's view of the shared data structure, and be able to reshape that into something it understands. This is still easier than the 64-bit application to 32-bit kernel extension case since there is still no address space remapping necessary. The 64-bit kernel and kernel extension

have full natural addressability to 32-bit application address space as well. The existing `_as_is64()` exported kernel service will still be used to distinguish whether the calling process is 32-bit or 64-bit. Please note that if the device driver is using the macro `IS64U` defined in `user.h` that it must not define `_KERNSYS`. If that happens, then the driver will end up making a direct reference to the **U-block**, which is a concern for updates and future incompatibilities if the definition of the **U-block** were to change.

Finally, one brief point regarding the `IOCINFO ioctl()` operation. The `devinfo` structure is and will remain a fixed-type field data structure. So, the view of this data structure from either a 32-bit or 64-bit compilation environment is identical. If and when a device driver must support a device that causes one of these fields to overflow its 32-bit capacity (for example, the `numblks` field of the disk union), that device driver will be responsible for having a new union added to the `devinfo` structure that will provide a fixed-type field data structure representing the new information as well as defining an appropriate new `devtype` corresponding to it.

3.3.13 32-bit Application Support

Full 32-bit application support is required of all 64-bit device drivers. There are no address space concerns, since there is no address remapping required to gain addressability to 32-bit application address space. The same model exists for direct references to application space, utilizing services like `copyin()`, `copyout()`, `xmattach()`, `xmemin()`, `xmemout()`, `xmempin()`, etc.. 64-bit kernel extensions will need to handle the difference in data structure sizes when shared with, i.e., passed between, 32-bit processes. Similar to what 32-bit kernel extensions had to do for 64-bit applications, the 64-bit kernel extension will need to reshape the data structures as necessary to understand them. The `IS64U` macro (via the `_as_is64()`) kernel service can be used to determine if a process is 32-bit.

What this means to common source device drivers that build both 32-bit and 64-bit executables is that there will need to be `#ifdef __64BIT_KERNEL` paths to define the appropriate code paths for the various compilation modes. For example, consider the following structures and code paths:

```
/*
 * General Structure Definition
 */
struct xyz {
    int  jj;
    long kk;
```

```

    void * mm;

} xyz;

#ifdef __64BIT_KERNEL

/*
 * Structure definition to give 64-bit extension 32-bit's view of data structure
 */

struct xyz32 {

    int jj;

    __long32_t kk;

    __ptr32 mm;

} xyz32;

#else

/*
 * Structure definition to give 32-bit extension 64-bit's view of data structure
 */

struct xyz64 {

    int jj;

    long long kk;

    __ptr64 mm;

} xyz64;

#endif

if (IS64U) {

```

```

/*
 * If 64-bit process....
 */
#ifdef __64BIT_KERNEL
    copyin((caddr_t)arg, &xyz, sizeof(struct xyz)); /* copyin general struct */
#else
    copyin((caddr_t)arg, &xyz64, sizeof(struct xyz64)); /* copyin 64-bit struct */
    as_remap64(xyz64.mm, size, &xyz.mm); /* remap the 64-bit data pointer */
#endif
} else {
    /*
     * else 32-bit process...
     */
#ifdef __64BIT_KERNEL
    copyin((caddr_t)arg, &xyz32, sizeof(struct xyz32)); /* copyin 32-bit struct */
#else
    copyin((caddr_t)arg, &xyz, sizeof(struct xyz)); /* copyin general struct */
#endif
}

```

3.3.14 64-bit Application Support

Full 64-bit application support is required of all 64-bit device drivers. The definition of "full 64-bit application support" may very well mean that your driver returns **EINVAL** if called by a 64-bit application and you don't support some specific function for 64-bit apps. The general idea is that the 64-bit kernel will not "police" individual calls through the **devsw** table or **sysconfig()** like the 32-bit kernel does. This level of policing is being pushed down to the individual 64-bit device drivers.

In the 32-bit kernel, **sysconfig()** used the **SYS_64BIT** bit of the command to determine if the module should be callable by 64-bit applications. This bit is ignored in the 64-bit kernel. The device driver's entry point will need to check whether the caller is a 64-bit application and either handle it or return a failure.

The level of policing in the 64-bit kernel will be limited to the **devswadd()** service. In order for the **devswadd()** registration of a device driver to be successful in the 64-bit kernel environment, it must indicate full support for 64-bit applications with the **DEV_64BIT** flag of the **d_opts** field of the **devsw** structure. Support for 64-bit applications by 64-bit kernel extensions is completely natural as data structures are viewed consistently and no address remapping is required.

The model for direct references to application space is common with that of the 32-bit application space, using the common set of services like **copyin()**, **copyout()**, **xmattach()**, **xmemin()**, **xmemout()**, **xmempin()**, etc..

3.3.15 Pinning Memory

With the exception of **pinu()/unpinu()**, the existing pin and unpin interfaces will continue to be supported and function as is (**xmempin()**, **xmemunpin()**, **pin()**, **unpin()**, and **pincode()**, **unpincode()**). The **pinu()** and **unpinu()** interfaces will not be supported in the 64-bit kernel. Instead, device drivers should **xmattach()** to the user space data and then use **xmempin()** and **xmemunpin()**. This change can be common between the 32-bit and the 64-bit device drivers, since the latter method is currently supported on 32-bit and will be supported on 64-bit.

3.3.16 Cross Memory

To the device drivers, the cross memory model appears unchanged. All of the existing cross memory services (**xmattach()**, **xmdetach()**, **xmempin()**, **xmemunpin()**, **xmemin()**, **xmemout()**) will be supported and function as they do today. The size and content of the cross memory descriptor (struct **xmem**) will be changing in the 64-bit kernel, but the **xmem** structure is and should be treated as opaque.

Currently, the cross memory model can support a single segment crossing (**XMEM_PROC2**, for big data buffers that span contiguous segments). In the 64-bit kernel there is no limit on the number of segment crossings supported by the cross memory model.

Kernel extensions should always use cross memory services when dealing with a cross memory descriptor and never attempt to interpret its contents. Client file systems have been guilty of this in the past by **vm_att()**ing directly to the **subspace_id** within the cross memory descriptor to gain addressability to the address space.. There are kernel services provided , **xm_att()** and **xm_det()**, (as of AIX 4.3) that attach and detach to/from an address space described by a cross memory descriptor that should be used for this purpose.

3.3.17 Error Logging

Each error template should be investigated to see if any of the detail data being logged grows in size due the 64-bit port. If so, the driver owner has a couple of choices. One would be to update the error template to reflect the new detail data size and define a size-invariant structure (same size in 32-bit or 64-bit) so that the same number of bytes are logged from either the 32-bit or 64-bit driver. The second option would be to define a new error template for the 64-bit case, with the proper **#ifdef**'s within the driver that chose the correct one.

3.3.18 Power Management

For now, assume that Power management will be supported by the 64-bit kernel environment just as it is in the 32-bit kernel environment. For the initial porting of the drivers, porting the power management specific functions of the driver should be the least priority.

4.0 Converting 32-bit Applications to 64-bit Applications

Before going through the effort of converting an application from 32-bit to 64-bit, it is important to understand whether the conversion will lead to a measurable benefit in scalability and performance. To make that determination, let's first understand the benefits of 64-bit systems and what changes to the application are needed to take advantage of these benefits.

The Intel Merced chip is an implementation of the Intel IA-64 bit architecture and Monterey IA-64 is a 64-bit kernel. This combination offers software developers the following advantages not available on 32-bit systems:

- Full 64-bit addressing that expands the addresses available to the application far beyond the 4GB limit on 32-bit systems.
- 64-bit data elements (integers) with instructions for performing efficient arithmetic computations.
- Support for large data structures and executables.
- Support for physical memory beyond 4GB.
- Support for larger file sizes.
- Greater scalability of system data types such as *time_t*, *date_t*, ..

Software developers must then determine if the application:

- Can utilize the large address space for more buffer pool, for mapping files into memory, for *shmat* and *mmap*
- Can benefit from more physical memory (> 4GB) and, if so, is the user of the application likely to implement it on a system with more than 4GB?
- Needs 64-bit integers.
- Needs larger files and data structures than can be supported on 32-bit systems.

4.1 Overview

Most applications are written in one or more high level languages. Since applications written in assembly or macro assembly will need complete rewrites, porting issues will be discussed in terms of C. Variations of the problems occur with other languages.

Most well written programs will compile and run without change, where "well written" implies the use of good programming practices including:

- Conformance to the ANSI/ISO C standard
- Portability considerations high in the design, implementation and maintenance phases of the software cycle
- Use of prototyped functions declarations throughout Most applications are written

In reality, portability issues are often the first to be sacrificed to meet completion schedules or are forgotten in the software maintenance cycle. In other cases, production code is written

without regard for portability. The base of much of the C source code to be ported to 64 bit Monterey IA64 has existed for a long time and been through many maintenance and enhancement cycles. During these processes it is quite easy for assumptions, implicit or explicit, about either absolute or relative sizes of int, long, and pointer data types to become part of the C source code.

These assumptions in combination with the changes in size and alignment of basic data types will be the source of most problems porting existing source code to a 64 bit system. This document will detail the areas where problems may occur and explain how the C compiler in combination with lint can be used to isolate and correct problems.

4.2 C and C++ Data Type Size and Alignment Issues

Support for a 64 bit address space and larger scalar arithmetic ranges in LP64 mode naturally requires changes in at least some of the basic C data types. As shown in the following table, pointers, longs, and long long ints in LP64 mode are 8 bytes (64 bits). The alignment restrictions for these data types, as well as the size and alignment of long double floating point types, have changed for performance considerations. It should be noted that the rules for size and alignment for ILP32 are the same as those for IA-32.

C & C++ Data types	AIX on PowerPC		Monterey on Merced	
	32-bit source	64-bit source	ILP32	LP64
	Size/alignment	Size/alignment	size/alignment	size/alignment
char	1/1	1/1	1/1	1/1
short	2/2	2/2	2/2	2/2
int	4/4	4/4	4/4	4/4
long	4/4	8/8	4/4	8/8
long long	8/8	8/8	8/4	8/8
pointer	4/4	8/8	4/4	8/8
float	4/4	4/4	4/4	4/4
double	8/4	8/4	8/4	8/8
long double	16/16	16/16	12/4	16/16

* Note that long long data types in 32-bit applications are 8 bytes in size and aligned on 8 byte boundaries. 64-bit applications will continue to treat long long data types as 8 bytes in size and aligned on 8 byte boundaries.

In porting 32-bit applications to 64-bit, some of the problems that will be encountered are related to:

- sizeof(int) != sizeof(long)
- sizeof(void *) != sizeof(int)
- objects change size
- lack of prototyped function declarations

4.3 Sizeof(int) != Sizeof(long)

In ILP32 mode, both "int" and "long int" are 32 bits in size. Because of this similarity, these types may have been used interchangeably in production code. As shown in the table above, in LP64 mode, long data is 64 bits in length. A general guideline is to review existing use of long data types throughout the source code. If the values to be held in such variables, fields and parameters will fit in the range of 2Gig -1 to -2Gig or 4Gig to 0, then it is probably best to use int or unsigned int, respectively.

4.4 Truncation of 64 bit value when assigned to a smaller type

The assignment of a long int value to a smaller type will result in truncation of the 64 bit value. This may be exactly the intention of the code, but where int and long have been used interchangeably this truncation may be an unexpected source of problems.

```
1  int i1, i2, i3;
2  long l1, l2, l3;
3
4  extern long retlong(int);
5
6  void long2int(void) {
7      /* implicit truncation on the next 3 statements */
8      i1 = l1;      /* 64 bit value => 32 bit "i1"      */
9      i2 = l2 * l2; /* 64 bit expression value => 32 bit "i2" */
10     i3 = retlong(l3); /* 64 bit "l3" passed to 32 bit prototyped param
11                      64 bit return value => 32 bit "i3"      */
12
13     i1 = (int) l1;
14     i2 = (int) (l2 * l2);
15     i3 = (int) retlong((int)l3);
16 }
```

4.4.1 Recommendation

Examine all instances of narrowing assignment, particularly where the source is of type long or unsigned long, and decide whether such narrowing may be a problem or is intended. Use an explicit cast at the point of an intended narrowing conversion to indicate to the compiler and to lint that such narrowing is being done by design.

Lint assistance

Lint will report implicit narrowing integral conversions associated with the = operator, as shown in the following lint output.

```
assignment causes implicit narrowing conversion
(8) int = long
(9) int = long
(10) int = long
```

4.5 Explicit cast improperly applied

Explicit narrowing casts should be used on expressions, not operands. In the following example, in LP64 mode the compiler and lint will warn of the implicit narrowing of the long expression "l1 / i1" to an int in statement 6. Line 8 shows a similar statement with an explicit cast which will suppress the lint warning, but changes the result of the expression. The cast is applied to the long variable before the division. Statement 10 illustrates a properly applied explicit cast that will yield identical results with statement 6.

```
1  int i1;
2  int r1, r2, r3;
3  long l1;
4
5  void foo() {
6    r1 = l1 / i1;    /* gets compiler and lint warning */
7
8    r2 = (int)l1 / i1; /* l1 is truncated to 32 bits before divide */
9
10   r3 = (int) (l1 / i1); /* the result of the division is truncated to
11                          32 bits */
12 }
```

4.5.1 Recommendation

Narrowing casts should be applied to expressions.

Lint assistance

Lint will flag the statement on line 6 as containing an implicit narrowing conversion.

```
assignment causes implicit narrowing conversion
(6) int = long
```

4.6 A pointer to an int is not compatible with a pointer to a long

Pointers to different unqualified types are not compatible in C and a pointer to one type should not be assigned to a pointer of another type. For historical reasons, however, most compilers do not stringently enforce this restriction and comply with the ANSI/ISO C Standard by issuing a warning. In source code where `int` and `long` have been used interchangeably, pointers to `int` and `long` may have also been used interchangeably. In LP64 mode, these point to objects of different size and subsequent de-reference of such a pointer will clearly result in undefined behavior.

```
1 long *ptr1, *ptr2;
2 int *ptr1, *ptr2;
3
4 extern foo_int(int *);
5 extern foo_long(long *);
6
7 void bar(void) {
8
9     ptr1 = ptr1;
10    ptr2 = ptr2;
11
12    ptr1 = (long *)ptr1;
13    ptr2 = (int *)ptr2;
14
15    foo_int(ptr1);
16    foo_long(ptr1);
17
```

Both the C compiler and lint report the incompatibility of pointer assignments in statements 9, 10, 15 and 16 in the example above. In addition to the obvious object size mismatches that would occur if these pointers were de-referenced, the alignment requirements for `int` and `long` are different in LP64 mode. If a pointer to `long` is used to reference a memory address that is not 8-byte aligned, an alignment fault will occur. This will, in turn, require intervention by the operating system and degrade application performance.

Lines 12 and 13 in the above example, while valid C code, still present the same problems as lines 9 and 10. The explicit casts have probably been introduced into the source code to quiet the C compiler and lint. Use of lint with the `-p` option to select portability checking will flag these statements for review.

Any pointer type may be assigned to or from a `void *`. Any code which effectively assigns an `int *` to a `long *`, or the reverse, through an intermediate `void *` variable or function parameter may exhibit the undefined behavior possible in the above example.

4.6.1 Recommendation

Examine all instances of incompatible pointer assignments, particularly those involving a long * data type. The type of the object pointed to should be made consistent and that choice should be based on the range of values to be held by the object.

For cases where the types pointed to are intentionally different, as with char * pointers returned from older memory allocation or memory management routines, use an explicit cast to indicate to the compiler and to lint that this is intentional. A better solution is to bring the code up to ANSI C specifications and use void * for generic pointers. Remember to consider any alignment issues.

Lint assistance

In addition to flagging the incompatible pointer assignments as the compiler does, lint also notes that there is a potential alignment issue with the explicit cast of an int * to a long * in statement 12.

- (9) Assignment type mismatch
- (10) Assignment type mismatch
- (15) Argument is incompatible with prototype: arg #1
- (16) Argument is incompatible with prototype: arg #1

pointer cast may result in improper alignment
(12)

Running lint with a "-p" option will flag all pointer conversions, excluding conversions to or from void *, for review.

- pointer casts may be troublesome
- (9) (10) (12) (13)
 - (15) (16)

4.7 Lack of prototyped function declarations in scope of call statements

Passing arguments to a function is essentially the assignment of values to the formal parameters of the called function. For calls to functions with a prototyped declaration in scope, these assignments have implicit conversions where argument types differ from the corresponding formal parameter type. For calls to functions lacking a prototyped declaration in scope, *default argument promotions* are performed on each argument. For integer data types that are 32 bits or less in size, the integral promotions will yield 32 bit int or *unsigned int* types. If these 32 bit values are used as 64 bit values by the called function, the behavior, according to the ANSI/ISO C Standard, is undefined. This applies to both ILP32 and LP64 compilation modes.

The IA-64 calling conventions state that integral scalar parameters smaller than 64 bits are placed in the least significant bits of a 64 bit argument slot, padded on the left; the contents of

the padding are undefined. Passing a non-64 bit value to a function that will use the information as a 64 bit data type will result in using undefined bits. While some versions of the C/C++ compiler, particularly with optimization disabled, may sign or zero extend arguments to 64 bits, this behavior is not guaranteed.

A similar problem will occur with a function returning a 64 bit value and no prototyped function declaration visible at the point of call. The implicit return type is int and the callee will only expect a 32 bit value from the function called. The high order 32 bits of the return value are truncated. Note, use of implicit types is nonstandard for C++ and being considered for C by the ANSI/ISO standardization committees.

The following example illustrates various forms of external function declarations that appear in existing code, from non existent to prototyped. The calls to functions func1() and func2() exhibit both problems:

- only 32 bit values are passed as arguments; the high order 32 bits of the argument are undefined.
- only 32 bits of the return value are used following the function call.

The call to function func2_A() assumes an implicit int return type as happens in the previous two calls. The call to function func3() in the presence of a fully prototyped function declaration will correctly pass a sign extended 64 bit argument and handle a 64 bit return value.

argsret.1.c:

```
1 extern func2();
2 extern func2_A(long);
3 extern long func3(long);
4
5 static short s;
6 static int i, j;
7 long l1, l2, l3;
8
9 void foo () {
10     l1 = func1(s); /* no function declaration visible */
11     l2 = func2(i); /* implicit return type & old style param list */
12     l2 = func2_A(i); /* implicit return type */
13
14     l3 = func3(j);
15 }
```

argsret.2.c:

```
1 extern long l1, l2, l3;
2
3 long func1 (arg)
```

```

4   long arg;
5   {
6   return arg * 11;
7   }
8
9   long func2 (arg)
10  long arg;
11  {
12  return arg * 12;
13  }
14
15  long func2_A (long arg) {
16  return arg * 12;
17  }
18
19  long func3 (long arg) {
20  return arg * 13;
21  }

```

4.7.1 Recommendation

Prototyped function declarations should be visible at all call sites, particularly for functions with 64 bit parameters or 64 bit return types. This applies to ILP32 mode as well as LP64 mode. Both the compiler and lint should be used to locate all places where:

- functions appear to be declared implicitly
- functions are declared with an "old-style" parameter list
- functions appear to have an implicit return type
- lint reports that function types or arguments appear to be declared or used inconsistently across source files.

The combination will clearly locate problems in K&R or ANSI C source code and once corrected for LP64 mode will also work in ILP32.

Lint assistance

If lint is run on all source files that make up a binary, it will flag:

implicitly declared functions (at the point of call) functions declarations with "old-style" parameter lists (flagged at the point of call) functions with an implicit return type of int argument types used inconsistently function return types used or declared inconsistently

```

argsret.1.c:
(1) no type specifiers present
(2) no type specifiers present

```

implicitly declared to return int

```
(10) func1
```

called lacking visible prototype declaration

```
(11) func2
```

```
argsret.2.c:
```

value type used inconsistently

```
func1      argsret.2.c(5) long () :: argsret.1.c(10) int ()
func2      argsret.2.c(11) long () :: argsret.1.c(11) int ()
func2_A    argsret.2.c(15) long () :: argsret.1.c(12) int ()
```

value type declared inconsistently

```
func2      argsret.2.c(11) long () :: argsret.1.c(1) int ()
func2_A    argsret.2.c(15) long () :: argsret.1.c(2) int ()
```

function argument (number) used inconsistently

```
func1 (arg 1)  argsret.2.c(5) long :: argsret.1.c(10) int
func2 (arg 1)  argsret.2.c(11) long :: argsret.1.c(11) int
```

4.8 Integer expression with potential overflow is converted to a long

Arithmetic expressions are evaluated following the usual arithmetic conversions of all operands to a common type. The type of the expression is this common type. For an expression containing integral operands, that implies that small operands will be converted to int as needed to represent all values of the original type. The common type will only be larger than an int if an operand of the expression is an unsigned int, long, or unsigned long. What this means for LP64 mode is that expressions not containing a long or unsigned long type will be evaluated in terms of 32 bit values and yield a 32 bit result.

The following example illustrates an instance where in LP64 mode the actual results may not be what the user desired.

```
1  int a, b;
2  long l;
3
4  void foo (void) {
5
6      l = a * b;      /* 32 bit multiply with potential truncation */
7
8      l = (long) (a * b); /* 32 bit multiply with potential truncation */
9
10     l = (long)a * b; /* 64 bit multiplication - no truncation */
```

```

11     l = a * (long)b;    /* 64 bit multiplication - no truncation */
12 }

```

If the user intended to get a 64 bit result from the multiplication, lines 6 and 8 are incorrect. Both source statements contain a 32 bit multiplication with the result truncated to 32 bits and cast, implicitly in line 6, to a 64 bit long value. Statements 10 and 11 show the correct way to get a 64 bit multiplication of two smaller types. By casting either operand to a long prior to the multiplication, the other operand is implicitly cast to a long.

4.8.1 Recommendation

To have integral expressions produce 64 bit results, at least one of the operands must have a data type of long or unsigned long. If necessary, an appropriate cast to long or unsigned long should be applied to one of the operands. That cast will have the effect of percolating up the expression tree to yield a 64 bit result.

A cast applied to an expression of int or unsigned int type, implicitly or explicitly, will only yield a 64 bit sign or zero extended representation of the 32 bit value.

Lint assistance

None.

4.9 Untyped integral constants are *int* by default

To be more specific, the C Standard states that the type of an integer constant, depending on its format and suffix, is the first (smallest) type in the corresponding list which will hold the value. The quantity of leading zeros does not effect the type selection.

unsuffixed decimal number	int, long int, unsigned long int
unsuffixed octal or hexadecimal number	int, unsigned int, long int, unsigned long int
suffixed by u or U	unsigned int, unsigned long int
suffixed by l or L	long int, unsigned long int
suffixed by both u or U and l or L	unsigned long int

Code that:

- does not take into consideration that integral constants may be represented as 32 bit types even when used in expressions with 64 bit types
- assumes that long or unsigned long data is 32 bits in length
- depends on specific behavior at an assumed data type length

may behave differently when compiled for LP64 mode than when compiled for ILP32. The following example has several instances where code may not behave as expected.

Case number 1 is a special form of an integer expression with overflow being used in a 64 bit expression. The two constants in line 7 are 32 bit integer constants and the integer multiplication results in a 32 bit overflow with the truncated folded constant value of 1,658,683,392 added to "l2". By using the type suffix "L" to specify a long type on at least one of the constants, as shown in line 9, the multiplication will be done with 64 bit constants. Similarly, either constant could have been explicitly cast to a type of long.

Case 2 illustrates a hexadecimal constant with the 2**31 bit set. Because the significant bits of the constant in line 15 will fit into 32 bits, it has a type of unsigned int. The bitwise complement will yield an unsigned int constant of value 0x7fffffff. In ILP32 mode, the assignment and operator would effectively turn off the 2**31 bit. In LP64 mode, the unsigned int would be converted to a long with the value 0x000000007fffffff; effectively turning off 33 bits of the value on "l1". Line 16 would have the same result since leading zeros are insignificant in determining the data type of the constant. Lines 18 and 19 of the example show use of either an explicit cast or a type suffix, respectively, to be certain that the constant is treated as a 64 bit value. These two lines will turn off the 2**31 bit in both LP64 and ILP32 mode.

In case 3, the constant is a 32 bit unsigned int, with a value of 4,294,967,295 in both ILP32 and LP64 mode. The addition is done as an unsigned long, which is cast to a type long. In ILP32, the result has a value of 11 - 1 because of the truncation to 32 bits. In LP64 mode, the addition result is a 64-bit long with a value of 11 + 4,294,967,295.

Case 4 is an example of code that presumes to know the number of bits in a long data type. The code is attempting to extract bits 11 - 26 from the long variable l2 as a signed quantity. The left shift in line 34 is depending on truncation occurring at bit 32 and while the code will work in ILP32 mode, it will not port to LP64 mode. Code which assumes to know the size of any data type other than char is not portable. Code that assumes the size of a long or unsigned long data type will certainly be a problem porting to LP64 mode. Line 36, where

- 8 = number of bits per char and should be represented as an architecture dependent #define
- 27 = one more than the highest bit desired in the result
- 16 = size of the field being extracted

will yield identical results in both LP64 and ILP32 modes.

```
1 long l1, l2;
2
3 void foo(void) {
4
5 /* case 1 - constants are int.
```

```

6  */
7  11 = 12 + 20000000 * 30000000; /* 32 bit multiplication. */
8
9  11 = 12 + 20000000L * 30000000; /* 64 bit multiplication. */
10
11
12 /* case 2 - constant is an unsigned int. leading zeros are not
13 **      significant.
14 */
15  11 &= ~(0x80000000); /* turns off 33 bits. */
16  11 &= ~(0x0000000080000000); /* turns off 33 bits. */
17
18  11 &= ~((long)0x80000000); /* turns off bit 2**31 */
19  11 &= ~(0x80000000L); /* turns off bit 2**31 */
20
21 /* case 3 - code depending on truncation at 32 bits on overflow.
22 */
23  11 += 0xffffffff; /* 11 = 11 - 1 in ILP32 mode on
24                    2's complement system */
25                    /* 11 = 11 + 4,292,967,295
26                    in LP64 mode. */
27
28 /* case 4 - assuming that the size of a long is 32 bits and depending
29 **      on truncation of bits at 32 and beyond on shift left.
30 */
31 /* isolating bits 11 to 26 as a signed number - bit 0 is least
32 ** significant bit.
33 */
34  11 = (12 << 5)>> 16; /* depends on truncation at bit 32 */
35
36  11 = (12 << (8 * sizeof(12) 27))>> (8 * sizeof(12) - 16);
37 }

```

4.9.1 Recommendation

Usage of all constants including symbolic constants established with preprocessor #define statements should be reviewed. Special attention should be given to

- long or unsigned long expressions containing constants used in integer subexpressions which may overflow the maximum or underflow the minimum values expressible in 32 bits.
- expressions containing octal or hexadecimal constants whose high order bit is 2**31.
- expressions depending on truncation at bit 32 on an overflow.
- left shift expressions that assume truncation at bit 32.

Lint assistance

Unfortunately, other than the warning about overflow in the constant folding on line 7, lint cannot assist you in locating porting problems of these types.

4.10 `Sizeof(void *) != Sizeof(int)`

As the term LP64 implies and the table illustrates, in LP64 mode, pointers are 64 bits in length and aligned on 8 byte boundaries. This change in size will present a problem when porting existing ILP32 code to LP64 if:

- pointers are converted to int or unsigned int with the expectation that the pointer value will be preserved.
- the code assumes that pointers and int are the same size in an arithmetic context.
- expects a pointer as the return value from a function lacking a declaration in scope, implicitly declared to return an int or declared to return any data type that is 32 bits or less in size.

4.11 Truncation of a 64 bit pointer value when converted to a smaller integral type

As in the case of an LP64 long, assignment of an LP64 pointer value to a 32 bit data type variable will result in truncation of the pointer value. The pointer value cannot reliably be reconstructed from the int or unsigned int. If the address value being converted is in range of 0 to 4 gigabytes, which happens to fit in 32 bits, the code may appear to work only to fail with a different memory layout.

Since ANSI conforming C compilers are required to provide a diagnostic, usually a warning, for integral to pointer and pointer to integral assignments, existing source code is likely to have an explicit cast on these assignments. These explicit casts have been introduced to suppress the diagnostics from the compiler and lint. In LP64 mode, if the conversions are to any type less than 64 bits, these conversions are likely to be a source of porting problems. The following example shows a combination of explicit and implicit pointer to integer conversions done at assignment time that could ultimately lead to problems.

```
1 int i;
2 long l;
3 char * chptr;
4 void * voidptr;
5
6 extern void bar_int(int, int);
7 extern void bar_long(long, long);
8
9 void foo() {
10
```

```

11  i = chptr;          /* implicit - loss of bits */
12  i = voidptr;       /* implicit - loss of bits */
13  i = (int)chptr;    /* explicit - loss of bits */
14  i = (int)voidptr;  /* explicit - loss of bits */
15
16  bar_int(chptr, (int)voidptr); /* loss of bits - both args */
17
18  l = chptr;
19  l = (long)chptr;
20  bar_long(chptr, (long)voidptr);
21
22  }

```

4.11.1 Recommendation

Code involving conversions of pointers from or to integral values should be reviewed. If these pointer to integral conversions are absolutely necessary, the integral type should be either long or unsigned long and an explicit cast to long or unsigned long should be used.

Fully prototyped function declarations should be in scope at the point of all calls, allowing the C compiler and/or lint to scrutinize pointer to integral conversions of function arguments.

Lint assistance

Lint will not only flag all occurrences of non-conforming implicit pointer to integer conversions, but also flag all explicit conversions that may lose significant bits.

```

(11) improper pointer/integer combination: op "="
(11) conversion of pointer loses bits
(12) improper pointer/integer combination: op "="
(12) conversion of pointer loses bits
(13) conversion of pointer loses bits
(14) conversion of pointer loses bits
(16) improper pointer/integer combination: arg #1
(16) conversion of pointer loses bits
(16) conversion of pointer loses bits
(18) improper pointer/integer combination: op "="
(20) improper pointer/integer combination: arg #1

```

4.12 Assumption that pointers and int are same size in arithmetic context

With the exception of a pointer +/- an integer value and pointer difference, pointers may not directly be used with arithmetic or bitwise operators. There are times when a pointer must be explicitly cast to an integral type to be used with these operators. Such an example would be the UnixWare kernel's use of bitwise shifts and bitwise AND operations to determine the memory

segment containing a particular address. These explicit casts should be to either long or unsigned long which will preserve the 64 bit values in LP64 mode and the 32 bit values in ILP32 mode.

The following example contains source code from the Bourne shell command which does its own memory management. The code on line 9 assumes that the size of a pointer is the same as an int which is incorrect in LP64 mode. The result is that 64 bit pointer will be converted to a 32 bit value and OR'ed with the BUSY bit. The 32 bit integer result is then cast back to a pointer and 32 bits of the address have been lost.

```
1 #define BUSY 01
2
3 void foo(void) {
4     struct blk *p, *word;
5     word = (struct blk *)(((int)p) | BUSY);
6
7     /* correctly casting to 64-bit integer */
8     word = (struct blk *)(((unsigned long)p) | BUSY);
9 }
```

4.12.1 Recommendation

All pointer casts to integer types should be to either long or unsigned long. The source will then work for the ILP32 and the LP64 compilation models.

Lint assistance

As in the previous example, lint will flag pointer to integer conversions which may result in lose of bits.

(5) conversion of pointer loses bits

4.13 Pointer return type or argument types in the absence of a prototyped function declaration

This is a specific form of the porting issues dealing with 64 bit values used as function parameters and function return types in the absence of a prototyped function declaration in scope. In LP64 mode, a null pointer value (integer constant zero) used as an argument to a function without a prototyped function declaration, may be passed only a 32 bit zero. The high order 32 bits will be undefined. Likewise, in LP64 mode, a 64 bit pointer returned by a function to a callee that does not have a prototyped function declaration in scope, will be treated as an int and truncated to 32 bits.

```
ptrargsret.1.c:
1 extern func2();
2 extern func2_A(char *);
3 extern char * func3(char *);
```

```

4
5 #define NULL 0
6
7 void foo () {
8     char * ptr1;
9     char * ptr2;
10    char * ptr3;
11
12    ptr1 = func1(NULL); /* no function declaration visible */
13    ptr2 = func2(NULL); /* implicit return type & old style param list */
14    ptr2 = func2_A(NULL);/* implicit return type */
15
16    ptr3 = func3(NULL);
17 }

```

ptrargsret.2.c:

```

1 char * func1 (arg)
2     char * arg;
3 {
4     return arg;
5 }
6
7 char * func2 (arg)
8     char * arg;
9 {
10    return arg;
11 }
12
13 char * func2_A (char * arg) {
14     return arg;
15 }
16
17 char * func3 (char * arg) {
18     return arg;
19 }

```

4.13.1 Recommendation

As recommended previously, prototyped function declarations should be visible at all call sites. Both the compiler and lint should be used to locate all places where:

- functions appear to be declared implicitly
- functions are declared with an "old-style" parameter list
- functions appear to have an implicit return type

- lint reports that function types or arguments appear to be declared or used inconsistently across source files.

The combination will clearly locate problems in K&R or ANSI C source code and once corrected for LP64 mode will also work in ILP32.

Lint assistance

If lint is run on all source files that make up a binary, it will flag:

implicitly declared functions (at the point of call) functions declarations with "old-style" parameter lists (flagged at the point of call) functions with an implicit return type of int argument types used inconsistently function return types used or declared inconsistently

ptrargsret.1.c:

- (1) no type specifiers present: assuming "int"
- (2) no type specifiers present: assuming "int"
- (12) improper pointer/integer combination: op "="
- (13) improper pointer/integer combination: op "="
- (14) improper pointer/integer combination: op "="

implicitly declared to return int

(12) func1

called lacking visible prototype declaration

(13) func2

ptrargsret.2.c:

name defined but never used

foo ptrargsret.1.c(7)

value type used inconsistently

func1	ptrargsret.2.c(3) char *() :: ptrargsret.1.c(12) int ()
func2	ptrargsret.2.c(9) char *() :: ptrargsret.1.c(13) int ()
func2_A	ptrargsret.2.c(13) char *() :: ptrargsret.1.c(14) int ()

value type declared inconsistently

func2	ptrargsret.2.c(9) char *() :: ptrargsret.1.c(1) int ()
func2_A	ptrargsret.2.c(13) char *() :: ptrargsret.1.c(2) int ()

4.14 Objects Change Size

Data objects that contain pointer, long, long long, or long double data types will have different sizes in ILP32 and LP64 modes. While the long long data type is currently 8 bytes in both models, the alignment restrictions differ. Both the size and alignment of long long will change before hardware availability. The following simplistic example of a linked list C data structure that can be used to illustrate the size difference.

```
struct dummy {
    struct dummy *next;
    struct dummy *prev;
    int data;
    char *name;
};
```

In an ILP32 model, the structure occupies 16 bytes of memory. In the LP64 model, this structure is 32 bytes in size. The size increase is the result of doubling the size of the three pointers (12 bytes) and an additional 4 bytes of alignment padding preceding the last pointer.

This change in data structure sizes may not be a problem. If the data is to be solely used by the binary that produced the data or another program compiled for the same compilation model, the size difference is not an issue with the exception of a potential size problem. A problem does exist where an LP64 model binary must consume data produced by a ILP32 model binary or where the data flow is in the opposite direction.

With careful design, compatible data structures can be defined to allow sharing of data between binaries from different models. Combined use of

```
#if #model(lp64) or #if #model(ilp32)
#pragma packed (4) and #pragma packed ()
```

will allow declaration of a structure that is binary data compatible between compilation models.

```
struct shared_models {
#ifdef #model(lp64)
    long    64bit_value;
    int     32bit_value;
    int     other_32bit_value;
    long double big_fp_value;
#else
    long long 64bit_value;
    long     32bit_value;
    int     other_32bit_value;
    long double big_fp_value;
    int     padding;
#endif
};
```

```
#endif
};
```

The above structure illustrates a C data structure that is binary data compatible in either ILP32 or LP 64 mode. The declaration preserves the alignment and size of each structure member.

In cases where conformance to the preferred LP64 alignments is not feasible, the structure declaration can be wrapped in a set of `#pragma pack` directives to cause the structure layout to match that of the ILP32 model.

```
#pragma pack (4)    /* set to ILP32 most strict alignment */
struct s {
    ....
    ....
};
#pragma pack ()    /* return to alignment of current model */
```

Sharing of pointer values between ILP32 and LP64 applications is meaningless. In serious database-oriented applications pointers rarely appear in declarations of data written to mass storage devices. These applications are normally concerned about efficient use of storage and already avoid pointers. File offsets can be expressed in terms of the 64 bit data type available in each model.

In cases where a LP64 program must deal with an ILP32 data structure that contains pointers more effort is required. Assuming that data written out in pointer fields is irrelevant or expressed in terms of some offset and will be filled in when the structure is memory resident, a new data structure that encapsulates the old data can be defined. Care must be taken to preserve the alignments in the old structure.

4.15 Integer Constants

- The type is determined by shape and value
- Leading and high order zeroes only serve to denote octal and have no other effect on size.

General rules:

- Decimal constants find first signed type that holds the value, small or large
- Other bases find first signed or unsigned type that holds the value, small or large
- Suffixes (combinations of *u* or *U*, and *l* or *L*, and *ll* or *LL*) generally restrict the choices.
- Porting code that uses integer constants must:
 - Consider that integer constants may be more than 32 bits
 - Not assume that *long* or *unsigned long* data is 32 bits
 - Not depend on the specific behavior at an assumed data type length

Integer constant examples include:

* Expression truncated at 32 bits

```
long1 = long1 + 20000000 * 30000000; /* 32 bit expression */
long2 = long2 + 20000000L * 30000000; /* 64 bit expression */
```

* Expression depends on 32 bit truncation

```
long1 += 0xffffffff; /* long1-1 for ILP32
                    Long1 +4294967295 for LP64 */
```

* Constant has *int* size, not *full size* (leading zeroes do not increase the size)

```
long1 &= ~0xffff0000; /* clears 48 bits */
long1 &= ~0x00000000ffff0000; /* clears 48 bits */
long2 &= ~(long) 0xffff0000; /* clears 16 bits)
long2 &= ~0xffff0000L; /* clears 16 bits */
```

* LONG_MIN, LONG_MAX, ULONG_MAX will have different values in LP64

Type	Hexadecimal	Equation
LONG_MIN	0x8000000000000000L	$-(2^{63})$
LONG_MAX	0x7FFFFFFFFFFFFFFFL	$(2^{63})-1$
ULONG_MAX	0xFFFFFFFFFFFFFFFL	$(2^{64})-1$

* Shifts expecting 32 bit operands can be hidden in macro expansions:

```
ulong1 = (ulong1 << 5) >> 16; /* ILP32: keeps bits 11-26
                               LP 64: bits 11-58 */
long1 = (long1 << 5) >> 16; /* ILP32: might sign extend 11-26
                              LP64 bits 11-58 */
ulong1 = (ulong1 & 0x7fff800) >> 11;

long1 = (long1 << (CHAR_BIT * sizeof(long) - 27))
        >> (CHAR_BIT * sizeof(long) - 16);
```

4.16 Stack layout changes due to larger data elements

- System data types such as *time_t* become 8 bytes long
- Fixed Size Data Types defined in *<sys/types.h>*. For example:

Fixed Size Data Types		ILP32 and IA-32		LP64	
signed	unsigned	Size (bits)	Align (bytes)	Size (bits)	Align (bytes)

int8_t	uint8_t	8	1	8	1
int16_t	int16_t	16	2	16	2
int32_t	int32_t	32	4	32	4
int64_t	int64_t	64	4	64	8

5.0 Byte Ordering

Endianness refers to how a data element and its individual bytes are stored (or addressed in memory). For BE, the lowest address is associated with the most significant (or leftmost) byte of a multi-byte value, while a little-endian gives the lowest address to the least significant (or rightmost) byte of a multi-byte value. In general, bit zero is associated with the most significant bit (MSB) for BE; while it is associated with the least significant bit for LE.

A 4-byte word with LE and BE representations are shown to illustrate that the data itself in a word are the same in both cases (see Figure 1); while the byte address associated with each data byte is different. Bytes are addressed in an opposite direction and the rest remains the same!

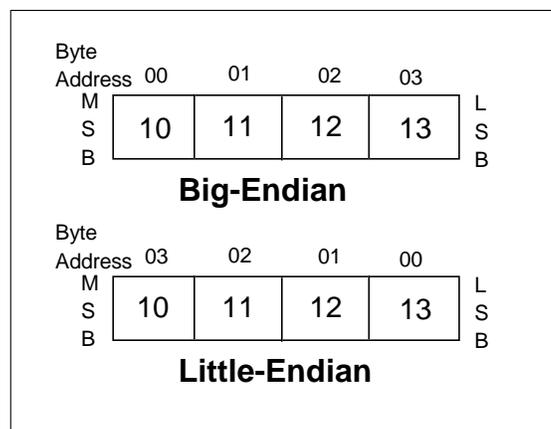


Figure 1. 4-byte word with LE and BE representations

Now, let's look at a more complicated case of a 64-bit double-word with multiple data elements.

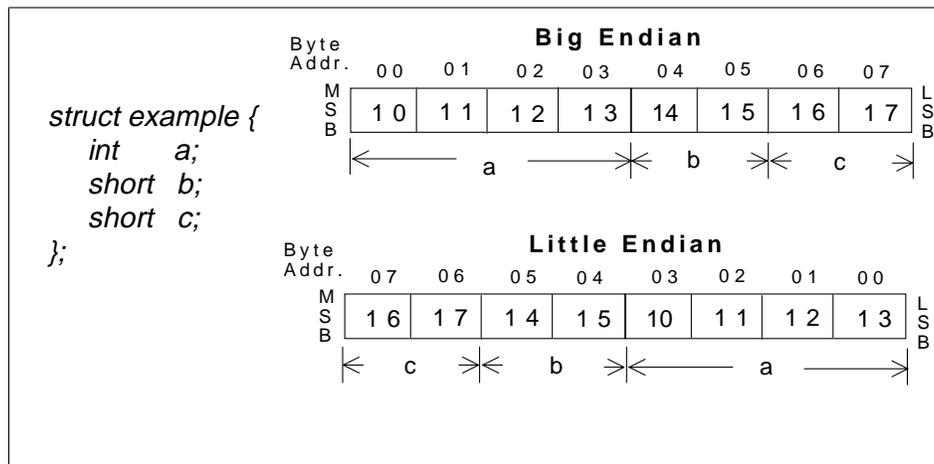


Figure 2. Data Elements with the Same Address in LE and BE Storage Models

In Figure 2, we summarize the byte ordering and addressing attributes of LE and BE storage models. In Figure 3, each data element has the same address in either LE or BE modes, e.g. variables a, b, c, d, e, and f, are located at the addresses 00, 08, 0C, 10, 18, and 1C, respectively.

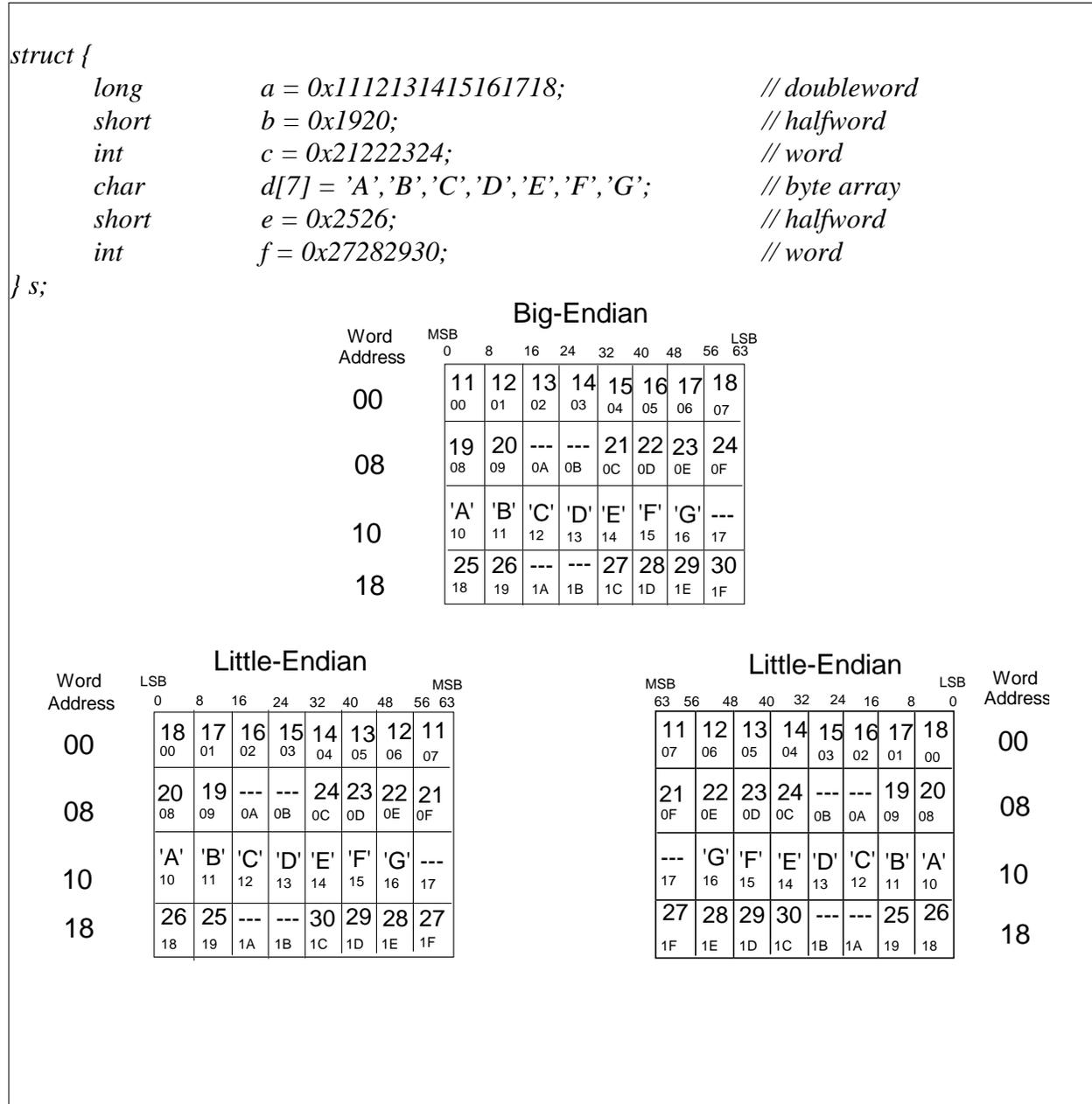


Figure 3. Data Structure in BE and LE

The byte order within a multi-byte data (e.g. variables a, b, c, e and f) in LE mode is in the reverse byte order of the data in BE mode. For a single byte data type, i.e. the character array *d[]*, the endianness is of no consequence - characters (or other single-byte data types) are at the same

byte locations in LE or BE mode. Note that long integers are 64 bits in Merced's 64-bit environment; and each data element is aligned at its natural boundary (at multiples of its data size), with padding if necessary, to avoid mis-aligned data accesses.

Also shown in Figure 3 are two representations of the LE data storage model. On the left, the addresses of individual bytes remain the same as that of the BE case by reversing the significance of the byte locations. The right presents the addresses in a reverse order keeping the significance of byte locations the same as that of the BE model. Typically, the right one is used to represent a LE storage model.

It is clear that byte ordering isn't an issue if a program that writes a word to memory then reads the same location as a word - it sees the same value if a variable is referenced consistently. But, if a program tries to read the same value one byte at a time (when a word is written), it may get different results depending on whether the processor is BE or LE - the bytes may appear to be "reversed" inside a multi-byte data.

6.0 Dealing with Endianness in Porting

In general, a program module is endian-neutral--the compilation will resolve the differences in byte ordering between LE and BE, if multi-byte data elements are not referenced as individual bytes (or its proper subset) or vice versa. On the other hand, using a union data structure; casting a data element; or manipulating bit fields can reference individual bytes in a multi-byte data element in a reversed byte order. This can be problematic when porting code between LE and BE machines. Another source of endian dependencies comes from sharing data across platforms. Typically, data are converted to a canonical data format for sharing to avoid endian dependency. Similarly, endianness problems result when interfacing with external devices (these can typically be solved by hardware and/or software solutions).

It is suggested in [1] that running “*lint*” or other programming aids can point out dubious uses of C construct. Other than that, there is no real magic in identifying and changing code automatically to correct endianness problems. In this section, several code examples are illustrated to show how to look for code patterns that can have potential endianness problems. In some cases, changes for making the code endian-neutral or endian-aware are suggested.

6.1 Data Type Mismatch

Data type mismatch is a major source for endian dependency. A data should be treated at run time as defined by its data type to avoid endian dependency problem. These can be discovered by inspection or lint. Using lint can indicate potential castings that are hiding endian dependency problems.

For example, a 4-byte integer should be treated by the processor as an indivisible data element of integer data type. The individual bits and bytes in the integer is viewed in opposite orders by LE and BE. Accessing the integer as a smaller unit can produce different results by LE and BE processors, as shown in the next code example.

```
int    a = 0x11121314;
char   b, *ptr;
...
ptr = (char *) &a;           // pointer ptr points to a
b = ptr[1];                 // b is 0x13 in LE and 0x12 in BE
...
```

The code is changed, as follows, to make it endian-aware for porting.

```
#define    BIG_ENDIAN        0
#define    LITTLE_ENDIAN    1
int       a = 0x11121314;
char      b, *ptr;
...
```

```

char endian() {
    short t = '0x0001;
    return *((char *) &t);           //return 00 for BE and 01 for LE
}
...
ptr = (char *) &a;                  // pointer ptr points to a
if (endian() == BIG_ENDIAN) b = ptr[1]; else b=ptr[2]; // b is 0x12 in BE and LE
...

```

In this case, the `endian()` routine is used to detect endianness at run time. This is necessary for a bi-endian machine which supports both LE and BE data storage models dynamically. However, a price in performance is paid for the flexibility in performing run-time checking. It is desirable in porting AIX to IA-64 that conditional compilation, by predefining data storage model of LE or BE, be used to avoid performance penalties at run time.

A better solution to the above problem would be to use an endian neutral solution.

```

/*
 * Macro to get bits 16-23 (in terms of the significance of bit positions) from
 * an integer value done by shifting the bits down to the zero bit position
 * and then anding the most significant 24 bits off.
 */
#define INTB16TO23(a) ((a >> 16) & 0xff)
...
b = INTB16TO23(a);           // b is 0x12 in BE and LE
...

```

In this case, since the code wanted bits 16-23 to be stored in `b`, using the shift operation lets the processor handle the multi-byte data as a unit and perform the appropriate operations.

Another case of endian dependency comes from referencing multiple data elements as a single, large data element, as illustrated in the next code example.

```

short a[2];
int b;
...
a[0] = 0x1112;
...
a[1] = 0x1314;
...
b = *((int*) &a[0]);          // b is 0x11121314 in BE and 0x13141112 in LE
...

```

Data type mismatch can also occur when using a union data structure. Union allows variables with multiple data elements having different types to share a common storage area to save memory. But, LE and BE can see different data value if the same storage area is used as one data type for writing and the other for reading or vice versa.

```
union short_or_int {
    short  a[2];
    int    b;
};
short c;
...
short_or_int.b = 0x11121314;
c = short_or_int.a[0];      // c is 0x1112 in BE and 0x1314 in LE
...
```

At issue, again, is that LE and BE address the short data *short_or_int.a[]* in the union in opposite orders, though the integer *short_or_int.b* remains the same in the two storage models!

6.2 Overlaid Data (with Bit Fields)

In some code, overlaid data, which has more than one data type in a data element, may be used to encode multiple pieces of information within a data element to save memory.

For example, it is typical that a descriptor data structure (in device drivers) uses a single word for both flags and a pointer. One possibility is that a descriptor word contains flags in the most significant byte and a 16-bit or 24-bit pointer (address) in the least significant bytes.

Another example is that a word, e.g. *int*, can be used to encode data for efficient use of memory space. The next code example shows a 4-byte word used for encoding personal data.

```
struct personal_data{
    unsigned age:7;           // age from 0 to 128
    unsigned height:7;       // height from 0 to 128
    unsigned gender:1;       // male / female
    unsigned pad:17;         // aligned to a word boundary
};
```

No changes are required if this structure is *not* used as an interface between entities that are different endianness and if the data are referenced by their appropriate tags. If the data is not referenced by the appropriate tag, but is instead referenced by overlaying another type, then problems similar to the previous section can occur. These are easily handled by using the real tag for the structure element or using a more generic method of referencing the data.

The problem with using the bit-field structure as an interface between entities with different endianness is that not only do the bit-fields need to be reversed, but the data units subdivided by bit-fields may need to be byte swapped to get a successful conversion.

To refer to the same encoded data in our example, the data structure needs to be changed, as follows, in porting from a BE to a LE machine.

```
struct personal_data{
    unsigned pad:17;           // aligned to a word boundary
    unsigned gender:1;        // male / female
    unsigned height:7;        // height from 0 to 128
    unsigned age:7;           // age from 0 to 128
};
```

After swapping the positions of the bit-fields within the structure, the whole integer would have to be byte swapped to get a value that could be used on the BE systems. The reversals are required due to the way our compilers generate the bit packed structure. The gcc compiler on a LE platform packs from LSb to MSb. The gcc compiler on a BE platform and the AIX compiler generate from MSb to LSb. So, in this example, we reverse all the fields so that the LE compiler while packing from LSb to MSb generates a structure that is packed from MSb to LSb like the BE compiler.

When looking closer at this code example, however, you'll note that the reason for the code change in porting is for sharing data in between LE machine and BE machines - data stored by a machine is shared by an opposite endian machine. The code does not need be changed in porting to AIX if the "personal_data" is not stored by PowerPC for IA processors or vice versa! Though bit fields are stored and referenced in opposite orders on a LE and a BE machine, they should not care if each bit field is addressed as defined. Note that one exception is when a bit field is used to map data for interface purposes and that interface requires a certain data format.

6.3 Endianness in Exchanging or Sharing Data

Another class of endian problems is code that is used as an interface between systems of varying endianness; for example, device drivers that use BE data that is byte swapped to LE data for use with a LE device/adaptor. Also, the TCP/IP/UDP/RPC protocols require that data be sent in "network byte order" or BE. These are examples of canonicalizing data. The idea is that an interface exists that has defined byte ordering for the data used by that interface. In the device driver case, the PCI device defines LE as the canonical form in most cases. The networking code has chosen BE as its canonical form.

In general, this is a problem for communicating between devices (device driver to device/adaptor interactions), network communications, or applications that share data that is granulated in chunks bigger than a byte. For example, a database, stored binarily, shared between an AIX

PowerPC (BE) system and a Monterey IA-64 (LE) system could have problems because the data would not be byte swapped appropriately for one of the systems.

These problems are typically handled by the application or data sender canonicalizing the data. The data sender usually performs some operations on the data to convert the data to the canonical form and then sends the data. The data receiver reads the data and performs some operations to convert the data from canonical form to a usable form. In the case of the networking code, the data receiver may be either LE or BE.

For example, IPv4 addresses and TCP port values in the TCP/IP header are manipulated as 32-bit unsigned integers and 16-bit unsigned integers, respectively. These need to be in host endianness (meaning the endianness that the system is running under) because math operations are performed on them. These need to be converted to BE before sending and must be converted from BE when receiving. To do this conversion, the code defines a set of routines that take a value and convert from host endianness to BE form or vice versa. In fact, these are POSIX defined. They are the `htonl`, `ntohl`, `htons`, and `ntohs` routines (s refers to short and l refers to a 32-bit quantity). In AIX, these are macros that pass through and do nothing. In porting to IA-64, these routines need to swap data bytes because the host is now in LE! Note that endianness is usually referred to as *order* in the documentation for these routines.

The application programs must choose their own canonical form, decide that data will not be shared, or provide utilities to convert between the forms. XDR (eXternal Data Representation) is one of the protocols that provide a canonical data format for sharing data across heterogeneous systems. At this time, there is no plan for sharing system data between an AIX BE system and an AIX LE system.

Endianness issues in interfacing to an external I/O device can be resolved by various hardware and software solutions - which complicates the solutions in porting device drivers from a PowerPC to a IA system. Though PowerPC supports the BE data storage model, its I/O busses, whether IBM's own MicroChannel or the latest PCI, are LE-based. In RS/6000 systems, the I/O controller, which is the bridge between system bus and its I/O busses, provides a data steering function to convert data from LE to BE (and vice versa) in reading from or writing to a device. This data steering function is applied to both DMA (Direct Memory Access) and PIO (Program I/O) or MMIO (Memory-Mapped I/O) data. In essence, the I/O controller treats data as byte streams such that byte 0 in the system goes to byte 0 in I/O, byte 1 to byte 1, and so forth. This brings up an interesting scenario - bytes in multiple-byte data need to be swapped before being passed to the I/O! In porting to IA-64, these multi-byte data elements need to be "un-swapped"!

In AIX, several PIO macros are provided to reverse the byte order of the data used when interfacing with a BE device. (Most of the I/O devices are LE based; while there are BE based devices, the bus is LE.)

To port AIX to Merced, it is suggested that endian-aware routines/macros be defined to consolidate the driver-to-device interface. These routines/macros will present data in the correct format as governed by endianness of its device, I/O bus, and host.

Most of the endian dependencies in code result from referencing data differently from its natural, defined data type. For example, addressing internal bytes of a multi-byte data element; referencing multiple data elements as a single, large data element; packing multiple bit field in a data element; casting data; and using union data structure can potentially produce erroneous results and/or crash the system when the code is ported to an opposite-endian platform.

Another source of endian dependency is in assuming the endianness of the run-time platform. For example, AIX assumes that its processor supports the BE storage model. In interfacing with its PCI devices, which support the LE data storage model, AIX may call endian conversion routines to reverse the byte order in a data structure.

When porting to an opposite-endian platform, recompilation typically generates code with correct byte ordering. However, it is likely that some code with endian dependency will need to be identified (by inspection or using programming aids such as “*lint*”) and changed manually before all endian related issues can be resolved. In porting AIX code, the machine-independent

- Packing bit fields into a single data element can be problematic if the data needs to be stored to a persistent storage device shared by an opposite-endian machine. But, it is not an issue if the data is not shared between BE and LE machines. The internal locations of bit fields in a data element are of no consequence between the two data storage models if the bit fields are referenced as defined. Though code will work correctly (in its endianness), comments associated with code may need to be changed to present the internal bit patterns in a reversed order.

7.0 References

- [1] Steven Zucker, Endianness in Solaris, SunSoft Report, Feb. 1998
- [2] Martin Hopkins, Endian Issue Recommendations, IBM Academy Report, Mar. 1995
- [3] James R. Gillig, Endian-Neutral Software, Part 1, Dr. Dobb's Journal, Oct. 1994
- [4] James R. Gillig, Endian-Neutral Software, Part 2, Dr. Dobb's Journal, Nov. 1994
- [5] Cathy May et al, The PowerPC Architecture, Appendix D: Little-Endian Byte Ordering, Morgan Kaufmann Publishers, May 1994.
- [6] Apple/IBM/Motorola, PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture, Appendix C: Bi-Endian Design, Nov. 1995
- [7] RS - 64 Bit Runtime Architecture and Software Conventions for IA-64, Chapter 4 - Data Representation, Intel Internal Document, Ref. No. SC-2135.
- [8] IBM Corp., AIX Kernel Extensions and Device Support Programming Concepts
- [9] IBM Corp., POWERstation and POWERserver: Hardware Technical Information General Architecture