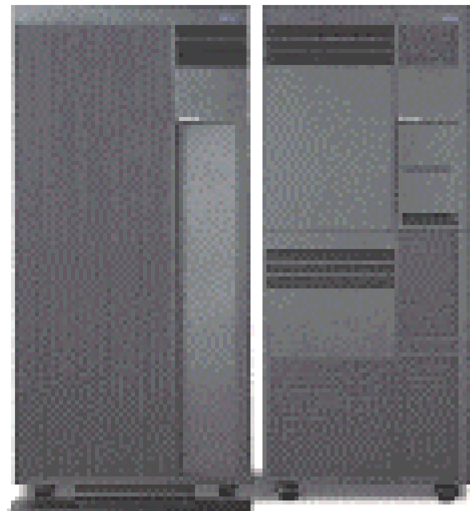
Migrating C and C++ Applications to AIX 5L on IA-64

Partners in Development
September, 2000



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 IBM's intellectual property rights may be used instead.

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 suppliers of those products. IBM may have patents or pending patent applications covering subject matter in this presentation. Furnishing 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 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 5L, AIX/6000, C Set++, CICS, CICS/6000, DB2, ESCON, IBM, LANStreamer, LoadLeveler, Magstar, MediaStreamer, 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. SCO is a registered trademark of Santa Cruise Operations, Inc. Sequent, NUMA-Q, DYNIX/ptx, and ptx are registered trademarks of Sequent Computer Systems, Inc., a wholly owned subsidiary of IBM.

Contents

CONTENTS 1

INTRODUCTION 2
Helpful Terms and Definitions 2

PART 1. MIGRATING AIX APPLICATIONS TO AIX 5L ON IA-64....3
Introduction 3
General Differences Between AIX 4.3 and AIX 5L on IA-64....3
AIX System calls for Binding....6

PORTING AIX 32-BIT DEVICE DRIVERS TO AIX 5L ON IA-64....9
IFDEFS....9
Driver Configuration....9
Memory Mapped I/O 12
Other Kernel Services and Considerations 19

DEALING WITH ENDIANESS....24
Byte Ordering 24
Porting 26
Overlaid Data (with Bit Fields)....28
Exchanging or Sharing Data 29

CONVERTING 32-BIT APPLICATIONS TO 64-BIT APPLICATIONS....31
Overview....31
C and C++ Data Type Size and Alignment Issues 32
Explicit Cast Improperly Applied....33
Pointer to an int Icompatible with a Pointer to a long....34
Lack of Prototyped Function Declarations in Scope of Call Statements....35
Integer Expression with Potential Overflow Is Converted to a long....37
Untyped Integral Constants Are *int* by Default....38
Sizeof(void *) != Sizeof(int)....39
Truncation of a 64-bit Pointer Value When Converted to a Smaller Integral Type....40
Assumption That Pointers and int Are Same Size in Arithmetic Context....41
Pointer Return Type or Argument Types in the Absence of a Prototyped Function Declaration 41
Objects Change Size....43
Integer Constants 44
Stack Layout Changes due to Larger Data Elements 45

PART 2. MIGRATING DYNIX/PTX PROGRAMS TO AIX 5L ON IA-64....46

MIGRATING C APPLICATIONS 46
C Language Incompatibilities 46

MIGRATING ASSEMBLY LANGUAGE PROGRAMS....60

MIGRATING C++ PROGRAMS....60
C++ Compiler Differences....61
C++ Class Libraries 64

PART 3. REFERENCES....65

Introduction

It our intent to facilitate an easy migration of applications from AIX V4.3 and DYNIX/ptx to AIX 5L on 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.

Thanks to the following for their contributions: Casey Cannon, Thomas Chen, Mike Lyons, Randy Swanberg, Cynthia Sax, Don Wood, Mike Day, Scott Porter, Donald Stence, Bill Buros

Helpful Terms and 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.

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

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

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

AIX 5L on IA-64
: The New AIX version 5 operating system that will run on POWER and Itanium based Systems

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

Part 1. Migrating AIX Applications to AIX 5L on IA-64

Part 1, Migrating AIX Applications to AIX 5L on IA-64, provides information on general differences between AIX 4.3 on POWER and AIX 5L on IA-64, porting AIX 32-bit device drivers AIX 5L on IA-64, byte ordering, and endianess in porting.

> ***Developer Tip***
> If you are interested in migrating your application to 64-bit, first move your code to 64-bit on AIX. Then, migrate the 64-bit code from AIX on POWER to AIX 5L on IA-64.

Introduction

AIX on the PowerPC addresses data in Big Endian (BE) order, while traditional Intel processors address data in Little Endian (LE) order. In porting AIX to IA-64, we chose to maintain the Intel LE byte ordering. As a result, application source that depends on byte ordering and runs successfully on AIX/POWER may not run successfully on AIX 5L on IA-64 without modifications to eliminate that dependency. 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. When this is not the case, changes to the source may be required.

Note that some functionality available on AIX on the POWER may not be supported on AIX 5L on IA-64 in its first release. An exhaustive list is not available at this time, but the following is known to be missing:

- 3D Graphics Support
- DCE
- SOM/DSOM
- JVM 1.1.8 (the JVM will be 1.3)

General Differences Between AIX 4.3 and AIX 5L on IA-64

Alignment Differences

Double on AIX/POWER (32 and 64 bit) has a size of 8 bytes, but is aligned at 4 byte boundary. For ILP32 the size is 8 bytes and the alignment is also 8 bytes.. This may cause a structure containing "double" to have different size and mapping characteristics on AIX/POWER than on AIX 5L on IA-64. See table below

			AIX/POWER 32-bit	**AIX 5L - ILP32**
struct {				
	int	i;	# size 4; alignment 4	size 4; alignment 4
	long long	j;	# size 8; alignment 8	size 8; alignment 8
	long	k;	# size 4; alignment 4	size 4; alignment 4
	float	m	# size 4; alignment 4	size 4; alignment 4
	double	n	# size 8; alignment 4	size 8; alignment 8
	long double	p;	# size 16; alignment 16	size 16; alignment 16
	}			

ILP32 pplications that wish to have the same structure mapping as for IA-32 can use the following pragma:

```
#pragma align=ia64unix386
```

This pragma will force the following size an alignment rules for ILP32:

```
struct {
        int          i;     # size 4; alignment 4
        long long    j;     # size 8; alignment 4
        long         k;     # size 4; alignment 4
        float        m      # size 4; alignment 4
```

```
double          n       # size 8; alignment 4
long double     p;      # size 16; alignment 4
}
```

Default for *char*

The default compilation mode for *char* on AIX/POWER is *unsigned*, while on AIX 5L on IA-64 it is *signed.* AIX applications that need the default behavior for *char* on Itanium to be *unsigned* can use the ***-qchars=unsigned*** compile-time flag to accomplish the same desired effect.

Floating Point Support

There are two different extended precision on the table, 128 bits and 80 bits. In PPC/POWER platform, long double is a type of a software implementation which takes a storage of two doubles (128 bits). In ILP32 (Intel) platform, long double is 80 bit long and is hardware supported. Because 80 bit long double is hardware supported in IA64, it is very valuable to have extended range and precision without compromising the performance. Although PPC/POWER version of long double is more precise than IA64's natural extended precision, it is slower and has smaller range of values. The default size for long double when using the VA 5.0 C and C++ compiler is 64-bits. In order to get 80 bit extended precision, the –qlongdouble=80 compiler flag.
Some floating point APIs which are supported by the AIX kernel on POWER will not be supported on IA-64. These are:

- fp_trapstate
- fp_fpscrx
- fp_trap
- fp_flush_imprecise
- fp_raise_xcp
- fp_iop_snan
- fp_iop_infsinf
- fp_iop_infdinf
- fp_iop_zrdzr
- fp_iop_infmzr
- fp_iop_invcmp
- fp_iop_sqrt
- fp_iop_convert
- fp_iop_vxsoft

There are also other differences such as range, resolution, NANs and infinities.

nlist/knlist

knlist on AIX 5L for IA-64 takes an nlist64 structure. This is necessary since the IA64 kernel exists above 4GB of addressability.

Module Format

AIX 5L on IA-64 supports the UNIX Sytem V.4 generic ABI for IA-64. Shared library creation and usage as well as static and dynamic linking and loading of applications is different than what is supported on AIX for POWER. Both the module and debugger format are different as follows:

	AIX/POWER	AIX 5L IA-64
Module Format	XCOFF	ELF
Debugger Format	stab strings	DWARF2

Information on the generic IA-64 ABI can be found at the following URL: http:www.sco.com/developer/gabi

Import/Export file Support

As an accomodation for existing AIX applications, the AIX 5L on IA-64 linker will provide support for the use of Import and Export files. This support is different from the support on AIX/POWER in both Syntax and Semantics which may require makefile changes. The command line arguments to ld are:

```
-B import:file
-B export:file
```

Import/Export Syntax

Directives begin with a '%', which must be in column one. There is currently one directive, "%soname". The directive is followed by a string (quoted or not):

```
%soname name
```

This directive has the effect of adding the named shared object to the needed list of the object being built. (The needed objects of a dynamic executable or shared object can be listed with the -Lv option of the dump command.) The comment character is '#'. It may occur in any column. The comment goes to the end of line.

Symbols should be listed one per line. Symbol entries may contain an optional tag which may have an optional number:

```
sym_name [{ws} tag [{ws} decimal_number]] {eol}
```

where tag is one of syscall, function, object, and decimal_number is a system call number for the syscall tag and a size for an object. {ws} stands for white space, either space or tab; {eol} stands for end-of-line.

In an export file, symbols with the syscall tag will be marked for export by the system loader as system calls. All symbols not marked for export will not be visible outside of the shared object defining them. The %soname directive is ignored in an export file. Multiple import files can be specified on the ld command line, but only one export file.

The -Bsyscall option is still present and functioning, but its functionality is now additionally obtainable through export files. The -Bsyscall option will be depricated in the future.

libld.a Support

Since the module format on AIX 5L on IA-64 is ELF, the XCOFF specific processing library **libld.a** will not be ported. Instead, **libelf** will provide the object file specific processing code for the ELF environment.

Libraries and Archives

On AIX/POWER, 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 AIX 5L on IA-64. The System V.4 semantics for shared library creation, static and dynamic linking, does not allow for a single archive to be used in the same manner as on AIX/POWER. For AIX 5L on 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. The linker's implementation determines its target mode (ia32, ia64l32, ia64l64).

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 an ELF file of the wrong type is encountered in the search. Instead the link-editors will 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.

Symbol Resolution

On AIX 5L on IA-64, symbol resolution in a running program is performed by the runtime linker, according to the rules specified in the generic IA-64 ABI (gABI). At link time, shared objects referenced on the command line are listed in the "needed module" section of the program. Symbols are resolved at runtime by sequentially searching the "needed modules" list until a definition is found.

On AIX, symbol resolution is performed at link time. If a symbol is defined in a shared object (or "module") referenced on the command line, the linker records the symbol name and defining module in the loader section of the program. When a program is executed, imported symbols must be resolved by finding them in the defining module, as recorded at link time.

On both platforms, an import file can be used in place of a shared object (module) that is not available or has not yet been built.

The inability to associate a symbol to a specific library on AIX 5L on IA-64 makes the order in which the libraries are specified very important. For instance, with respect to **rpc**, if you want the streams behavior, your library order must be **libnsl**, then **libc**, and if you want sockets behavior, it must be **libc** followed by **libnsl**. For example:

1. Module "A" contains a definition for function "X"
2. Library "foo" also has a definition for a different function "X"
3. Library foo has a function "Y" which calls function "X"

On AIX/POWER , function "Y" will get the "X" present in library "foo".
On AIX 5L on IA-64, function "Y" will get the "X" present in Module "A".

AIX System calls for Binding

The AIX **load(), loadquery() and unload()** APIs will continue to be supported on AIX 5L on IA-64. However, **loadbind()** will not be supported.

Linking and *ld* flags

The AIX linker differs from the SVR4 linker (used in AIX 5L on IA-64) in features and functionality. The flags used as well as their default behavior are often different. The AIX 5L on IA-64 linker ***will not support relinking of an executable***. Some of the linker flags on AIX may have implied semantics when linking on AIX 5L on IA-64, others have coresponding flags, and the rest are not supported. The following table shows the AIX linker flags and their semantics with the corresponding AIX 5L on IA-64 linker flags and semantics:

AIX/POWER Linker Flag	AIX 5L/IA-64 Equivalent Flag
-DNumber locates the initialized data	Possible with the Map File (-M option)
-eLabel sets the entry point to Label	-e *epsym* sets the entry point to epsym

AIX/POWER Linker Flag	AIX 5L/IA-64 Equivalent Flag
-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)
-lName processes libName.a	-l x
-LDir adds Dir to the list of search directories	-L *path* *Note: -L does not record the path in the binary*
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 library 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 any unresolved external references	-z *defs* do not allow undefined symbols (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 *path* 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 *text* 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

AIX/POWER Linker Flag	AIX 5L/IA-64 Equivalent Flag
-bnso -bI:/lib/syscalss.exp	-a produces a statically linked executable file (must not be used with -r and -dy options)

The following AIX flags either have no meaning on AIX 5L on IA-64 or have no equivalance. Encountering these flags on the ld command line may either cause an error or be ignored:

-k -z -basis -bautoexp -bbigtoc -bbindcmds:File -bbinder:File -bbindopts:File -bcomprld -bcrld -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 -bnocrld -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

Note: Both the Visual Age V5.0 Compiler and the static linker "ld" have a –B option. If you have *makefiles* that use the cc driver to invoke the linker, you must pass the –B options to the linker as follows*:*

-Wl,"-B…"

where the letter following the W is an el, and the … is a suboption of the –B. For example, to use the linker's –Bexport option you specify:

-Wl,"-Bexport:file"

The quoting of the string is only to keep any space delimiters from indicating the next argument to the compiler.

Porting AIX 32-bit Device Drivers to AIX 5L on IA-64

This section provides information on how to take a 32-bit PPC AIX driver and migrate it to AIX 5L on IA-64. It is intended to provide guidance to AIX 5L on IA-64 bring-up driver writers and to input to the DDK development.

IFDEFS

By convention, a number of #defined values can be used in controlling device driver and kernel extension source. **#if __ia64** is used to distinguish compilations targeted at the AIX 5L on IA-64 architecture. 64-bit compiles of kernel extensions and device drivers will define **#if __64BIT_KERNEL** and **__64BIT__.** Therefore, for single-source device drivers that generate a 32-bit PPC AIX device driver and a 64-bit device driver (AIX 5L on PPC or IA-64), the differences can be controlled with these ifdefs. Finally, **_POWER_MP** continues to be used to identify a compilation directed toward a multiprocessor environment. This can be used for single-source device drivers that generate a 32-bit PPC AIX UP device driver, and for 32-bit AIX MP and 64-bit AIX 5L on POWER and IA-64 device drivers (which are required to be MP-safe).

Driver Configuration

Configure Methods

There are two issues to consider. First, the config methods need to be made endian-safe. Second, config methods need to handle the 64-bit kernel implications.

Currently, the plan is to continue to have the existing 32-bit application configuration methods (modified for any endianess problems) work as is, and to load the 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 devnos in the device's DDS, because the 64-bit device driver implicitly assumes a dev_t is 64 bits. Ideally, since the device fileset will be impacted and shipping^^**confused?**^^ 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 a 64-bit devno. On an IA-64 system, the method would always use a 64-bit devno. On a PPC system, the method would have a runtime 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;

#ifdef __ia64
        major = genmajor64();
        minor = genminor64(major);
        dds.devno = makedev64(major, minor);
#else
                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);
        }
#endif
```

All ODM information will stay the same and thus is not impacted. For AIX 5L on IA-64, there is a possibility that some PdAt attributes (such as describing bus memory space requirements that are already detectable via PCI configuration mechanisms) will no longer be required.

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 groundwork 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 that there are existing application interfaces which 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 t **dev_t,** interfaces that surface devnos 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 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	R	Major Number	Minor Number
	S		
	V	3	3
63	62	61 2	1 0

V = Version bit
0 = Old Style 32-bit devno (Major:31-16; Minor 15-0)
1 = New Style (Major:61-32; Minor 31-0)
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 *makedev* 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 /* end of else __64BIT_KERNEL */
```

All 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, but continues to call **genmajor()/genminor()** and passes

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 2^16th major or minor number limit.

Memory Mapped I/O

Probably the most significant driver-visible kernel service change between 32-bit POWER and AIX 5L on IA-64 concerns memory-mapped I/O.

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 (AIX 5L on POWER or IA-64).

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 discontiguous 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.

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. **io_map_init** 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 cannot 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. Although the **io_map.size** field is specified in number of bytes, the underlying mapping actually occurs in terms of virtual pages (0x1000 bytes). So if the initial mapping is for 4 registers in the bus address region to be mapped into virtual page number zero, any subsequent mapping into the same segment must be to a virtual page number greater than zero—you cannot map a second set of 4 registers into a different byte range of virtual page zero.

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 true because 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 successfuly **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 addressibility to all the mappings.

For example, the following creates an I/O segment and maps three 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)
        dd_backout(..);

/*
 * 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)
        dd_backout(..);

/*
 * 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)
  dd_backout(..);
```

Attach to an I/O Mapping

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

This service sets up addressibility 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 **0x300000**00. The driver had to add in the virtual page offset (PAGESIZE*1, or 0x1000 bytes) 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.

Detach from an I/O Mapping

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

This service removes addressibility 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.

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 POWER driver, and 64-bit POWER and IA-64 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 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 addressibility 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 addressibility 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
```

AIX 5L on IA-64 Bus I/O Space

The AIX 5L on IA-64 architecture presents two additional differences that device driver writers must handle. First, there is only 64K (16 bits of addressing) of platform bus I/O space for the entire system (similar to IA32 environments today). This is in contrast to CHRP systems, where there is 32-bits of bus I/O space per PHB (PCI Host Bridge). ISA I/O space on AIX 5L on IA-64 is within this single shared 64K space (while in CHRP, it is within the low 64K of its parent PHB's I/O space). The net of this for AIX 5L on IA-64 is that I/O space is a much more limited resource, and device drivers should use it only when absolutely required. Note it is TBD whether there will be a mechanism for a driver package to communicate to system configuration utilities that it does not need an I/O space assignment, despite the value read from the associated PCI Base Address Register. Regardless, driver writers should use Bus Memory space in favor of Bus I/O space whenever possible.

The second impact from the AIX 5L on IA-64 I/O space model concerns how it is memory-mapped from the perspective of the driver executing on the CPU. Like PPC, I/O space is memory mapped (i.e., it is accessed via a virtual address returned by io_map). However, the virtual address space is discontiguous with respect to the physical 64K address space, spreading 4 ports per page across a 64 MB virtual address space (see AIX 5L on IA-64 PRM for more details). What this means to device driver writers accessing bus I/O space (PCI or ISA) is that an address transformation is required. The following illustrates how to initialize a handle to access the command register block for the primary channel of an IDE controller in legacy mode (8 registers, starting at address 0x1f0 of ISA I/O space, 1 register starting at 0x3f6):

```
/*
 * setting up the io handle for the ide command block registers for
 * primary channel
 */
```

```
        struct io_map       iom;
        /* io map structure(ioacc.h)*/

        iom.key     = IO_MEM_MAP;
        iom.flags   = 0;
        iom.busaddr = (long long)IA64_IOPORT(0x1f0);
        /* start of register block */
        iom.bid     = BID_ALTREG(dds.bus_bid, ISA_IOMEM);
        iom.size    = IA64_IOPORT_RANGE(0x1f0, 8);
        /* 8 registers in this block */
        ioh = io_map_init(&iom, 0, NULL);

        if (ioh == NULL)
              dd_backout(...);

        iom.busaddr = (long long) IA64_IOPORT(0x3f6);
        /* start of 2nd register block */
        iom.size = IA64_IOPORT_RANGE(0x3f6, 1);
        /* 1 register in this block */
        aioh = io_map_init(&iom, 2, ioh);
        /*  first mapping covers virtual page numbers
        *   zero and one, 4 bytes starting at 0x01f0 an
        *   4 bytes starting at 0x11f4.  This mapping will
        *   be on virtual page two, at 0x23f6.*/
```

The AIX 5L on IA-64_IOPORT and IA64_IOPORT_RANGE macros will be provided in ioacc.h. For illustrative purposes, the possible system definition of the macros could be:

```
#define IA64_IOPORT(port)      (((port & 0xFFFC) << 10) | (port & 0xFFF))
#define IA64_IOPORT_RANGE(port, num_regs) \
                    (IA64_IOPORT(port+num_regs-1)—IA64_IOPORT(port) + 1)
```

Note that it has not been completely clarified whether ports will appear in the first 4 bytes of a page, or at their natural page offsets. The above assumes the latter; if this is not true on the actual hardware, minor adjustments to the macros will be needed, but the io_map_init call made by drivers will be the same.

Continuing on, the following example shows one way the IDE cylinder low register, located at 0x1f4 of ISA I/O space in legacy mode, could be accessed. This illustrates how driver writers might need to adjust the register offsets used from the start of register blocks to accomodate the discontigous addressing in AIX 5L on IA-64:

```
#ifdef __ia64
#define CYL_LOW_OFFSET  0x1004
#else
#define CYL_LOW_OFFSET  0x4
#endif
        volatile char * eaddr;
        eaddr = io_map(ioh);
        eaddr += 0x1f0;
        /* add in the page offset for the start of the register block */
              data = *((volatile eaddr *)(io + CYL_LOW_OFFSET));
        /*  add in the offset from start of the register block */
```

Finally, the following illustrates a second kind of problem driver writers might encounter in migrating from PPC memory-mapped I/O to the discontiguous mapping of I/O space in AIX 5L on IA-64. In this case, structure offsets need to be adjusted:

```
struct at_cmd_block {
```

```
        char   data;
        char   error;
        char   sec_cnt;
        char   sec_num;
#ifdef __ia64
        char   pad[0x1000];
        /* adjust for discontiguous ports */
#endif
        char   cyl_low;
        char   cyl_hi;
        char   drv_head;
        char   status;
};
               volatile struct at_cmd_block *reg_pointer;
        eaddr = (caddr_t) io_map(ioh);
        eaddr += 0x1f0;
        /* add in the page offset for the start of the register block */
        reg_pointer = (volatile struct at_cmd_block *) eaddr;
               data =  reg_pointer->cyl_low;
```

There are additional coding techniques that could be used for I/O space register access on PPC that will need modification on AIX 5L on IA-64, but the above should indicate the kind of adjustments necessary.

REALMEM_BID

The **REALMEM_BID** will still be supported in the 64-bit environment, but it will only be supported by **rmmap_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**.

Unaligned/String/Multiple Accesses to I/O

The **64-bit PPC** architecture does not guarantee atomicity of access when performing unaligned, string, or multiple operations. While **AIX 5L on IA-64** does not provide string or multiple operations, it appears to still be subject to the same issues on unaligned accesses. There are two issues with this when dealing with these operations to I/O (non-system memory) 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 cannot handle the broken up access or because the 4-byte data store is not atomic.

The second issue deals with "stuttering," or reissuing 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 PPC 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" that 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 PPC kernel alignment handler will recognize attempted unaligned accesses to I/O space and crash the system. The AIX 5L on IA-64 kernel, on the other hand, will not detect that the unaligned access is to I/O space, and will not crash the system.

As a programming guideline for device driver writers, it is recommended that drivers avoid unaligned accesses to I/O space (and, where available, similarly avoid string or multiple operations to I/O space). This is a much simpler and

safer approach than determining whether the device can correctly handle the way advanced CPU architectures can break up and/or repeat these kinds of operations.

Other Kernel Services and Considerations

DMA

For the most part, for mainstream device drivers , there is currently 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 *.**

There are two AIX 5L on IA-64 design points that are not yet closed, which potentially could have a minor impact on DMA services interfaces. Additional information will be provided as soon as the final design is settled.

1. How handling contiguous physical addressing requirements by devices will be handled (e.g., rmalloc)
2. How to handle devices capable of only addressing 32 bits in systems with greater than 4GB of memory, given the lack of TCEs in the AIX 5L on IA-64 platforms.

There is no support for pluggable ISA cards in AIX 5L on IA-64, and hence no support for ISA bus masters. Also, the dreq_active interface is no longer supported.

If there were any drivers using the **xmemdma()** service (which is doubtful), they will find out that **xmemdma()**does not 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.

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.)

EPOW

The EPOW registration and processing remains unchanged. Note that whether or not the driver's registered EPOW handler ever gets called depends on platform capabilities. The underlying implementation may differ drastically from PPC systems, but the abstraction presented to device drivers remains the same.

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(),** grows 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. Thus, 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 concerned the ability to 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.

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
     }
```

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 (PPC) 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..

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.

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 addressibility 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.

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.

Power Management

The initial release of AIX 5L on IA-64 will not support device driver power managment. It is unclear what changes to the 32-bit POWER device driver power management framework (*pm_register_handle*, *pm_planar_control*, etc.) will eventually be needed on IA-64. For the initial porting of the POWER device drivers to AIX 5L on IA-64, it is recommended that the power management (PM)-specific functions of the driver be migrated as well. It is acceptable to initially hide the PM-related code behind IFDEFs, to simplify testing issues until the underlying framework is more developed.

Linking/loading

No device driver-visible changes have yet been identified in kernel extension linking and loading. It is possible that some minor build environment changes will be needed, such as a modified way to describe a kernel extension’s exported symbols. This will be documented when the design is completed and approved.

Packaging

AIX 5L on IA-64 packaging details are still be determined. Support for the traditional package layout in an installp format will be provided (it still needs to be determined how the changes for 64-bit POWER packaging affect AIX 5L on IA-64). Alternative packaging formats (such as UDI) will also be supported, but should not be used in migrating

existing POWER AIX device drivers. Additional packaging details will be documented as the design is completed and approved.

Dealing with Endianess

Byte Ordering

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

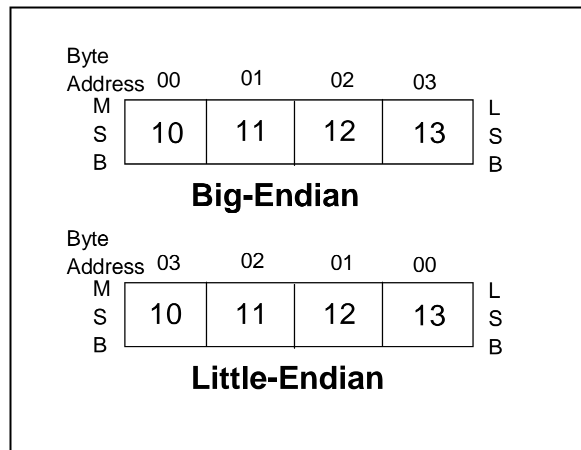


Figure 1. 4-byte word with LE and BE respresentations

Now, let's look at a more complicated case of a 64-bit double-word with multiple data elements. In Figure 2, variables a, b, and c all are located at the same addresses 00, 04, and 06, respectively, in both LE and BE cases. Data elements and the individual bytes within a data element are stored in a consistent left-to-right order for BE. For LE, on the other hand, data elements are stored in a right-to-left order; while data bytes within a data element remain in a left-to-right order. This can be confusing, particularly in reading the memory dump on a little-endian machine.

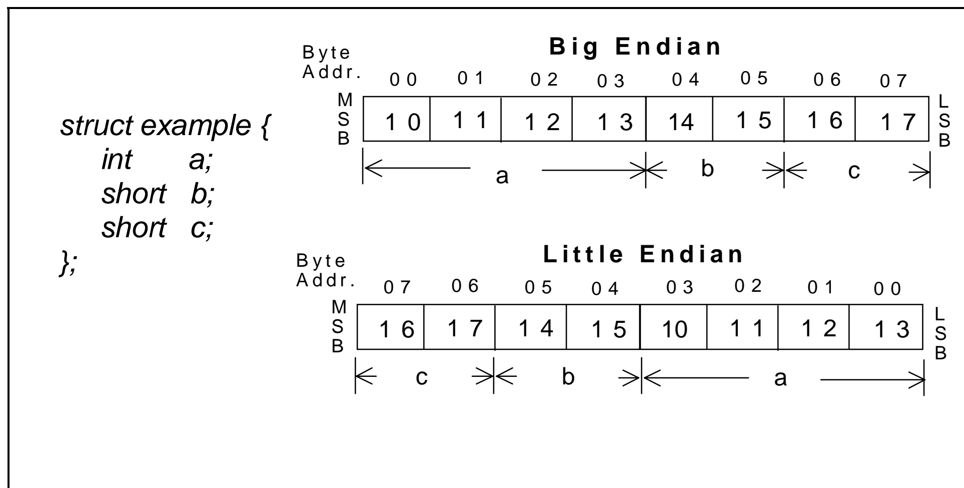


Figure 2. Data Elements with the Same Address in LE and BE Storage Models

Figure 3 summarizes 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.

```
struct {
long        a = 0x1112131415161718;            // doubleword
short       b = 0x1920;                        // halfword
int         c = 0x21222324;                    // word
char        d[7] = 'A','B','C','D','E','F','G'; // byte array
short       e = 0x2526;                        // halfword
int         f = 0x27282930;                    // word
} s;
```

Big-Endian

Word Address	MSB 0	8	16	24	32	40	48	56 LSB 63
00	11 00	12 01	13 02	14 03	15 04	16 05	17 06	18 07
08	19 08	20 09	--- 0A	--- 0B	21 0C	22 0D	23 0E	24 0F
10	'A' 10	'B' 11	'C' 12	'D' 13	'E' 14	'F' 15	'G' 16	--- 17
18	25 18	26 19	--- 1A	--- 1B	27 1C	28 1D	29 1E	30 1F

Little-Endian

MSB 63	56	48	40	32	24	16	8 LSB 0	Word Address
11 07	12 06	13 05	14 04	15 03	16 02	17 01	18 00	00
21 0F	22 0E	23 0D	24 0C	--- 0B	--- 0A	19 09	20 08	08
--- 17	'G' 16	'F' 15	'E' 14	'D' 13	'C' 12	'B' 11	'A' 10	10
27 1F	28 1E	29 1D	30 1C	--- 1B	--- 1A	25 19	26 18	18

Little-Endian

Word Address	LSB 0	8	16	24	32	40	48	56 MSB 63
00	18 00	17 01	16 02	15 03	14 04	13 05	12 06	11 07
08	20 08	19 09	--- 0A	--- 0B	24 0C	23 0D	22 0E	21 0F
10	'A' 10	'B' 11	'C' 12	'D' 13	'E' 14	'F' 15	'G' 16	--- 17
18	26 18	25 19	--- 1A	--- 1B	30 1C	29 1D	28 1E	27 1F

Figure 3. Data structure in BE and LE

The byte order within a multibyte 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 Itanium'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 misaligned data accesses.

Figure 3 also shows two representations of the LE data storage model. On the left, the addresses of individual bytes remain the same as those in 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 is not 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. If a program tries to read the same value one byte at a time (when a word is written), however, it may get different results depending on whether the processor is BE or LE—the bytes may appear to be "reversed" inside a multibyte data.

Porting

In general, a program module is endian-neutral—the compilation will resolve the differences in byte ordering between LE and BE, if multibyte 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 multibyte 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.

Data Type Mismatch

Data type mismatch is a major source for endian dependency. A data should be treated at runtime as defined by its data type to avoid endian dependency problems. 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 are 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=prt[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 runtime 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 runtime.

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 multibyte 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 values 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, although the integer *short_or_int.b* remains the same in the two storage models.

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 in endianness and if the data is referenced by the 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.

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/adapter. 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/adapter interactions), network communications, or applications that share data that is granulized in chunks bigger than a byte. For example, a database, stored binarily, shared between an AIX POWER (BE) system and AIX 5L on 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 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 buses, 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 multibyte 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 Itanium, 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 multibyte 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 runtime 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 code should be changed to avoid endian dependency; while the machine-dependent portion of the code may be rewritten to support LE if the endian dependency can not be resolved easily.

You can choose between changing the data structure and changing the code referencing the data structure in making code endian-neutral or endian-aware. It is generally a good practice to *keep the data structure* and *change the code* referencing the data.

The attributes of the LE and BE storage models are summarized as follows:

- The BE addresses individual bytes in a multibyte data element from Most Significant Byte to Least Significant Byte (from left to right), similarly to how data elements are referenced (from left to right).
- For LE, data elements and individual data bytes within a data element are referenced in opposite directions.
- The starting address of a data element in both LE and BE storage models remains the same across the two data storage models.
- Individual bytes within a multibyte data element are addressed in a reversed order between a BE and a LE data storage model.
- For single-byte data types, endianness is of no consequence—characters (or other single-byte data) are at the same (starting) addresses in LE or BE mode.
- Endianness is of no consequence if a data element is referenced consistently using the same data type as defined.
- The endian dependency becomes a potential problem if internal bytes and/or a proper subset of a data element are referenced individually and/or multiple data elements are referenced as an aggregated, single data element.
- 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.

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 Itanium chip is an implementation of the Intel IA 64-bit architecture, and AIX 5L on 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 4 GB 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 4 GB.
- Support for larger file sizes.
- Greater scalabilty 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 (>4 GB) and, if so, is the user of the application likely to implement it on a system with more than 4 GB?
- Needs 64-bit integers.
- Needs larger files and data structures than can be supported on 32-bit systems.

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

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 AIX 5L on IA-64 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 section details the areas where problems may occur and explain how the C compiler in combination with lint can be used to isolate and correct problems.

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.

	AIX on PowerPC		AIX 5L on IA-64 on Itanium	
C and C++ Data Type	**32-bit Source Size/Alignment**	**64-bit Source Size/Alignment**	**ILP32 Size/Alignment**	**LP64 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
longlong	8/8	8/8	8/8	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	8/8	8/8
longdouble	16/16	16/16	16/16	16/16

Note that long long has a size of 8 bytes and is aligned on 8 byte boundaries in both 32-bit and 64-bit modes. Also note that integrals and floats are aligned according on their natural boundary (same as their size) which is different than the rules for IA-32. Applications requiring the IA-32 alignment rules can use the* **"#pragma align=ia64unix386". *However the size of long double will remain 16 bytes even when the pragma is used.*

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

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.

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 = i2 * 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) (i2 * l2);
15       i3 = (int) retlong((int)l3);
16     }
```

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
```

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    }
```

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
```

Pointer to an int Icompatible 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 dereference of such a pointer will clearly result in undefined behavior.

```
 1   long *ptrl1, *ptrl2;
 2   int *ptri1, *ptri2;
 3
 4   extern foo_int(int *);
 5   extern foo_long(long *);
 6
 7   void bar(void) {
 8
 9   ptrl1 = ptri1;
10   ptri2 = ptrl2;
11
12   ptrl1 = (long *)ptri1;
13   ptri2 = (int *)ptrl2;
14
15   foo_int(ptrl1);
16   foo_long(ptri1);
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 dereferenced, 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.

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) Assingnment 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)

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 that 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 * l1;
7   }
8
9   long func2 (arg)
10      long arg;
11   {
12      return arg * l2;
13   }
14
15   long func2_A (long arg) {
16      return arg * l2;
17   }
18
19   long func3 (long arg) {
20      return arg * l3;
21   }
```

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
```

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.

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.

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 may behave differently when compiled for LP64 mode than when compiled for ILP32 if it:

- 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

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 l1–1 because of the truncation to 32 bits. In LP64 mode, the addition result is a 64-bit long with a value of l1 + 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        l1 = l2 + 20000000 * 30000000;      /* 32 bit multiplication. */
8
9        l1 = l2 + 20000000L * 30000000;     /* 64 bit multiplication. */
10
11
12    /* case 2—constant is an unsigned int.  leading zeros are not
13    **          significant.
14    */
15        l1 &= ~(0x80000000);                /* turns off left most 33 bits. */
16        l1 &= ~(0x0000000080000000);        /* turns off left most 33 bits. */
17
18        l1 &= ~((long)0x80000000);          /* turns off bit 2**31 */
19        l1 &= ~(0x80000000L);               /* turns off bit 2**31 */
20
21    /* case 3—code depending on truncation at 32 bits on overflow.
22    */
23        l1 += 0xffffffff;                   /* l1 = l1—1 in ILP32 mode on
24                                            2's complement system */
25                                            /* l1 = l1 + 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        l1 = (l2 << 5)>> 16;               /* depends on truncation at bit 32 */
35
36        l1 = (l2 << (8 * sizeof(l2)- 27))>> (8 * sizeof(l2)—16);
37    }
```

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.However, there is compiler flag **-qlonglit** that will make default integral literal of type **long** instead of **int**.

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.

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 GB, 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   }
```

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 nonconforming 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
```

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   }
```

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 loss of bits.

```
(5) conversion of pointer loses bits
```

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   }
```

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 ()
```

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.

```
#if  #model(lp64)   or    #if  #model(ilp32)
```

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
            };
```

The above structure illustrates a C data structure that is binary data compatible in either ILP32 or LP64 mode. The declaration preserves the alignment and size of 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 an 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.

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
  - Do not assume that *long* or *unsigned long* data is 32 bits
  - Do 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	0xFFFFFFFFFFFFFFFFL	(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);
```

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	it64_t	64	8	64	8

Part 2. Migrating DYNIX/ptx Programs to AIX 5L on IA-64

This part includes information specific to migrating C, assembly language, and C++ applications from DYNIX/ptx to AIX 5L on IA-64.

Migrating C Applications

In addition to the information covered in other sections of this manual, the following factors should be considered when migrating your DYNIX/ptx C applications to AIX 5L on IA-64:

- C language incompatibilities
- C header file incompatibilities
- C compilation incompatibilities
- Linker incompatibilities
- Environment differences
- Shell Differences
- Tool Differences
- API incompatibilities
- ABI incompatibilities
- Licensing changes
- Installation changes

The remainder of this section discusses each of the preceding items. Note that additional features or options that are supported by AIX 5L on IA-64 are not described here. The list includes only incompatibilities that may prevent an application that previously ran under DYNIX/ptx from running or from running as expected on AIX 5L on IA-64. Refer to the IBM AIX 5L on IA-64 documentation for information on additional features or options.

C Language Incompatibilities

Any extensions to ISO C or implementation-defined differences that may cause incompatibilities are documented in the following subsections.

Preprocessor Directives

The following table lists the DYNIX/ptx preprocessor directives and pragmas that are not supported on AIX 5L on IA-64. Per the ANSI C standard, pragmas not supported by an implementation will be ignored. However, depending on the desired effect, you may want to replace an unsupported pragma with an equivalent facility on AIX 5L on IA-64, to rewrite your code to eliminate the use of an unsupported pragma entirely, or to address the original problem that caused use of the pragma in a different manner.

DYNIX/ptx Preprocessor Macro Name	Description	Equivalent Macro Under AIX 5Lon IA-64/ Recommendation
#assert *name (token)*	Associates *name* with *token*. *name* can then be used in conditional preprocessing directives.	Supported for C, C++
#unassert *name (token)*	Unassociates *name* with *token*. If *token* is not specified, all associations for *name* are removed.	Supported

DYNIX/ptx Preprocessor Macro Name	Description	Equivalent Macro Under AIX 5Lon IA-64/ Recommendation
#pragma int_to_unsigned *name*	Identifies *name* as a function whose type was int in a previous release of the C compiler, but whose type is unsigned in the current release.	None.
#pragma pack(*n*)	Defines *n* as the strictest alignment for any structure member. *n* must have a value of 1, 2, or 4.	Consider using "#pragma options align=packed" and "#pragma options align=reset".
#pragma sequent_*	Varies.	None.
#pragma weak *symbol*[=*definition*]	Declares *symbol* as weak. A weak symbol is similar to a global symbol but has lower precedence.	Will be supported

Predefined Macros

The following predefined macro, which is recognized by the DYNIX/ptx C compiler, is not recognized by the AIX 5L on IA-64 compiler:

`__IDENT__`

It is recommended that any applications that use the __IDENT__ predefined macro be updated to replace __IDENT__ with __LINE__.

Assembly Language Macros

Assembly language macros, a Sequent extension to ISO C, are not supported on AIX 5L on IA-64. Assembly language macros begin with the keyword `asm`. You must recode any application that uses assembly language macros to eliminate them. Note, however, that assembly language functions called from C programs are supported. See also "Migrating Assembly Language Programs" later in this section.

C Header File Incompatibilities

This information will be provided in a future version of this document.

C Compilation Differences

Specifying a Compilation Mode

Under DYNIX/ptx, you specified the compilation mode by passing a flag to the compiler or the default compilation mode, -Xt, was assumed. On AIX 5L on IA-64, the command used to invoke the C compiler determines the default compilation mode. The following table describes these commands.

AIX 5L C Compiler Invocations	Description
xlc	Invokes the compiler for C or C++ source files with a default language level of ansi, and specifies the compiler option -qansialias to allow type-based aliasing.
cc	Invokes the compiler for C or C++ source files with a default language level of extended and specifies the compiler options -qnoro and -qnoroconst (for placement of string literals or constant values in read/write storage).
c89	Invokes the compiler for C or C++ source files with a default language level of ansi, and specifies the compiler options -ansialias and -qnolonglong (disabling use of longlong), and sets -D_ANSI_C_SOURCE (for ANSI-conformant headers).

In addition, threaded applications must be compiled using the "_r" variation of the invocation. For example, "cc_r " sets the same compilation mode as "cc", but also allows creation of POSIX-threaded applications.

The following table shows the DYNIX/ptx compilation mode options and the closest corresponding invocation on AIX 5L on IA-64. Using the listed equivalent option does not guarantee syntactical or semantic compatibility, but it may reduce your porting effort. The default for all modes is to produce 32-bit binaries unless the -q64 option is specified or the OBJECT_MODE environment variable is set to 64.

DYNIX/ptx Compatibility Mode Option	Closest AIX 5L on IA-64 Equivalent
-Xs	None
-Xt (default compilation mode)	cc -qlanglevel=classic
-Xa	xlc
-Xc	c89

C Compiler Option Incompatibilities

The following table lists and describes the DYNIX/ptx C compiler options that have a different meaning on AIX 5L. Any makefile that uses these options must be updated appropriately.

Option	DYNIX/ptx Description	AIX 5L on IA-64 Description	Recommendation
-v	Perform more and stricter semantic checks and enable certain lint-like checks on the named C files.	Instruct the compiler to report information on the progress of the compilation.	Use lint on the C source file before compiling the program.
-#	List on standard error the compiler components and their arguments as called by cc.	List on standard error the full pathname of the compiler components and their arguments as called by cc but do not execute them.	Use -# and then reinvoke the compiler with the same command line minus "-#".

DYNIX/ptx C compiler options that are not supported on AIX 5L are listed in the following table. Any makefile that references these options must be updated either to use the corresponding option on AIX 5L or to eliminate the option.

Unsupported DYNIX/ptx C Compiler Options	Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-##	List on standard error the full pathname of the compiler and its components.	Use -# and then reinvoke the compiler with the same command line minus "-#".
-###	Provide the same functionality as -##, but no execution is performed.	Use -#.
-A*name(token)*	Associate *name* with the specified token as if by a #assert preprocessing directive.	Supported as on DYNIX/ptx
-A-	Ignore preassertions and predefined macros (other than those that begin with __).	Not Supported
-H	Cause the path name of each file included during the current compilation to be listed on standard error, one per line.	None.
-KPIC	Generate position-independent code.	None. The AIX 5L C compiler produces PIC by default.

Unsupported DYNIX/ptx C Compiler Options	Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-Kminabi	Direct the compilation system to use a version of the C library that minimizes dynamic linking without changing the application's ABI conformance.	None.
-Kthread	Create a threaded application. By default, applications are non-threaded.	Invoke the compiler with the "xlc_r" command or the appropriate _r-suffixed command.
-Q{y|n}	Enable/disable writing version information to the final output file. -Qy is the default behavior under DYNIX/ptx.	None.
-q{l|p}	Produce code that counts the number of times each source line is executed.	For -ql, none. For -qp, "cc -qp -dn" is equivalent to "cc -p" under AIX 5L on IA-64.
-V	Print the version number of the compiler on standard error.	None.
-Wc,-i	Treat all variables that could be declared volatile as if they were declared volatile. This option was intended for use on programs that should have volatile on some variable declarations but do not.	None. Any variables that should be typed as volatile must be declared to be of volatile type.
-Wc,-Og	Generate exactly the same code as would be generated for -g when no optimization level is specified.	None; the -qoptimize=0 option provides minimum optimization under AIX 5L on IA-64.
-Wc,-O0	Provide default code optimization. Optimizations that would increase code size or that require significant additional compilation time are not performed.	None; -qoptimize=0 is the default optimization option under AIX 5L on IA-64.
-Wc,-O1	Same as -Wc,-O0 plus invariant code motion, strength reduction, and code duplication. No function inlining is performed.	None; refer to the optimization option descriptions in the IBM cc(1) man page or the C documentation for information on the optimizations available with AIX 5L on IA-64.
-Wc,-O2	Same as -Wc,-O1 plus inlining of small user-defined functions found in the same .c file.	None; refer to the optimization option descriptions in the IBM cc(1) man page or the C documentation for information on the optimizations available with AIX 5L on IA-64.
-Wc,-O3	Same as -Wc,-O2 plus inlining of intrinsic library functions.	None; refer to the optimization option descriptions in the IBM cc(1) man page or the C documentation for information on the optimizations available with AIX 5L on IA-64.
-Wc,-SO	Send the assembly language output files to standard output.	None.
-Wc,-i386	Perform i386 specific optimizations and generate code suitable for i386 processors.	None; this option is obsolete.

Unsupported DYNIX/ptx C Compiler Options	Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-Wc,-i486	Perform i486 specific optimizations and generate code suitable for i486 processors.	None; this option is obsolete.
-Wc,-P5	Perform Pentium specific optimizations and generate code suitable for Pentium processors.	None; this option is obsolete.
-Wc,-P6	Perform optimizations specific to the Pentium Pro and generate code suitable for Pentium Pro processors.	None; this option is obsolete.
-Wc,-Pmmx	Perform optimizations specific to the Pentium II and generate code suitable for Pentium II processors.	None; this option is obsolete.
-Wc,-seq	Assert the -Xs option (Sequent compatibility mode) and recognize the non-ANSI keywords shared, private, and fortran. The macros __STDC__, __STDC_VERSION__, and __STRICT_ANSI__ are not predefined. This option was intended to support porting from pre-ANSI C to ANSI C.	None; this option is obsolete.
-Wc,-pw	Suppress "portability" warnings generated during compilation process.	None.
-Wc,-Y	Treat extern and static variables as shared rather than private.	None.
-Wc,+abi-socket or -Wc,+bsd-socket	Search the library paths for a file name cc_options located in a directory named abi-socket or bsd-socket. respectively. The library paths are those specified by -L, -YL, -YU, -YP or /lib:/usr/lib if nothing is specified.	None. The default under AIX 5L on IA-64 is BSD sockets.
-Wofl,-option	Use information in the program to perform ordering for locality.	None. None of the -Wofl options are supported in AIX 5L on IA-64.
-W0,xstring	Place all literal strings in read-only memory segment. Literal strings used in a const char * context are normally placed in a read-only segment while those in a char * context are normally placed in the read-write (.data) segment.	When the C compiler is invoked as "xlc", both const char * and char * literal strings are placed in read-only memory. You can use the -qnoro option to override this. When the C compiler is invoked as "cc", both are placed in read-write memory, unless you specify the -qro option on the command line.
-W0,-xstring_merge_ro	Create only one copy of identical read-only (const char *) literal strings.	This is the default on AIX 5L on IA-64 when the C compiler is invoked as "xlc". When the C compiler is invoked as "cc", multiple copies are created unless you specify the -qro option on the command line.

Unsupported DYNIX/ptx C Compiler Options	Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-W0,-xstring_merge_rw	Create only one copy of identical read/write (char *) literal strings.	This is the default on AIX 5L on IA-64 when the C compiler is invoked as "xlc". When the C compiler is invoked as "cc", multiple copies are created.
-W0,-xstring_merge	Shorthand for specifying both -W0,-xstring_merge_ro and -W0,-xstring_merge_rw.	This is the default in AIX 5L on IA-64 when the C compiler is invoked as "xlc". When the C compiler is invoked as "cc", multiple copies are created.
-W0,-noflatstk	Set up a standard stack frame for every function when optimizing code. By default, the -O option causes the stack frame of certain functions to be flattened.	None; users do not have control over this optimization.
-W1,-resvfptr	Reserve the frame pointer register so that it will not be used as a scratch register when optimizing code. By default, the -O option allows the frame pointer register to be used as a scratch register when a frame pointer is not needed.	None.
-Xs	Use the Sequent compatibility mode of compilation, which is closely compatible with the pre-ANSI C compiler. This disables recognition of the ANSI keywords const, volatile, and signed, and enables recognition of the non-ANSI keyword fortran. The compiler warns about language constructs that have differing behavior between ANSI C and pre-ANSI C and uses the pre-ANSI C interpretation. The macros __STDC__, __STD_VERSION__, and __STRICT_ANSI__ are not predefined.	None; this option is obsolete.
-Xt	Use the transition mode of compilation. The compiler warns about language constructs that have differing behavior between ANSI C and pre-ANSI C and uses the pre-ANSI C interpretation. The predefined macro __STDC__ has the value 1. The predefined macro __STD_VERSION__ has the value 199409L. The predefined macro __STRICT_ANSI__ is not defined. This was the default compilation mode under DYNIX/ptx V4.5.	No equivalent option, but -qlanglvl=classic may suffice for some programs or consider invoking the compiler as **cc**. For more information on the cc invocation, refer to "C Compilation", earlier in this section.

Unsupported DYNIX/ptx C Compiler Options	Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-Xa	Use the ANSI mode of compilation. The compiler warns about language constructs that have differing behavior between ANSI C and pre-ANSI C and uses the ANSI interpretation. The predefined macro __STDC__has the value 1. The predefined macro __STD_VERSION__ has the value 199409L. The predefined macro __STRICT_ANSI__ is not defined.	Use the -qlanglvl=ansi option or invoke the compiler as **xlc**. For more information on the xlc invocation, refer to "C Compilation", earlier in this section.
-Xc	Use the conformance mode of compilation. The compiled language and associated header files are assumed to be strictly ANSI C conforming. That is, no non-ANSI support is provided. The predefined macro __STDC__has the value 1. The predefined macro __STD_VERSION__ has the value 199409L. The predefined macro __STRICT_ANSI__ has the value 1.	Use the -qlanglvl=ansi option or invoke the compiler as **c89**. For more information on the c89 invocation, refer to "C Compilation", earlier in this section.
-Y*c*, *pathname*	Use the directory specified by *pathname* as the location of the tools for the component specified by *c*.	Use the -B*pathname* -t*c* option. *pathname* must end in '/' and *c* can be one or more of the following: p preprocessor a assembler l linkage editor c compiler front-end b compiler back-end

Linker Differences (ld)

The DYNIX/ptx linker options listed in the following table are not supported on AIX 5L on IA-64. These options must be replaced with the equivalent option on AIX 5L on IA-64 or removed from any makefiles that reference them.

Unsupported DYNIX/ptx Linker Options	DYNIX/ptx Description	Equivalent AIX 5L on IA-64 Option or Recommendation
-Bfull	Same as -Bexport, but does not support a symbol list.	Use -Bexport.
-R *path*	Specify library search directories to the runtime linker. *path* is a colon-separated list of directory pathnames. If present and not NULL, it is recorded in the output object file and passed to the runtime linker.	Use the LD_RUN_PATH or LD_LIBRARY_PATH environment variables to specify library search directories to the runtime linker.
-T*	Ordering for locality options.	None.
-O*	Ordering for locality options.	None.

Unsupported DYNIX/ptx Linker Options	DYNIX/ptx Description	Equivalent AIX 5L on IA-64 Option or Recommendation
-Uf[v]	Supply a dummy value (zero) for each undefined symbol and proceed to completion. This overrides the default behavior of listing undefined symbols and terminating with an error message.	None.
-Ur[v]	Make additional passes through the list of libraries to resolve symbols.	None.
-w	Turn off the warning about file offsets and memory segments that are not congruent. This option is intended for use with the -M option only.	None.
-YL,*dirlist*	Change the default directory used for finding libraries-the first default directory searched by ld (/lib under DYNIX/ptx) is replaced by *dirlist.*	None; use -YP,*dirlist* and specify all directories containing libraries to be searched in the order in which they must be searched.
-YU,*dirlist*	Change the default directory used for finding libraries--the second default directory searchcd by ld (/usr/lib under DYNIX/ptx) is replaced by the directory specified by *dirlist*.	None; use -YP,*dirlist* and specify all directories containing libraries to be searched in the order in which they must be searched.
-z nopage0	Do not bind anything to address zero. This option allows runtime detection of null pointers.	None.
-z defexec: *execfile*	Use dynamic symbols in the executable called *execfile* to resolve references that would otherwise be undefined in a new shared object. This option is ignored unless used with -G and -z defs.	None.
-z nooverlap	Issue a fatal error if segment attributes specified in a mapfile cause segment addresses to overlap. By default, this option is off with only a warning issued.	None.
-z objlist:*objfile*	Get the names of object or archive files from the file specified by *objfile*.	None.
-z searchself	When creating a shared object, mark the object so that the dynamic linker looks for symbols to bind within the object first, before any other shared objects are searched. This overrides the dynamic linker's normal breadth-first searching algorithm. This option is the default for shared libraries built with -Bsymbolic.	None.

Environment Differences

By default, temporary files for the compilation tools are stored in /tmp; many of the tools use /usr/tmp. You can set the TMPDIR environment variable to change the location of temporary files.

The default compilation mode is 32-bits unless the OBJECT_MODE environment variable is set or the -q64 compilation option is specified.

Shell Differences

Information on shell differences may be useful for migrating applications that include scripts.

Shell Invocation

The following table shows the differences in shell invocations between DYNIX/ptx and AIX 5L on IA-64.

Shell Invocation	DYNIX/ptx	AIX 5L on IA-64
Invoke default shell	/bin/sh. Bourne shell.	/usr/bin/sh. Korn shell.
Invoke restricted shell	/bin/rsh, /bin/sh -r for Bourne shell. /bin/rksh for Korn shell.	/usr/bin/Rsh, /usr/bin/bsh -r for Bourne shell.
Invoke remote shell	/usr/bin/resh. User's shell invoked.	/usr/bin/rsh, /usr/bin/remsh. User's shell invoked.
Invoke Bourne shell	/bin/sh	/usr/bin/bsh
Invoke Korn shell	/bin/ksh	/usr/bin/sh, /usr/bin/ksh
Invoke C Shell	/bin/csh	/usr/bin/csh

Bourne Shell Differences

Bourne shell differences between DYNIX/ptx and AIX 5L on IA-64 are listed in the following table.

DYNIX/ptx	AIX 5L on IA-64
${*parameter*:?*word*} -- if *parameter* is not set or is null, print *word* and exit from the shell	If *parameter* is not set or is null, print *parameter*: *word* and exit from the shell
[[=*c*=]] -- Matches a single character in the class to which *c* belongs.	Not supported
[[.*cc*.]] -- Matches any character which has the same relative order in the current collation sequence as *cc*.	Not supported
Default search path is /bin:/usr/bin	Default search path is /usr/bin:/etc:/usr/sbin:/usr/ucb:$HOME/bin:/usr/bin/X11:/sbin:.(current directory)
Supports newgrp special command	Not supported
SIGHUP trap handled immediately	No special handling for SIGHUP
Supports ulimit special command	Same except does not support -n option, which sets or displays the maximum file descriptor + 1, or the -v option, which sets or displays the maximum virtual memory size.

C Shell Differences

C shell differences between DYNIX/ptx and AIX 5L on IA-64 are listed in the following table.

DYNIX/ptx	AIX 5L on IA-64
Supports built-in command alloc, which displays the amount of dynamic memory acquired	Not supported
Supports -n option to echo command to suppress final newline	Not supported
Supports limit/unlimit built-in commands	Same except doesn't support stacksize as a resource
Argument list limit is 10240 bytes	Argument list limit is 4096 bytes

Korn Shell Differences

Korn shell differences between DYNIX/ptx and AIX 5L on IA-64 are listed in the following table.

DYNIX/ptx	AIX 5L on IA-64
Conditional expression "-a *file*" is same as "-e *file*"	Only true if *file* is a symbolic link that points to a file that exists.

Tool Differences

Unsupported DYNIX/ptx Tools

The following tools that were available under DYNIX/ptx are not available under AIX 5L on IA-64:

cof2elf, cprs, hidesyms, lprof.

as

A new assembler is provided with AIX 5L on IA-64. For more information on the assembler, refer to the IBM as(1) man page. See also "Migrating Assembly Language Programs."

debug

TBD

dis

A new disassembler is provided with AIX 5L on IA-64. For more information on the disassembler, refer to the IBM dis(1) man page.

make

AIX 5L does not support the following make(1) options, which are supported by DYNIX/ptx:

-b Compatibility mode to support old version of make
-Pn Supports parallel execution of commands in makefile

prof

Under DYNIX/ptx, the function name is the last field in the output displayed by prof(1). On AIX 5L on IA-64, it is the first field.

lint

Where possible, we recommend that you remove lint from your DYNIX/ptx applications before porting them to AIX 5L. You can use lint's -j option to enable complaints about explicit narrowing conversions from casts and the -t option to check portability to an LP64 implementation of C.

The following lint directives, supported under DYNIX/ptx, are not supported AIX 5L. Although it is not necessary to remove them from your code, you should be aware that they will have no effect:

```
/* CONSTCOND / or /* CONSTANTCONDITION */
/* CASTOK */
/* EMPTY */
/* FALLTHRU / or /* FALLTHROUGH */
/* LINTED message */
/* PRINTFLIKEn */
/* PROTOLIBn */
/* SCANFLIKEn */
/* SYMUSED */
/* UNIONOK */
```

The following table lists lint options that were supported under DYNIX/ptx but are not supported AIX 5L. You must remove these from any scripts that you port to the AIX 5L on IA-64 environment.

Unsupported DYNIX/ptx Lint Options	Description
-m	Suppress complaints about functions and external symbols that could be declared static.
-F	Print path names of files as they are processed.
-I*dirname*	Search for header files in the directory specified by *dirname* before searching the current directory and/or the standard place.
-j	Enable complaints about explicit narrowing conversions from casts.
-k	Alter the behavior of LINTED directives so that instead of suppressing warning messages for the code following LINTED directives, lint prints an additional message containing the comment inside the directive.
-K{thread | nothread}	Turn on the appropriate preprocessor flags for threaded or non-threaded applications.
-L*dirname*	Search for lint libraries in *dirname* before searching the standard place.
-s	Produce one-line diagnostics only.
-V	Print version number to standard error.
-W *filename*	Write a .ln file to the file specified by *filename*, for use by cflow(1).
-R *filename*	Write a .ln file to the file specified by *filename*, for use by cxref(1).
-W0,Y*dirname*	Search for header files in *dirname* instead of the standard place.
-Wc,*	Varies.
-Xs, -Xt, -Xa, -Xc	Specify a compilation mode.
-YI,*dirname*	Search for header files in *dirname* instead of the standard place.
-y	Specify that the file is to be treated as if it contained a LINTLIBRARY directive.

Other C Programming Tools Differences

The following table shows which options for the C programming tools are not supported on the AIX 5L on IA-64 environment. Refer to the next table for information on options which differ between DYNIX/ptx and AIX 5L on IA-64.

Tool	DYNIX/ptx Options Not Supported On AIX 5Lon IA-64	Option Description	Recommendation
ar	-V	Print version number to standard error.	None.
cb	-V	Print version number to standard error.	None.
cflow	-V	Print version number to standard error.	None.
cpp	-T	Use only the first eight characters to distinguish preprocessor options. Provided for backward compatibility.	None.
	-Y*dir*	Use the directory specified by *dir* in place of the standard directory (/usr/include) to search for include files.	None; can only specify directory to look in first (-I*dir*) rather than in place of the standard directory.
	-H	Print on standard error the pathnames of included files.	None.
	-V	Print version number to standard error.	None.
ctrace	-b	Use only basic functions in the trace code, that is, those in ctype, printf, and string.	None.

Tool	DYNIX/ptx Options Not Supported On AIX 5Lon IA-64	Option Description	Recommendation
	-V	Print version number to standard error.	None.
cxref	-d	Disable printing declarations.	None.
	-l	Disable printing local variables.	None
	-C	Run only the first pass of cxref.	None
	-F	Print the full pathnames of the referenced files.	None.
	-Lcols	Modify the number of columns in the LINE field.	None.
	-V	Print version number to standard error.	None.
	-W	Change the default width of a field.	None.
dump	-S*number* or -S*number1*,*number2*	Dump the core file segment with segment number *number* or the range of core file segments from *number1* to *number2*.	None.
lex	-V	Print version number to standard error.	None.
make	-P[*n*]	Permit *n* command sequences to be done in parallel.	None.
nm	-g	Print only external (GLOBAL) symbols.	Pipe the output through grep: "nm a.out | grep 'GLOB' "
	-e	Print only WEAK and GLOBAL symbols.	None.
	-P	Print information in a portable output format as specified by POSIX.	None.
	-I	Sort symbols by index before they are displayed.	None.
prof	-V	Print version number to standard error.	None.
strip	-r	Do not strip static or external symbol or relocation information.	None.
yacc	-Q{y|n}	Place version number information in y.tab.c. The -Qn option, the default, suppresses this.	None.
	-V	Print version number to standard error.	None.

The following table lists C programming tools that provide the same functionality on DYNIX/ptx and AIX 5L but through specifying different options.

Tool	Functionality	DYNIX/ptx Option	AIX 5L on IA-64 Option
nm	Prepend the name of the object file or archive to each output line.	-A	-r
	Print the value and size of a symbol in octal format	-t o	-o
	Print the value and size of a symbol in hexadecimal format	-t x	-x

API Incompatibilities

The following Sequent-specific library is not supported on AIX 5L. If your application references any function in this library, you must recode to replace the function with an AIX 5L supported function.

```
/usr/lib/libseq.so (-lseq)
```

The following table lists the functions that comprise libseq.so. If your application uses any of these libseq functions, you must either eliminate the call or replace the function with an equivalent function AIX 5L.

Function Names	Description	Recommendation and/or Equivalent Function on AIX 5L on IA-64
acladd, acldel, aclchng, aclread, aclwrite, aclerror, aclbtoa, aclatob, aclvalid	Access control list (acl) operations	TBD
a_getacl, a_stat, a_lstat, a_setacl	Get file access control list (acl) information	TBD
atoll, strtoll, strtoull	Convert string to long long integer	TBD
attach_proc	Attach a process to a quad with the specified resource	TBD
au_ctl, au_entry, au_getauthid, au_setauthid, au_getpmask, au_set-mask	Provide audit control	TBD
bdflush	Flush modified disk buffers to disk	TBD
boot	Boot the uptime kernel from the standalone kernel on SCI-based systems9	TBD
cd_getdevmap, cd_setdevmap, cd_suf	Operate CD-ROM device	TBD
cfg_ctl, cfg_sys	Provide device configuration control	TBD
cfg_findobj, cfg_idxobj, cfg_info_init, cfg_info_free, cfg_gethdr, cfg_getobj, cfg_nextobj, cfg_objidx	Read and search device configuration information	TBD
cfg_report_count, cfg_report_get, cfg_report_overflow	Extract reports from cfg_sys error buffer	TBD
creat64, fopen64, fsetpos64, fgetpos64, fseeko64, ftello64, ftw64, lockf64, llseek, lseek64, mmap64, nftw64, open64, statvfs64, getrlimit64, setrlimit64, truncate64, ftruncate64	Manipulate large files	TBD
DIO_Read, DIO_Write, DIO_Read64, DIO_Write64, DIO_Readv64, DIO_Writev64, DIO_Ainit, DIO_Await, DIO_Apoll*	Provide synchronous/asynchronous direct I/O on UNIX files and character special files	TBD
detach_proc	Detach a process from a previously assigned quad	TBD
devt_to_name	Find pathname of a device with the specified dev_t	TBD
engdata_init, getengno, getquadno	Get engine or quad information	TBD
engemptyset, engfillset, engaddset, engdelset, engismember, engisempty	Manipulate sets of engines	TBD
fstat64, lstat64, stat64	Get large file status	TBD
getbootflags, setbootflags	Get/store system boot flags	TBD
get_disk_stats, get_ndisks,	Get disk information	TBD
get_ntapes, get_tape_stats	Get tape information	TBD
getgeombyname	Get disk geometry by name	TBD
getpagesize	Get system page size	TBD
getphysdev, setphysdevent, fgetphysdev,	Get/control logical to physical	TBD

Function Names	Description	Recommendation and/or Equivalent Function on AIX 5L on IA-64
dgetphysdev, endphysdevent	device mappings	
get_process_stats	Get resource utilization information	TBD
getswapstat	Obtain summary swap space usage statistics	TBD
getdtablemax	Get maximum in-use descriptor table index	TBD
getdtablesize/setdtablesize	Get/set descriptor table size	TBD
getkerndata	Get the contents of a kernel data structure	TBD
getrgnname	Get the region name of the caller or the specified process	TBD
getscsiinfo, getscsimatch	Get scsiinfo structure	TBD
gettimeofday_mapped	Get current time of day	TBD
getusclk, usclk_init	Get/initialize microsecond clock	TBD
addmntent, endmntent, getmntent, hasmntopt, setmntent	Get/control filesystem descriptor file entry	TBD
lwp_trace	Observe/control LWPs	TBD
mmap64, mmapq, mmap64q	Map an open file into the processes's address space	TBD
mptrace	Provide multiprocess trace facility	TBD
nblocks	Calculate number of blocks and indirect blocks	TBD
ndb_ctl	Provide device naming database control	TBD
offline_all	Take all but one active processor off line	TBD
priv_ctl	Vectored superuser privilege control	TBD
proc_ctl	Manipulate process attributes	TBD
qexecl, qexecv, qexecle, qexecve, qexeclp, qexecvp	Execute a file with specified quad placement	TBD
qfork, shfork, shqfork	Create a new process	TBD
quademptyset, quadfillset, quadaddset, quaddelset, quadismember, quadisemptyset, quadandset, quadorset, quaddiffset	Manipulate sets of quads	TBD
quad_loc	Locate the quad set containing a specified resource	TBD
quotactl	Manipulate disk quotas as 512-byte blocks	TBD
rqcreate, rqdelete, rqgetattr, rqgeteattr, rqsetattr, rqsethome, rqshutdown, rqstat	Create a new system run queue/control run queue created via rqcreate	TBD
read_constab	Read constab entries into a structure	TBD
rgn_*	Create/control regions	TBD
shmatvw	Attach a virtual window to a shared memory segment	TBD
shmgetq	Get shared memory segment identifier associated with a quad	TBD

Function Names	Description	Recommendation and/or Equivalent Function on AIX 5L on IA-64
shmgetv	Restrict paging policy to a set of quads	TBD
sigcontext	Get signal context	TBD
sigstack	Set/get signal stack context	TBD
sync_op	Create/control process synchronization primitives	TBD
sys_boot	Boot the uptime kernel from the standalone kernel on SCI-based systems	TBD
sysinfo	Get/set system information strings	TBD
tmp_affinity	Bind a process to a processor	TBD
tmp_ctl	Allow processes to query status of the processor and quad pool resources	TBD
virtwin	Create new virtually-windowed address translations to a mapped object	TBD
vm_ctl	Examine/change virtual memory tuning parameters	TBD
vm_getinfo	Examine system memory information	TBD

ABI Incompatibilities
This information will be provided in a later version of this document.

Licensing Changes
This information will be provided in a later version of this document.

Installation Changes
This information will be provided in a later version of this document.

Migrating Assembly Language Programs
The AIX 5L compiler produces highly optimized code. It would be difficult to write assembly language programs that produce better code than the compiler. If you rewrite your assembly language programs in C, you will only need to recompile to take advantage of future compiler technology.

In contrast to the DYNIX/ptx compilation tools, in-line assembly code (e.g., asm ("XXX")) is not supported.

Migrating C++ Programs
The preceding information on migrating C applications also applies to migrating applications written in C++. The following additional factors should also be considered for C++ applications: C++ compiler differences and C++ class libraries. The remainder of this section describes each of these factors.

C++ Compiler Differences

Invoking the C++ Compiler

The DYNIX/ptx C++ compiler is invoked by the name "c++". The AIX 5L Visual Age 5.0 C++ compiler is invoked by using the "xlC" command. See "Specifying a Compilation Mode" in the "C Compiler Differences" section for optional invocations.

C++ source files can have ANY suffix except .o, .a, .c, .so if the -+ option is specified.

C++ Compiler Command Line Differences

The following table lists the DYNIX/ptx C++ command line options that are not supported by the AIX 5L C++ compiler. **Any makefile file which references these options must be updated appropriately**.

<table>
<tr><th>Unsupported DYNIX/ptx C++ Compiler Option</th><th>DYNIX/ptx Description</th><th>Equivalent AIX 5L on IA-64 Option/Recommendation</th></tr>
<tr><td>-@ file</td><td>Place the contents of file at the current position on the command line.</td><td>None.</td></tr>
<tr><td>-Aname(token)</td><td>Associate name with the specified token as if by a #assert preprocessing directive.</td><td>–A is supported</td></tr>
<tr><td>-A-</td><td>Ignore predefined assertions and predefined preprocessor macros (except those beginning with __).</td><td>Supported</td></tr>
<tr><td>-abilngdbl</td><td>Force values typed as long double to be 12 bytes instead of 8.</td><td>–qlongdouble=80</td></tr>
<tr><td>-[no]alttok</td><td>Allow use of digraph characters.</td><td>Use -q[no]digraph.</td></tr>
<tr><td>-ansi</td><td>Specify source conforms to ANSI standard.</td><td>Use -langlvl=ansi.</td></tr>
<tr><td>-ansilibs</td><td>Use ANSI standard libraries.</td><td>None.</td></tr>
<tr><td>-B{dynamic|static}</td><td>Specify whether static or dynamic libraries should be linked.</td><td>Use -b{dynamic|static}.</td></tr>
<tr><td>-[no]bool</td><td>Enable the bool keyword.</td><td>–q[no]keyword=bool</td></tr>
<tr><td>-cfront{2|3}</td><td>Specify that source written for cfront 2.1 or cfront 3.0 is to be compiled.</td><td>TBD</td></tr>
<tr><td>-D-</td><td>Ignore predefined preprocessor.</td><td>None.</td></tr>
<tr><td>-d limit</td><td>Set the driver error limit to the value specified by limit.</td><td>None.</td></tr>
<tr><td>-d{y|n}</td><td>Specify static or dynamic linking.</td><td>Use -b{dynamic|static}.</td></tr>
<tr><td>-[no]dollar</td><td>Allow '$' in identifiers.</td><td>Use -q[no]dollar.</td></tr>
<tr><td>-dry</td><td>Display main driver actions but do not perform them.</td><td>Use -#.</td></tr>
<tr><td>-dryrun</td><td>Display driver actions but do not perform them.</td><td>Use -#.</td></tr>
<tr><td>-e limit</td><td>Set the compiler error limit to the value specified by limi.t.</td><td>Use -qmaxerr=limit:e.</td></tr>
<tr><td>-[no]eh</td><td>Enable exception handling.</td><td>Use -q[no]eh.</td></tr>
<tr><td>-[no]explicit</td><td>Enable support for the explicit specifier on constructor declarations.</td><td>–qnokeyword=explicit</td></tr>
<tr><td>-Fmaxopt</td><td>Enable maximum high-level optimization</td><td>Use -O3.</td></tr>
<tr><td>-force</td><td>Force the definition of virtual function tables where the heuristic used by the compiler provides no guidance.</td><td>Use -qvftable.</td></tr>
</table>

Unsupported DYNIX/ptx C++ Compiler Option	DYNIX/ptx Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-H	Only preprocess source files and write a list of all #include files to standard output.	None, but the -qmakedep option provides similar output.
-help	Display a quick-reference compiler option summary before terminating.	None.
-I*dirname*[:*dirname*…]	Modify include file search path.	-I*dirname* is supported, change -I*dir1*:*dir2* to -I*dir1* -I*dir2*.
-include *filename*	Include file specified by *filename* first (before source files).	None.
-instant{all|used}	Control template instantiation.	See -qtempinc
-K{none|PIC|pic}	Control generation of position-independent code.	None; on AIX 5L on IA-64, all code is position independent.
-keep	Keep any temporary files created.	None.
-L*dirname*[:*dirname*]	Add the directory specified by *dirname* to the list of directories ld searches for libraries.	-L*dirname* is supported, change -L*dir1*:*dir2* to -L*dir1* -L*dir2*.
-[no]namespace	Enable support for namespaces.	–q[no]keyword=namespace
-newforinit	Cause any declarations within a for statement to have the scope of the block contained within the for statement. This is the default unless one of the following options is specified (in which case -trdforinit is the default): -version3, -cfront2, -cfront3, -preansilibs.	this is the default on AIX 5L on IA-64. To get the old behavior use: -qlanglvl=noansifor
-[no]newvec	Allow overloading of operators new[ ] and delete[ ].	None.
-nocrts	Do not load default startup modules.	None.
-nodefs	Prevent the driver from passing any default assertions, macros, include directories, libraries or startup modules to the compiler or linker.	None.
-noinline	Prevent any inline function calls.	Use -qnoinline.
-nolibs	Prevent the driver from passing default libraries to the linker.	None.
-O, -O1, -O2, -O3	Enable various optimization levels; -O implies all.	Refer to the AIX 5L on IA-64 documentation for information on optimization levels.
-preansilibs	Use cfront-compatible versions of the streams and complex libraries.	None.
-Q*tool path*	Use *path* as the pathname to the compiler tool specified by *tool.*	Use -B*path* -t*tool. path* must end in '/'.
-Qinclude *dir*[:*dir*…]	Use the specified directories instead of /usr/include.	No exact equivalents but you can use the -qnostdinc option to prevent the standard locations from being searched and the -I*dir* option to specify which directories to search.
-Qinstall *path*	Specify the root directory of the C++ compiler install tree.	None.
-Qlib[:*dir*…]	Specify the directories to be used instead of /lib and /usr/lib.	None.

Unsupported DYNIX/ptx C++ Compiler Option	DYNIX/ptx Description	Equivalent AIX 5L on IA-64 Option/Recommendation
-Qoption *tool args*	Pass the arguments specified by *args* to the specified compiler tool.	None.
-Qpath *pathname*	Specify the directory to search first for compilation tools and startup object files.	No equivalent option but you can use -B*path* to specify the path for the compiler or the compiler component to use. To change the path searched for startup object files, you can modify the configuration file pointed to by /etc/ibmcxx.cfg.
-Q{y|n}	Control the writing of tool version information to the final output file. -Qn is the default.	None.
-remarks	Enable remark-level error messages.	Use -qinfo=all.
-[no]restrict	Control recognition of the restrict keyword; -norestrict is the default.	None; restrict is not a valid keyword on AIX 5L on IA-64.
-[no]rtti	Control generation of runtime type information.	Use -q[no]rtti.
-seqlngdbl	Force long double to be 8 bytes long.	Default on AIX 5L on IA-64, or use -qnoldbl128.
-shared	Use shared versions of the C++ libraries instead of static archive libraries.	None.
-[no]std	Control use of the standard namespace.	None.
-nostdincs	Prevent the driver from passing any default include directories to the compiler.	None.
-nostdlibs	Prevent the driver from passing any default library directories to the linker.	None.
-[not]strict	Prevent the compiler from generating warnings or messages about non-standard constructs.	None; however, on AIX 5L on IA-64, -qinfo=all may provide some of the same information as -strict.
-strictwarn	Cause the compiler to generate warnings for any constructs that do not conform to the C++ standard.	None.
-suppress	Suppress the definition of virtual function tables where the compiler's heuristic provides no guidance.	None.
-sysdefs	Define preprocessor macros that describe the current user and machine.	None.
-t	Produce a C source from C++ source.	None, option conflicts with AIX 5L on IA-64's -t option.
-temp *dir*	Place temporary files in *dir*.	None; use the TMPDIR environment variable.
-terse	Display compiler messages in a shorter form.	None.
-time	Display timing information for compiler processes.	Use -qphsinfo.
-tpautooff	Disable automatic instantiation of templates.	None.
-trdforinit	Cause any declarations within a for statement to have the scope of the block containing the for statement. This is the	None; on AIX 5L on IA-64, declarations within a for statement have the same scope as

Unsupported DYNIX/ptx C++ Compiler Option	DYNIX/ptx Description	Equivalent AIX 5L on IA-64 Option/Recommendation
	default when one of the following options is specified: -version3, -cfront2, -cfront3, or -preansilibs.	the for statement.
-[no]typename	Control the recognition of the `typename` keyword. -typename is the default.	None; on AIX 5L on IA-64, `typename` is a valid keyword.
-V	Print the compiler's version number.	TBD
-version3	Indicate that source files written for previous versions of C++ are to be compiled.	No equivalent option but specifying -langlvl=compat may reduce your porting effort.
-[no]wchar	Control recognition of the `wchar_t` keyword. -wchar is the default.	Use –q[no]keyword=wchar_t
-writefiles	Pass the input files to the main driver via a temporary file instead of the command line.	None.
-X{a|c|s|t}	Specify that the source file to be compiled contains C language source and select the C dialect.	Use -qlanglvl=*language*. where *language* is one of the following: ansi, extended, or compat. Refer to the IBM AIX 5L on IA-64 documentation for more information on language levels.
-X{S|U}	Control whether the variables of type `char` are signed or unsigned. -XS is the default.	Use -qchars={signed|unsigned}. On AIX 5L on IA-64, ` -qchars=signed is the default.
-xar	Instruct the main driver to build an archive library instead of invoking the linking to create an executable.	None.
-y	Perform syntax checking only.	Use -qsyntaxonly. On AIX 5L on IA-64, the -y option enables compile-time rounding of floating point constants.
-Zm	Indicate that the program contains a misaligned structure member access.	None.
-Zp *align*	Specify *align* as the maximum alignment for a non-bitfield structure member.	None.

C++ Class Libraries

The DYNIX/ptx C++ compiler product includes the Rogue Wave Tools.h++ class library. TheAIX 5L C++ product does not. Applications that depend on the Tools.h++ library will need to purchase them directly from Rogue Wave.

Part 3. 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