AIX Version 4.1 Porting Guide

August 12, 1994

RISC System/6000 Division
Technology Transfer Department
Building 821, Internal Zip 2830
11400 Burnet Road
Austin, Texas 78758-3493

First Edition (August, 1994)

This edition applies to Version 4, Release 1, Modification Level 0 of the AIX Operating System.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

Comments may be addressed to:

IBM Corporation
Dept., J65S Building 821, Internal Zip 2830
11400 Burnet Road
Austin, Texas 78758-3493

Contents

Figures

Tables

Special Notices

This document describes changes made between the AIX Version 3.2 release and the AIX Version 4.1 release.

The information in this publication is not intended as the specification of any programming interfaces.

See the PUBLICATIONS section of the IBM Programming Announcement for the AIX Version 4.1 Operating System for more information about what publications are considered to be product documentation.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	AIX/6000
AIXwindows	AS/400
C Set ++	CAEDS
FFST	First Failure Support Technology
graPHIGS	IBM
InfoCrafter	InfoExplorer
Micro Channel	NetView
POWER Architecture	PowerOpen
PowerPC	PowerPC Architecture
PowerPC 601	PowerPC 603
POWERserver	POWERstation
RISC System/6000	RS/6000
SOMobjects	Turboways

The following terms are trademarks of other companies:

SunOS, SPARCstation, Network File System, NFS	Sun Microsystems, Inc.
Display PostScript	Adobe Systems, Inc.
Ethernet	XEROX Corporation.
iFOR/LS	Gradient Technologies, Inc.
NFS/NIS	Sun Microsystems.
OpenGL	Silicon Graphics, Inc.
OSF/Motif	Open Software Foundation.
PEX, Xwindow System	MIT
Spaceball	Spaceball Technologies, Inc.
UNIX	X/Open Company, Limited
Windows NT	Microsoft Corporation

Preface

This document describes changes made between the AIX Version 3.2 release and the AIX Version 4.1 release.

This document is intended for customers who want to migrate applications or kernel extensions to AIX Version 4.1. The audience is assumed to already be familiar with AIX Version 3.2.

How This Document Is Organized

The document is organized as follows:

- Chapter 1, "Introduction"
- Chapter 2, "AIX Version 4.1 Packaging Changes"

  This provides a summary of the packaging changes.
- Chapter 3, "iFOR/LS License Management"

  This provides an overview of iFOR/LS.
- Chapter 4, "Installation/Migration"

  This provides a summary of the installation/migration changes.
- Chapter 5, "Architecture Overview"

  This provides an overview of the chips introduced in AIX Version 3.2.5. The instruction set changes are included.
- Chapter 6, "Compiler, Loader, and Linker/Binder Changes"

  This provides a summary of the compiler changes, including the new compilation modes introduced for the new computer chips.
- Chapter 7, "Programming Changes"

  This provides changes that affect programmers.
- Chapter 8, "User Commands"

  This provides a list of changes to user commands.
- Chapter 9, "Multi-Processing"

  This provides an overview of multi-processing from a theoretical point of view.
- Chapter 10, "Multi-Processor Synchronization"

  This discusses changes to locking required due to threads and multi-processing.

AIX Version 4.1 Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- AIX Version 4.1 Licensed Program Specifications, GC23-2637
- AIX Version 4.1 Getting Started, SC23-2527
- AIX Version 4.1 Installation Guide, SC23-2550

- AIX Version 4.1 Startup Instructions for Preinstalled Systems, SC23-2668
- AIX Version 4.1 Quick Installation Guide, SC23-2650
- LPS for AIXwindows Environment, SC23-2638
- AIX Version 4.1 Network Installation Management Guide and Reference, SC23-2627
- AIX/and Related Products Documentation Overview, SC23-2456
- AIX Version 4.1 System Management Guide: Operating System and Devices, SC23-2525
- AIX Version 4.1 System Management Guide: Communications and Networks, SC23-2526
- AIX Version 4.1 System User's Guide: Operating System and Devices, SC23-2544
- AIX Version 4.1 System User's Guide: Communications and Networks, SC23-2545
- AIX Version 4.1 Problem Solving Guide and Reference, SC23-2606
- AIX Version 4.1 Messages Guide and Reference, SC23-2641
- AIX Versions 3.2 and 4.1 Performance Tuning Guide, SC23-2365
- AIX Version 4.1 Files Reference, SC23-2512
- AIX Version 4.1 Topic Index and Glossary, SC23-2513
- AIX Version 4.1 INed Editor Users Guide, SC23-2524
- AIX Version 4.1 Commands Reference, SB0F-1851; which may also be order by individual volumes:
  - AIX Version 4.1 Commands Reference, Volume 1, SC23-2537
  - AIX Version 4.1 Commands Reference, Volume 2, SC23-2538
  - AIX Version 4.1 Commands Reference, Volume 3, SC23-2539
  - AIX Version 4.1 Commands Reference, Volume 4, SC23-2540
  - AIX Version 4.1 Commands Reference, Volume 5, SC23-2639
  - AIX Version 4.1 Commands Reference, Volume 6, SC23-2640
- IBM RISC System/6000 System Overview and Planning, GC23-2406
- AIX Version 4.1 Assembler Language Reference, SC23-2642
- AIX Version 4.1 Writing a Device Driver, SC23-2593
- AIX Version 4.1 General Programming Concepts Volume 1: Writing Programs, SC23-2533
- AIX Version 4.1 General Programming Concepts Volume 2: Debugging Programs, SC23-2490
- AIX Version 4.1 Communications Programming Concepts, SC23-2610
- AIX Version 4.1 Kernel Extensions and Device Support Programming Concepts, SC23-2611
- AIX Version 4.1 Graphics, SC23-2612
- AIX Version 4.1 Technical Reference, SB0F-1852; which may also be ordered by separate volume:
  - AIX Version 4.1 Technical Reference, Volume 1: Base Operating System and Extensions, SC23-2614
  - AIX Version 4.1 Technical Reference, Volume 2: Base Operating System and Extensions, SC23-2615
  - AIX Version 4.1 Technical Reference, Volume 3: Communications, SC23-2616
  - AIX Version 4.1 Technical Reference, Volume 4: Communications, SC23-2617
  - AIX Version 4.1 Technical Reference, Volume 5: Kernel and Subsystems, SC23-2618
  - AIX Version 4.1 Technical Reference, Volume 6: Kernel and Subsystems, SC23-2619
  - AIX Version 4.1 Technical Reference, Volume 7: AIXwindows, SC23-2620

    - AIX Version 4.1 Technical Reference, Volume 8: Enhanced Xwindows, SC23-2621
    - AIX Version 4.1 Technical Reference, Volume 9: Enhanced Xwindows, SC23-2622
    - AIX Version 4.1 Technical Reference, Volume 10: Enhanced Xwindows, SC23-2623
    - AIX Version 4.1 Technical Reference, Volume 11: Master Index, SC23-2624
- AIX Version 4.1 Keyboard Technical Reference, SC23-2631
- AIX Version 4.1 Quick Reference, SC23-2529
- AIX Version 4.1 iFOR/LS Tips and Techniques, SC23-2666
- AIX Version 4.1 SOMobjects Users Guide, SC23-2680
- AIX Version 4.1 SOMobjects Programming Reference Manual, SC23-2681
- AIX Version 4.1 SOMobjects Quick Reference Guide, SC23-2682
- AIX Version 4.1 Desktop Script User′s Guide, SC23-2678
- AIXwindows Desktop Advanced User′s and System Administrator′s Guide, SC23-2671
- AIX Version 4.1 AIXwindows Internationalization Programming Guide, SC23-2511
- AIX Version 4.1 AIXwindows Programming Guide, SC23-2632
- AIX Version 4.1 Enhanced Xwindows Programming Guide, SC23-2636
- AIX Version 4.1 iFOR/LS System Management Guide, SC23-2665
- AIX Version 4.1 Xstation Management Guide, SC23-2713

Chapter 1. Introduction

1.1 Purpose of This Document

The intent of this document is to discuss the binary compatibility issues involved in going from AIX Version 3.2 to AIX Version 4.1. Only changes affecting binary compatibility and migration will be discussed here.

This document is a subset of the information contained in **All About AIX Version 4.1** .

Chapter 2. AIX Version 4.1 Packaging Changes

2.1 Packaging Definitions

Licensed Program Products (LPPs): A complete software product including all packages and filesets associated with that Licensed Program Product. For example, the Base Operating System (BOS) is a Licensed Program Product.

Package: A group of filesets with common function.

Fileset: Filesets provide a specific function. For example, **bos.net.tcp.nfs** is a fileset in the **bos.net** package. A fileset is the smallest individually installable unit. A fileset is a collection of functionally related files to form an installable unit.

The commands reference in AIX Version 4.1 includes which fileset a command belongs to in a Licensed Program Product.

Table 1. Example of Packaging Term Usage

Licensed Program Product	Package	Fileset
adacmp	adacmp	adacmp
bos	bos.INed	bos.INed
bos	bos.diag	bos.diag.rte

2.2 AIX Version 3 Licensed Program Product Prerequisite Checking

When an AIX Version 3.2 package has been broken into several AIX Version 4.1 filesets, the AIX Version 4.1 package which is the core fileset will make two entries for itself in the ODM databases: one with the old name so that AIX Version 3.2 Licensed Program Products will discover that they are installed and one with their new name. An entry will be added into the very base level for a set of things migrated from AIX Version 3.2. For example, **bos.net.tcp.client** will add the **bosnet.tcpip.obj** entry, but **bos.net.tcp.server** will not, since it prereqs client.

While this will prevent the old package from installing over part of the new filesets, it may have a side effect of potentially satisfying prereqs that really shouldn't be met:

- Files a and b are an AIX Version 3.2 fileset (aix32_lpp_name)
- z needs b, so it prereqs aix32_lpp_name,
- a is installed from an AIX Version 4.1 package, which does not include b but is the core fileset and installs a compatibility entry for aix32_lpp_name,
- z's prereq is satisfied, but it shouldn't have been.

Chapter 3. iFOR/LS License Management

3.1 iFOR/LS Application Developer′s Toolkit (ADK)

ADK is used by application developers to enable their code to work with license servers. ADK is available as an Licensed Program Product called ″Network License System″.

The most important components of the ADK are the libraries that bind license management calls, such as those to request and release licenses. iFOR/LS library calls are provided in **/usr/lib/netls/libnetls.a** and **libnck.a**.

The iFOR/LS API (application programming interface) has over 50 functions, including advanced features for managing queues of users waiting for licenses and querying the number of available and in-use licenses. Enabling an application is relatively straightforward and requires as few as three additional calls. Sample code will be available in **/usr/lib/netls/examples/client**.

Administration Tools: The following tool is provided in the **/usr/lib/netls/bin** directory:

- **ls_pass** - creates licenses for customers of licensed software applications.

Files: The **ls_pass** command stores information about customers in a file in the **/usr/lib/netls/conf** directory.

Chapter 4. Installation/Migration

4.1 lslpp Command Changes

The output of **lslpp -l** and **-L** has changed to reflect the new update strategy. **lslpp -m** is no longer valid. **lslpp -A** is no longer valid. Use the new **instfix** command to get APAR information.

In AIX Version 3.2, the **-l** flag gave information about products only. Now, **lslpp -l** will show all information necessary to describe the running level of code for a specified product. An **installp -l** can be done to find out what images are on any media.

In AIX Version 3.2, **lslpp -al** gave information about products, fileset updates and all the products that were labeled AVAILABLE. It wasn′t just the products on the install device that had AVAILABLE records in the Vital Product Database (VPD), but any install image the machine had ever seen, as well as any image that had been on the machine and subsequently rejected or removed. It was decided to not keep available records to reduce the amount of space required for the VPD and to speed up searches. The goal is to not have any AVAILABLE records kept in AIX Version 4.1, although there are a few rare instances where they can occur. **lslpp -al** will show information about all fileset updates that have been labeled COMMITTED for a product and will not show products labeled AVAILABLE.

Chapter 5. Architecture Overview

The purpose of this chapter is to explain the new hardware and its impact on application compatibility and performance, as well as to provide migration guidelines, plans and recommendations.

5.1 New Hardware

Prior to October 1993, all RISC System/6000 products were based on the POWER architecture. There are two POWER architecture implementations:

1. POWER (RS1), which is a 7 to 9-chip implementation, and
2. the RISC Single Chip (RSC), a 1-chip implementation.

Both implementations support the same instruction set and provide binary compatibility to all applications.

On September 21, 1993, IBM announced two new evolutionary architectures:

1. POWER2 (also known as POWERX, RIOS2, RS2) - an extension of the POWER architecture
2. PowerPC-601 - The first chip in an architecture family, developed jointly with Motorola and Apple

Additional PowerPC implementations will offer a wide variety of price/performance points that will be used in low-end products, such as laptop and notebook computers, to high-end servers and numeric intensive computing.

5.1.1 PowerPC Architecture

PowerPC architecture uses a single chip rather than multiple chips as in the POWER and POWER2 architectures. This will allow faster clock speeds and lower cost. Although it is a single chip, the PowerPC will still have multiple execution units. The number of execution units will vary with each implementation, affecting price and performance. Graphics capabilities are included in some of the chips to improve graphics performance (there is a compiler option to select the generation of the graphics instructions). The PowerPC architecture also provides support for external caches and multi-processor implementations.

The architecture includes optional features for application uses, operating system uses, and other purposes. The application-level optional features permit performance optimizations, and are intended to be used in dynamically linked implementation-specific libraries. The operating-system-level optional features include tracing facilities, and specialized support for controlling I/O devices.

The following changes are handled by the hardware and/or operating system and do not affect a user's uni-processor application. Only those developers writing assembly language code need to be concerned with the migration of their operating system.

1. PowerPC supports single-precision floating point, which is very important in the PC world. On the POWER architecture, all operations are done in

double-precision. This means that single-precision numbers are first converted to double-precision, and then converted back after the operation. It takes time to do the conversions, and a double-precision operation takes longer to perform on certain implementations and requires more storage.

2. PowerPC memory ordering is relaxed for multi-processing. Strong ordering means you have to do everything in the hardware, which slows you down. Relaxed ordering means the software can decide when you need that coherency.

   The architecture specifies a memory model with relaxed ordering requirements. For most multi-threaded applications executing on a uni-processor system and for single-threaded applications, this property of the architecture is transparent (that is, cannot be detected by the program). However, it does affect multi-threaded applications that execute on multi-processor systems, and programs that share memory with I/O devices on all systems. The architecture includes several synchronization instructions which can be used to ensure the desired ordering of memory accesses. A program must include these instructions at appropriate places to ensure correct results when the ordering of memory accesses is important.

3. Out-of-order execution is supported, which means that instructions are not necessarily executed in sequential order. An instruction in the instruction cache may be sent for execution to an idle execution unit, even though that instruction is not the next sequential instruction. For example, if an integer instruction is executing and the floating point unit is not busy, the floating point unit may be sent a floating point instruction to execute out-of-order at the same time that the integer unit is executing the integer instruction. Instructions that depend on the result of previous instructions (such as a previous load instruction) will not execute out-of-order.

4. The chip also supports speculative execution. When branches occur (if-then, case statements, and so on), it assumes that the branch is not taken unless the branch instruction has a negative displacement (meaning branching backward, such as back to the start of a do loop) and will continue to execute instructions. If the assumption is correct, then time has been gained by not waiting for the result of the compare. If the branch is taken, then the execution of instructions that occurred after the branch are simply thrown away.

5. The architecture also supports static branch prediction. Any branch that has a negative displacement (backward branch) is assumed to be taken. The hardware will begin to fetch instructions at the backward address. Branches with positive displacement are assumed not to be taken. Writers of assembly code can set a bit to guide the hardware whether the branch is likely to be taken. Compilers don′t set this bit, because they don′t have the information on which to base the decision.

6. New primitive instructions that can be used to easily construct semaphores, such as compare-and-swap and test-and-set are introduced. These operations are required for commercial applications, such as databases.

7. The processor is able to handle both big and little endian byte and bit ordering, where big endian is the default. Big endian notation addresses multi-byte fields in ascending order from left to right, resulting in the most significant byte always having the lowest address. Little endian notation reverses bit and byte addressing notations. Because the PowerPC-601 can

handle both byte and bit ordering means that it can support Intel-processor based operating systems and applications. The software can set, at start-up time, which mode will be run.

5.2 Application Binary Compatibility

Binary compatibility is the ability to take a compiled program (the executable) from one RISC System/6000 running a release of AIX to another RISC System/6000 running a later version/release of AIX and successfully run the program without having to recompile it or make source code changes. Binary compatibility is impacted by many things:

- The hardware itself
- The operating system
- The APIs (for example, libraries)

Since its introduction in June 1990, thousands of applications have been ported to the RISC System/6000. The binary compatibility of these applications has been preserved, despite the introduction of many significant hardware and software enhancements.

5.2.1 Hardware Compatibility

Customers with any combination of POWER-, PowerPC 601- and POWER2-based systems may continue to run their existing binaries as is.

All applications will migrate to POWER2 and most will migrate to PowerPC-601, as is, without the need for recompilation because, in most cases, differences in the hardware are masked by the operating system environment and compilers and will not be seen by the application.

5.2.1.1 POWER2 Compatibility

All AIX applications written for POWER-based RISC System/6000s (models available before 9/21/93) will run on the POWER2-based RISC System/6000s (for example, 58H, 590 and 990) without recompilation. They should run significantly faster because the POWER2 has a faster clock (66.5 MHz and 71.5 MHz), larger cache, and dual integer and floating point units.

Additional performance can be achieved by recompiling with the C for AIX compiler to further exploit the POWER2 technology. Double precision floating point applications will have more to gain from targeting the compilation for POWER2. The new POWER2 instructions have no impact on integer performance. Generally, only certain applications that manipulate sequential double precision floating point data will benefit from them.

Applications compiled using a compiler option to exploit POWER2 technology will not run on non-POWER2 based systems. (This is because, if the compiler generates instructions unique to the POWER2 machine, 11 of the instructions are not supported on POWER, and 8 of the instructions are not supported on PowerPC.)

5.2.1.2 PowerPC-601 Compatibility

All AIX applications written for POWER-based RISC System/6000s will run on the PowerPC 601-based RISC System/6000s (for example, 250, 25S, 25T, and 25W) without recompilation. (The only exception to this statement would be applications written using kernel extensions that use certain privileged assembly instructions. IBM is not aware of any applications using these instructions.)

Additional performance can be achieved by recompiling with the C for AIX compiler to further exploit the PowerPC technology, particularly those applications dominated by single precision floating point computations.

Applications compiled using a compiler option to exploit the PowerPC technology may not function properly on systems that do not employ that technology (POWER and/or POWER2).

5.2.2 New Applications and New Releases of Applications

New applications and recompilation of existing applications for use in a mixed processor environment should be compiled using the common mode options available in the C for AIX compiler or other compilers that support those options. See 6.1.2, "Compiler Modes" on page 14 for a description of common mode.

5.2.3 Backward Compatibility

Backward migration of PowerPC-specific applications to the POWER architecture is not possible, since the POWER processor does not support the new PowerPC instructions and simulation of the new PowerPC instructions is not available for POWER-based machines.

5.3 Instruction Set Differences

This section describes the instruction set differences between the POWER, POWER2 and PowerPC-601. Differences between the various PowerPC implementations are also discussed to avoid future migration problems among the various PowerPC chips.

The differences between POWER and PowerPC instruction sets fall into several categories:

- POWER instructions not implemented on the PowerPC. Generally, these instructions are emulated by the operating system.
- New instructions added to the PowerPC architecture. These instructions are not available on POWER and are not emulated by the operating system.
- Some instructions which are "user mode" operations on POWER are now "priviledged" instructions on the PowerPC.
- Some POWER instructions are supported with restrictions on the PowerPC.

Following is a diagram showing how the instruction sets of each of the architectures relate.

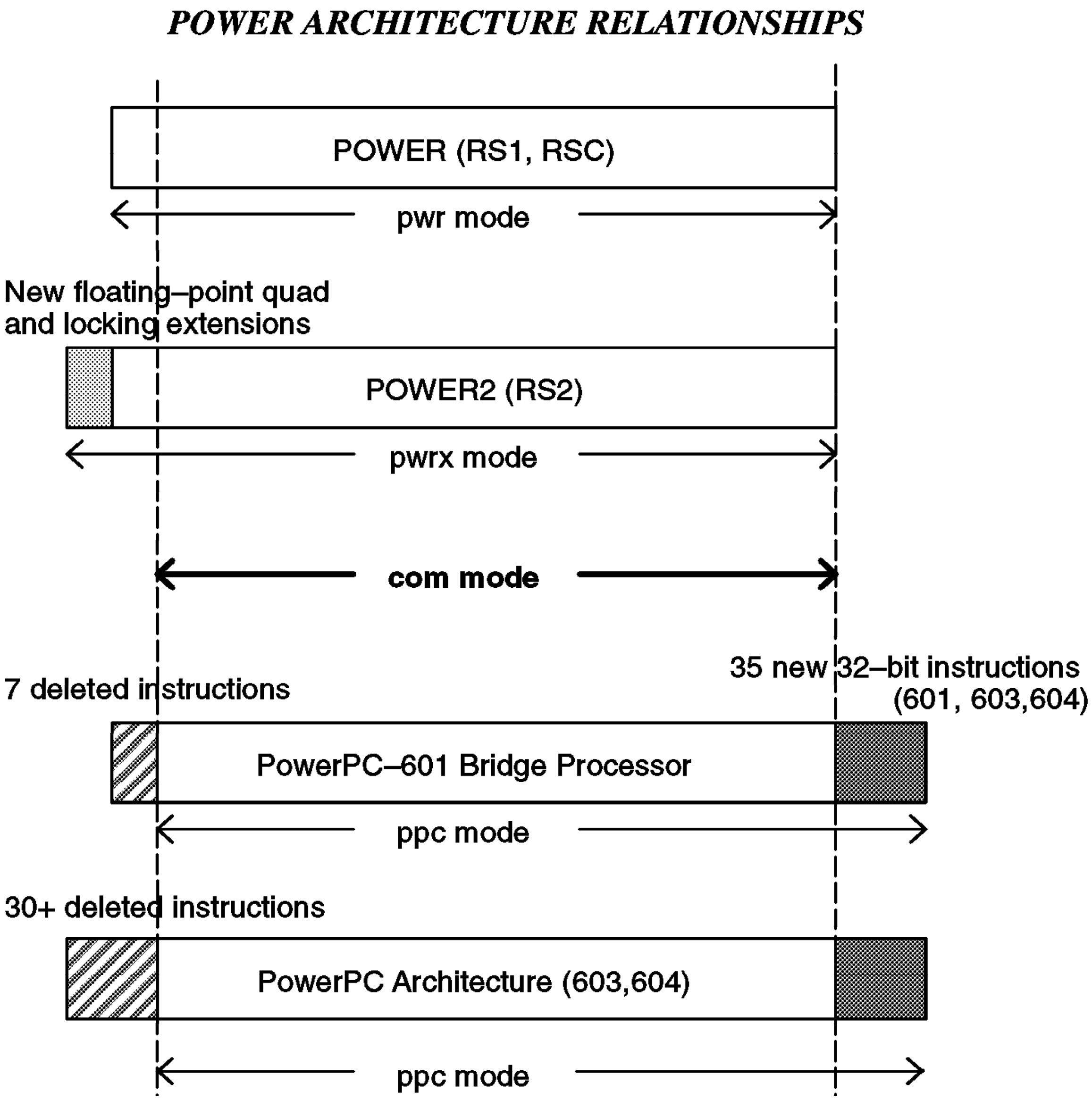

Figure 1. Architecture Instruction Set Relationships

5.3.1 POWER/PowerPC Instruction Set Relationships

Figure 1 shows four rectangles representing the instruction set of each architecture/implementation. The top rectangle, labeled POWER, represents the instruction set of the original POWER architecture, on which all RISC System/6000 systems were based until September of 1993.

The second rectangle represents the new POWER2 implementation. This instruction set is a superset of the POWER instruction set. It contains 11 additional instructions, which deal exclusively with floating point data. Eight of the 11 are for loading/storing quadword floating point.

The third rectangle, labeled PowerPC-601, represents the 601 bridge processor. The PowerPC-601 is the first implementation of the PowerPC architecture. It is intended to smooth the migration from POWER to PowerPC. Even though the PowerPC architecture eliminated over 30 instructions from the POWER instruction set, the PowerPC-601 kept most of them (all but seven, five of which are emulated). This allows all compiled applications to run on a PowerPC-601 at full speed, without the need for software emulation.

The bottom rectangle represents other PowerPC chip implementations. This shows that the PowerPC-603 and PowerPC-604 will support the deleted POWER instructions in Emulation mode only.

Common mode represents the intersection instruction set between POWER and PowerPC. A compiler flag can be used to restrict compilers to this set of instructions, thus allowing any application which is compiled in this mode to run successfully on POWER, POWER2 and PowerPC hardware without the need for emulation.

5.4 Migration Steps

The following flow chart illustrates the steps to be taken if you are migrating your code to a PowerPC-601 machine.

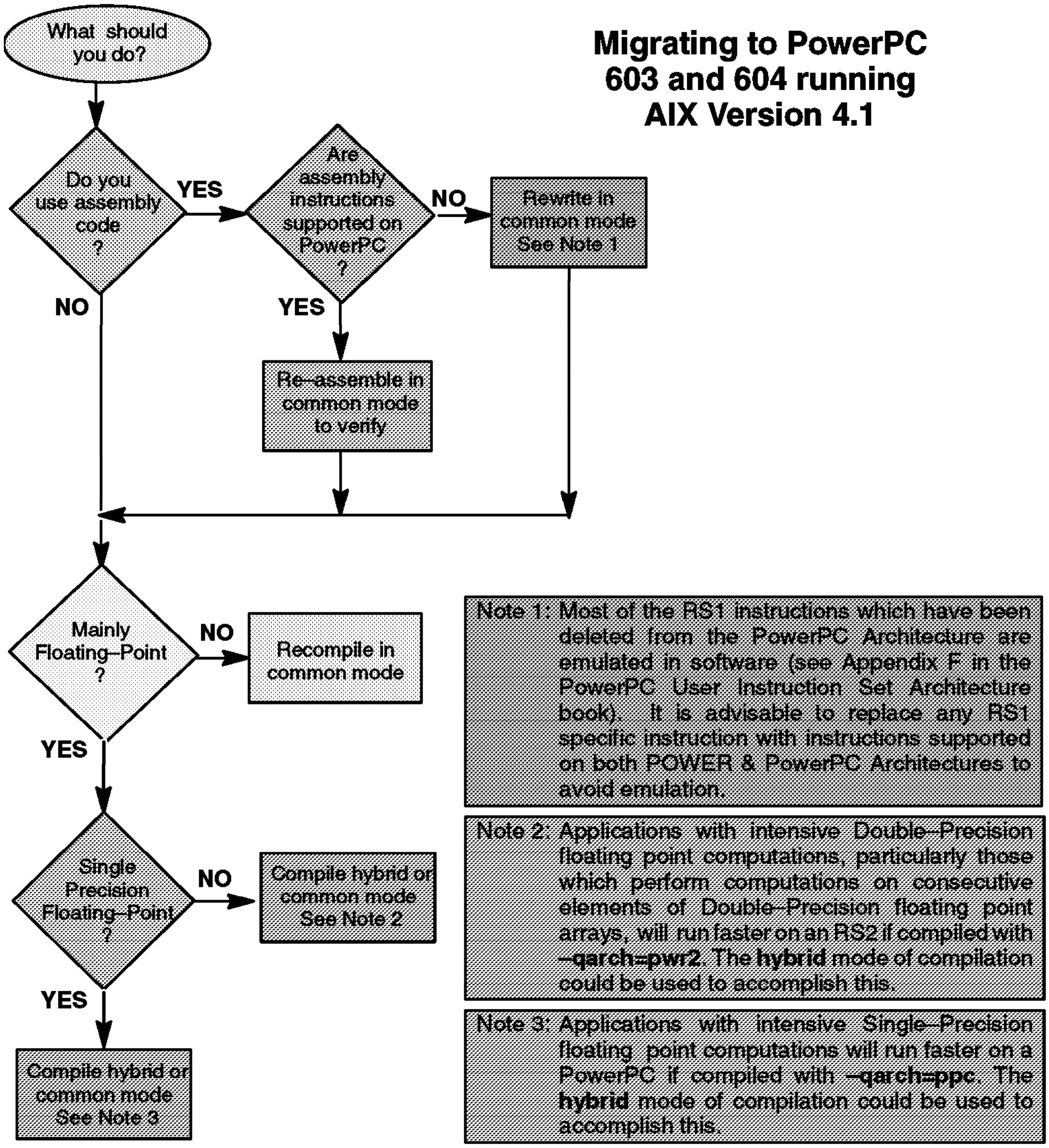

Figure 2. Migration to PowerPC-603 and 604

Chapter 6. Compiler, Loader, and Linker/Binder Changes

This sections deals with changes to the compilers, the loader and the linker.

6.1 Compiler Changes

6.1.1 C for AIX Replaces XL C Compiler in AIX Version 4.1

In AIX Version 4.1, the C for AIX compiler is available instead of the XL C compiler shipped in AIX Version 3.2. The XL C compiler will not be available or supported on AIX Version 4.1, so care should be taken that it is not migrated onto an AIX Version 4.1 system.

6.1.1.1 C for AIX and XL C Compatibility

The C for AIX compiler is a stricter compiler than the XL C compiler. The C for AIX compiler is not fully compatible with XL C compiler in the following cases:

1. The C for AIX compiler implements tighter ANSI conformance. For example, the following code is accepted by the XL C compiler, but not by the C for AIX compiler:

   - EXAMPLE 1: Mixing K&R (**The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie) and ANSI style function prototypes

     ```
     void func();
     void func(float f){ ..... }
     ```

     Accepting this code leads to run time problems since the float parameter in the definition is first promoted to double (default argument promotions). When the float argument is passed in, the wrong size registers are taken off the stack.

   - EXAMPLE 2: Null dimension multi-dimensional arrays are not allowed. Arrays of incomplete arrays are not allowed, not even on parameters:

     ```
     void f(int p[][]);
     ```

     All dimensions except the first must be specified for a multi-dimensional array. In the above example, ″p″ is defined to be an incomplete array of an incomplete type (an incomplete array of an incomplete array of int).

   - EXAMPLE 3: Tags introduced at parameter scope are not exported to the enclosing non-parameter scope.

     ```
     int f(struct a *);
     struct a { int a; };
     int f(struct a* i ) { return i->a; }
     ```

     The type ″struct a″ was introduced in a parameter list, and will go out of scope at the end of the function declaration or definition.

2. The C for AIX compiler may differ from XL C compiler in implementation defined parts of the language.

   Apparent order of evaluation of arguments to functions with assignments will generate different results with XL C Version 1.3 and C for AIX. Other industry compilers may generate even different results. This code is not portable and should be avoided.

```
#include <stdio.h>

void func(int i, int j, int k)
{
    printf(†i = %d, j = %d, k = %d\n†,i,j,k);
}

main()
{
    int r=1;
    int c=0;

    func(c=r,r,r=4);
}
```

With XL C Version 1.3, the results are:

```
i = 1 j = 1 k = 4
```

With C for AIX, the results are:

```
i = 1 j = 4 k = 4
```

- Preprocessor differences include:
  - Output preserves coordinates of each token.
  - No redundant #LINE directives or multiple blank lines in output.
  - Erroneous/incomplete macro invocations are expanded anyway. This includes:

```
—— Extra arguments (ignored)
—— Empty arguments
—— Missing arguments (treated as empty)
—— Incomplete argument list (treated as completed)
```

3. The C for AIX compiler differs from the XL C compiler where provisions for future extensions are implemented.
   - The C for AIX compiler requires that #pragma options align=*suboption* appears before the structure definition.

     The XL C compiler allowed the #pragma to appear anywhere before the last closing brace of the structure definition.
   - long long type is supported, even under ANSI mode. long long is recognized by all of the invocation commands except for **ic89** and **c89**, which disable it by specifying the -qnolonglong option.
   - Invoking the compiler does not define _ANSI_C_SOURCE, (unless you use the **ic89** or **c89** invocation commands). This is because there are many useful definitions in header files which are not detected when _ANSI_C_SOURCE is in effect. You can define _ANSI_C_SOURCE by supplying -D_ANSI_C_SOURCE at compiler invocation.
4. Indexing arrays out of bounds (either too high or too low (negative)). The C for AIX compiler does not pad arrays like the XL C compiler did, so watch for using array[x], where x is the size of the array (since the valid array indices are 0 to x-1). In addition, reading from an array further out of bounds may pick up different content than it used to, and writing further out of bounds may overwrite different data than it used to.
5. Using uninitialized variables. The content in uninitialized variables may be different between the two compilers.

6. Doing signed comparisons with pointers. If you have code that does something like 'if (p < 0)' where p is a pointer, the C for AIX compiler will optimize out this test because p can never be negative. The XL C compiler would treat this as a signed comparison.

6.1.2 Compiler Modes

The XL C compiler shipped with AIX Version 3.2.5 and C for AIX compiler have introduced several new flags to target code generation to the various instruction sets discussed earlier in this document. Following is the description of all the currently supported flags and their impact on binary compatibility and performance:

POWER (RS1) mode -qarch=pwr: This flag targets code generation to the POWER instruction set. Applications compiled with this flag will run on POWER, POWER2 and PowerPC-601 without the need for emulation. However, software emulation may be required when running on future PowerPC implementations, such as the 603 and 604.

POWER2 (RS2) mode -qarch=pwr2: This flag targets code generation to the POWER2 instruction set. Applications compiled with this flag will **only** run on POWER2 based systems if any of the unique POWER2 instructions are generated because there is no software emulation provided for these instructions on any other hardware.

PowerPC mode -qarch=ppc: This flag targets code generation to the PowerPC instruction set. Applications compiled with this flag will only run on PowerPC based system (601, 603, and 604) because there is no software emulation provided for the new PowerPC instructions on POWER-based and POWER2-based systems.

Common mode -qarch=com: In XL C on AIX Version 3.2.5, this flag generates no millicode calls (it is not "true" common mode). However, on AIX Version 4.1, this does generate "true" common mode.

Common mode targets code generation to the intersection subset of instructions which are common between POWER, POWER2, and PowerPC.

Applications compiled in "true" common mode will run on all RISC System/6000 platforms, regardless of the processor being used. This mode guarantees binary compatibility across old, new, and future hardware without the need for emulation. It is therefore the recommended mode of compilation, since it will allow software developers to have a single binary of their application that is suitable for all RISC System/6000s.

In this mode, the compiler avoids generating architecture unique instructions. Compiling with these flags may result in a slight performance degradation depending on the type of application and the hardware it executes on.

AIX Version 3.2.5 Common Mode: In AIX Version 3.2.5, the IBM XL C Version 1.3, C Set ++ Version 2.1, XL Fortran Version 3.1 and XL Pascal Version 1.1.3 compilers support the common mode of compilation.

There are 8 PTFs that deliver the full set of static libraries, each keyed to a specific optionally installable subset of AIX. These PTFs allow software vendors to develop common mode applications. By ordering any of the PTFs, you will receive all of the PTFs. The PTF numbers are: U432415, U432416, U432417, U432431, U432447, U432448, U432449, U432450. U432415 is the PTF for bos. The others are for the Optional Program Products.

It is recommended that AIX Version 3.2.5 plus the PTFs above be the development platform for applications intended to run on AIX Version 3.2.x and AIX Version 4.1. The only change necessary to incorporate this feature is to include a **-l** flag to point to the proper static libraries during the bind stage. Complete instructions will be found in the ″Memo to Users″ file of the PTFs.

Note: Common mode applications will not run on AIX releases prior to AIX Version 3.2.3.

Generating Common Mode Applications

To have a single binary for an application that runs on all RISC System/6000s without the need for software emulation, it is recommended that any future release of such an application be compiled in COMMON mode. This can be accomplished as follow:

1. Make sure that you are developing on AIX Version 3.2.5
2. If you are using XL C Version 1.3, C Set++ Version 2.1, or XL Fortran Version 3.1, use the following flags with the **cc** or **xlc** command:

   ```
   -qarch=com -qxflag=useabs
   ```

   The use of the above flags will insure that the compiler generated code will not contain any architecture specific instructions.

   Note: ″-qxflag=useabs″ is not an officially supported compiler flag. It is only meant to be used as a temporary solution. It is recommended that you use the compilers designed for AIX Version 4.1 to build ″true common″ applications.
3. Link the application using a specially provided library consisting of system static libraries compiled in COMMON mode. This library is available by ordering feature 2504 for AIX Version 3.2.5. This library must be specified on the **ld** command line to be searched ahead of all other system libraries using the **-l** ld flag.
4. A special export file, called **mili.exp**, must be created and used during the binding (linking) process. The **mili.exp** file defines symbols for system routines (millicode) that reside at low fixed memory locations. This file must contain the following:

```
#!
*
* millicode routines
*
__mulh                     0x3100
* .__mulh                            0x3100
__mull                     0x3180
* .__mull                            0x3180
__divss                    0x3200
* .__divss                           0x3200
__divus                    0x3280
*   .__divus                         0x3280
__quoss                    0x3300
.* __quoss                           0x3300
__quous                    0x3380
.* __quous                           0x3380
_clear_lock                0x3400
* ._clear_lock                       0x3400
_check_lock                0x3420
.* _check_lock                       0x3420
```

The name of the export file containing the above definitions must be used on the **ld** command using the **-bI** flag as follow:

```
ld -bI:filename .......
```

where filename is the name of the export file.

BINARY COMPATIBILITY: COMPILER MODES

CODE CAN RUN ON:	SOURCE COMPLIED FOR: COMMON	POWER CLASSIC	RS2	PowerPC 32–bit
POWER CLASSIC	OK	OK	—	—
RS2	OK	OK	OK	—
PowerPC 601	OK	OK	—	OK
PowerPC 603, 604	OK	OK (EMULATION)	—	OK

Figure 3. Compiler Modes

6.1.3 Tuning Flags

In addition to the general type of optimizations available in all XL compilers in AIX Version 3.2.5 and the C for AIX compiler, the multiple execution units of the POWER, POWER2 and PowerPC architectures allow for additional improvement in execution speed. This can be achieved by rearranging the sequential order of instructions to allow for more parallel execution among the various execution units. This type of optimization is referred to as **scheduling**. Since the number of execution units and their pipeline depth varies from one implementation to the next, it is important to be able to specify which target CPU is to be favored by the compiler for scheduling. This is accomplished by using the **-qtune** compiler flag.

In general, tuning for one architecture versus another will not result in a significant improvement in performance on the target CPU. It will also not significantly degrade the performance if the code is run on an architecture other than that for which it was tuned.

- -qtune=pwr : This flag directs the compiler to schedule instructions favoring the POWER implementation.
- -qtune=pwr2 : This flag directs the compiler to schedule instructions favoring the POWER2 implementation.
- -qtune=601 : This flag directs the compiler to schedule instructions favoring the 601 PowerPC implementation.

In the future, there will be additional options to specify tuning for the other PowerPC implementations (603 and 604).

If the **-qarch** flag is set to pwr, pwr2, or ppc, the **-qtune** flag will default to pwr, pwr2, or 601 respectively. If the **-qarch=com** flag is set explicitly or implicitly (by default), then the **-qtune** flag will default to pwr, but could be optionally set to a different CPU. The following table indicates the valid combinations of the **-qarch** and **-qtune** flags.

Table 2. Valid -qarch and -qtune Combinations

-qarch	**-qtune=pwr**	**-qtune=pwr2**	**-qtune=601**
com (common)	valid	valid	valid
pwr (POWER)	valid	valid	valid
pwr2 (POWER2)	valid	valid	**invalid**
ppc (PowerPC)	valid	**invalid**	valid

6.1.4 Compiler Mode and Tuning Performance Implications

As has been shown above, there are several processors to target and tune a compilation for, which in turn impacts where the application can run and how efficiently. Compiling and tuning an application for a specific architecture will always produce the best performance when it runs on that architecture, but then the code may not run at all or may run slower on a different processor (because of the need for emulation when it runs on other architectures). Common mode is the only compilation mode that guarantees that the application will run on all processors with close to optimal performance. Any performance degradation resulting from using the common mode will vary depending on the application itself and the hardware that it is running on. Following are some scenarios:

Common Mode Code Running on POWER

There are several POWER instructions available on the PowerPC. All XL compilers in AIX Version 3.2.5 and the C for AIX compiler make use of these instructions when targeting for POWER, but do not generate them when compiling in common mode. The majority of the POWER instructions are generated for operations such as:

- Integer multiply and divide
- Absolute and negative absolute functions
- Min and max functions
- Constant character string handling such as comparing and copying
- Other miscellaneous operations

In analyzing the dynamic instruction traces of several large applications which had been targeted for POWER, the deleted POWER instructions had a frequency of less than one percent. When compiling in common mode, a sequence of two, three or four instructions is substituted for each of the POWER instructions. This means that applications compiled with the **-qarch=com** and running on a POWER based system will run less than three percent slower than if they were compiled with the **-qarch=pwr**. In most cases this performance degradation will be offset by the compilers′ improvements in optimizations. Another benefit of compiling in common mode is the fact that only one binary is needed to run on all processors without emulation (which has a greater negative impact on performance).

Common Mode Code Running on POWER2

The POWER2 instruction set is a superset of the POWER instruction set. For this reason, executing an integer application compiled in common mode on an POWER2 based system has the same effect on performance as it does on POWER. However, POWER2 has 11 additional instructions which the compiler will not take advantage of when compiling in common mode, which potentially has an additional performance impact on certain types of applications.

The new POWER2 instructions deal exclusively with floating point data. Therefore, the performance impact to floating point applications, particularly double-precision floating point that are not targeted to POWER2, could be significant. This could affect applications that:

- Perform computations on consecutive elements of double precision arrays
- Perform a lot of square root operations
- Convert from floating point to integer operations

Common Mode Code Running on PowerPC

The PowerPC Instruction set has several new instructions not available on the POWER or POWER2 architectures. These new instructions cannot be utilized when compiling in common mode. The majority of the new PowerPC instructions deal with:

- Single precision computations
- Square root and convert to integer instructions
- Cache operations and other privileged instructions (not generated by compiler)

Looking at the new PowerPC instructions, it is obvious that they will not have any impact on integer only applications; therefore, we can predict that integer applications running on a PowerPC should have almost the same performance when compiled in common mode as when compiled for the PowerPC.

Performance of double-precision floating point applications compiled in common mode will be slightly impacted if they contain sufficient square-root and convert-to-integer operations. The most negative performance impact of compiling in common mode for the PowerPC is on those applications with extensive single-precision floating point computations. This is because, when compiling in common mode, the compiler generates double-precision computational instructions instead of single-precision instructions (due to the fact that the single-precision computational instructions are not available on POWER nor POWER2). These double-precision instructions take longer to execute on most PowerPC implementations and their result must be converted to single-precision prior to storing it into memory. The performance degradation for this type of applications will vary, but for a single-precision floating point compute intensive application, the performance can be significant.

6.1.5 Hybrid Mode of Compilation

For most applications, particularly those with little or no floating point computations, it is recommended that they be compiled in common mode. The advantages of having a single binary outweighs the very small performance degradation that may result from compiling in common mode. However, certain double-precision floating point applications can benefit from targeting for POWER2 (**-qarch=pwr2** flag) while other single-precision floating point applications can benefit from targeting for the PowerPC (**-qarch=ppc** flag). Of course, targeting for a specific architecture will either restrict the application from running or negatively affect its performance on other processors, both of which are undesirable consequences.

The hybrid mode of compilation allows a software developer to compile those performance critical routines within their application in more than one mode and provide run-time testing to determine which copy of the routine to execute based on the hardware type on which the application is running. The run-time test for determining the hardware type of where the application is running can be accomplished in one of the following ways:

1. Including **/usr/include/sys/systemcfg.h**.

   This include contains the _system_configuration data structure which contains information about the CPU type as well as other useful information about the hardware where the application is running . This include file is NOT AVAILABLE on any AIX release prior to AIX Version 3.2.5. Therefore, applications intended to run on prior releases *cannot* use this method for determining the hardware type. See "Sample CPU Type Determination Using CPU ID Testing" on page 22 for an example.

2. Installing a signal handler for Illegal Instructions (SIGILL).

   Code small assembly routines, each of which issues a processor-specific instruction. Invoke one of the assembly routines. If the SIGILL is not invoked, then the processor type is the one that supports the specific instruction; otherwise, repeat the test by invoking another assembly routine that issues a different processor unique instruction, and so on until the processor type is determined.

   This test should be done once early on in the application. Once the processor type is identified, a variable within the application should be set in order to eliminate the need for repeating the SIGILL test.

   See "Sample CPU Type Determination Using SIGILL Signal" on page 24 for an example of how to identify whether the processor type is an POWER2 or a PowerPC.

6.1.5.1 Run Time Selection Based on CPU Type

Using a hybrid mode of compilation, applications can be built that contain one or more functions which have been compiled with different target-machine options. A run-time test using one of the above methods will provide the necessary information needed to determine which copy of the function to call. Following are some examples:

Dynamic CPU Type Testing with By-pointer Calling

1. For a given function **foo**, create **foo2** as a modified copy of **foo**. Compile each with a different architecture flag (that is, compile **foo** with **-qarch=com** and **foo2** with **-qarch=ppc**).
2. Define a pointer to function (that is, ptr_to_foo)
3. In main, test the CPU type and assign either &foo or &foo2 to **ptr_to_foo**.
4. Invoke the function using ptr_to_foo.

Dynamic CPU Type Testing Without the Use of Pointers

1. For a given function **foo**, create **foo2** as a modified copy of **foo** and compile each with a different architecture flag (that is, compile **foo** with **-qarch=com** and **foo2** with **-qarch=pwr2**).
2. Within function **foo**, test the CPU type and either execute the rest of **foo** or call **foo2**. (**foo2** should not have the test for CPU type or the call to **foo2** in it).

6.1.5.2 Applications That Should Use Hybrid Mode

The hybrid mode of compilation is not recommended unless significant performance advantage can be obtained. The most obvious applications that should consider hybrid mode are those with the following characteristics:

1. Floating point applications where extensive computations on sequential elements of a double-precision floating point array are being performed. Also those with extensive floating point square root, and/or convert from floating point to integer operations. These applications will benefit when targeted to POWER2 by using the **-qarch=pwr2**.

2. Applications with extensive single-precision floating point computations will benefit when targeted to the PowerPC using the **-qarch=ppc**.

Sample CPU Type Determination Using CPU ID Testing

```
/*
**
** Sample program that demonstrates the ability to call different
** routines based on the machine architecture. Using this type of
** CPUID testing requires the availability of the systemcfg.h header
** file which is only available on AIX Version 3.2.5 or later.  Building an
** application on AIX Version 3.2.5 and/or AIX Version 4.1 with this test and then
** trying to run it on prior releases of AIX will fail.
*/

#include <stdio.h>
#include <sys/systemcfg.h>

/********************************************************************/

typedef struct {
     int  Value;
     char *Description;
} ValueToDescriptionTable;

ValueToDescriptionTable ArchitectureDescriptions[] = {
     { POWER_RS,     †PowerClassic† },
     { POWER_PC,     †PowerPC† },
     {       -1,     †UnknownArchitecture† }
};

ValueToDescriptionTable ImplementationDescriptions[] = {
  { POWER_RS1,  †RS1† },
  { POWER_RSC,  †RSC† },
  { POWER_RS2,  †RS2† },
  { POWER_601,  †601† },
  {        -1,  †UnknownImplementation† }
};

/********************************************************************/

void Print601Routine()
{
  printf( †Running on PowerPC (601)\n† );
}

void PrintPowerRSCRoutine()
{
  printf( †Running on RSC Chip\n† );
}

void PrintClassicRoutine()
{
  printf( †Running on PowerClassic\n† );
}

void PrintPower2Routine()
{
  printf( †Running on Power2\n† );
}

/********************************************************************/
```

```
typedef struct {
  void (*PrintYourself)();
  /* Add Other Routine Pointers here */
} DependentRoutines;

typedef struct {
  int                Implementation;
  DependentRoutines  RoutineList;
} SwitchTable;

SwitchTable RoutinesByImplementation[] = {
  { POWER_RS1, { PrintClassicRoutine    } },
  { POWER_RSC, { PrintPowerRSCRoutine } },
  { POWER_RS2, { PrintPower2Routine   } },
  { POWER_601, { Print601Routine      } },
  {        -1, { NULL                 } }
};

DependentRoutines    RoutinePointers = { NULL };

/*******************************************************************/

void SetMachineDependentRoutines( int ThisImplementation )
{
  SwitchTable *p = RoutinesByImplementation;

  while( p->Implementation != -1 ) {
    if( p->Implementation == ThisImplementation ) break;
    ++p;
  }

  RoutinePointers = p->RoutineList;

}

/*******************************************************************/

char *DescriptionFromValue( int                         TargetValue,
                            ValueToDescriptionTable* TargetTablePtr )
{
  while( TargetTablePtr->Value != -1 ) {
    if( TargetTablePtr->Value == TargetValue ) break;
    ++TargetTablePtr;
  }

  return( TargetTablePtr->Description );
}

/*******************************************************************/

main( int argc, char **argv )
{

  printf( †Architecture is %s\n†,
          DescriptionFromValue( _system_configuration.architecture,
                                ArchitectureDescriptions ) );
  printf( †Implementation is %s\n†,
          DescriptionFromValue( _system_configuration.implementation,
```

```
                                ImplementationDescriptions ) );
  printf( †Processor is %d bits\n†, _system_configuration.width );

  SetMachineDependentRoutines( _system_configuration.implementation );

  if( RoutinePointers.PrintYourself != NULL )
    (*RoutinePointers.PrintYourself)();
}
```

Sample CPU Type Determination Using SIGILL Signal

```
#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>

#define RS2_CPU 2
#define PPC_CPU 3

int cpu_type;

unsigned int rs2_func[] = {
    0xe0030000, /* lfq 0, 0(3)   Load Float Quadword instruction     */
    0x4e800020  /* bclr          Branch conditional to Link Register */
};

unsigned int ppc_func[] = {
    0xec01102a, /* afs 0, 1,2 Add floating point single-precision */
    0x4e800020  /* bclr   Branch conditional to Link Register     */
};

struct fcndes {                 /* function descriptor */
    int iar;
    int toc;
    int env;
};

jmp_buf context;

void sig_invop() {
    longjmp (context, 1);
}

/*
 * This function tests if it is executing on an POWER2 or PowerPC
 * CPU.  It does this by executing an POWER2 unique instruction.  If
 * it succeeds, then it is an POWER2.  If that fails, then it executes a
 * PowerPC unique instruction.  If it succeeds, then it is a
 * PowerPC.  Otherwise it is an POWER only instruction.  If a SIGILL is
 * generated, then the CPU is not an POWER2.
 */
int what_cpu(struct rs2_func *instr_ptr) {
    struct sigaction action, old_action;
    struct fcndes fds;
    void (*fptr) ();
    int rc;
    double dummy[2];    /* space for lfq */

    /* set up function descriptor to call assembly function.
       This is AIX dependent.  */
```

```
    fds.iar = instr_ptr;
    fds.toc = 0;
    fds.env = 0;
    fptr = (void (*) ()) (&fds);

    /* set up signal handler for Invalid Opcode */
    memset (&action, 0, sizeof(action) );
    action.sa_handler = sig_invop;
    sigaction(SIGILL, &action, &old_action);

    /* run the POWER2 instruction */
    if (setjmp(context)) {
        rc = 0;
    }
    else {
        (*fptr) (dummy);
        rc = 1;
    }

    /* Restore old signal action */
    sigaction (SIGILL, &old_action, NULL) ;
    return(rc);
}

int check_cpu() {
    if (what_cpu(&rs2_func[0]) ) {
        return (RS2_CPU);
    }
    else {
        if (what_cpu(&ppc_func[0]) ) {
            return(PPC_CPU);
        }
        else {
            return(0);
        }
    }
}

main() {
    cpu_type = check_cpu();
    if (cpu_type == RS2_CPU) {
        printf (†Is an RS2\n†); /* for this case invoke the function
                                     compiled with the -qarch=pwr2 */
    }
    else {
      if (cpu_type == PPC_CPU) {
        printf(†Is a PowerPC\n†);
      }
      else {
        printf (†Is an POWER\n†); /*
      }
    }
}
```

6.1.5.3 Optimization Level 3

The C for AIX compiler has significantly enhanced the **-O3** level of optimization. This option allows the compiler to use more aggressive optimization assumptions at the expense of longer compile time and increased compiler memory requirement. Floating point applications which should benefit the most from this aggressive optimization level include those designed not to raise exceptions, not to care about the sign of zero, and not to be sensitive to side effects in precision from re-associating floating point expressions. Applications dominated by memory accesses may also see some improvement from the more aggressive optimization assumptions.

The **-O3** optimization level does have the potential to change the semantics of a program unless the **-qstrict** flag is specified. When **-O3** is used on its own, code that may trigger an exception (loads and floating point calculations) is allowed to be moved by the compiler into areas where they are determined to be profitable, even though the operations may not have been executed in a stricter interpretation of the original program. Refer to the C for AIX and the XL Fortran User's Guide for additional options and details.

6.2 Loader and Linker/Binder Changes

6.2.1.1 General Description of Major Binder Improvements

This binder has the following new features:

- Increased Performance: A new set of internal algorithms are used for the symbolic resolution and other table manipulations to achieve increased process speed. For the testcases used in the development environment, the typical performance gain during program binding is two to five times faster in terms of system and user time usage.
- Data Stored In TOC: When the proper compiler flag (**-qtocdata**) is set, global scalars will be stored directly in a TOC entry. The size of this data type has to be four bytes or less to be stored in a TOC entry. This produces a smaller XCOFF, and allows the code to run faster at execution time due to reduced address indirection.

  When you specify **-qtocdata**, you most likely want to specify **-qdatalocal** also to state that all data items are not-imported so they can be candidates for storage on the TOC. Data items that are actually imported (e.g. errno) will have to be listed in **-qdataimported=errno** so the compiler will not give them a TOC storage class.

  Note: Some third-party debuggers may not work properly with object files generated using this new option.
- Reduced XCOFF Size With **-g**: The AIX Version 4.1 binder, when compared to AIX Version 3.2, produces much smaller executables from objects compiled with debug option. This is achieved by reducing the duplicated stab strings in the generated XCOFF files. This can be reduced further by using the **-bstabcmpct=2** flag.
- TOC Overflow Resolved: The AIX Version 3.2 binder generates XCOFF with a TOC size limit of 64K bytes. The AIX Version 4.1 binder has removed this limitation by creating second level TOC information. A slight performance

tradeoff is expected for an executable with huge TOC (>64K) due to one extra level of indirection for TOC entries beyond 64K limit.

- User Specified LIBPATH for Loader Section: AIX Version 4.1 binder allows user specified LIBPATH to override the path defined by **-L** and LIBPATH when generating the loader section of an XCOFF. This is extremely useful when the development environment and the end-user environment use different search path for shared objects.

- Marking old versions of shared objects in archives: In AIX Version 4.1, an older version of a shared object in an archive can be marked so that it is not visible during linking. The old version can still be loaded, so existing applications will run, but while linking, the old version is ignored. Marking is done with the strip command.

- The format of the address map files has changed. Tools that depend on the format of these files will have to be updated.

6.2.1.2 Binder (ld) Improvements

- The **-u <name>** option: this option is now supported. It was ignored in previous versions of the binder. The **-u** flag keeps an external symbol from being garbage collected.

- New options are listed below. See the **ld** man page or InfoExplorer for detailed information:

  - bbigtoc
  - bkeepfile
  - blibpath
  - bnoentry
  - bnolibpath
  - bnom (negate -m)
  - bnop
  - bnostrip (negate -s)
  - bnov (negate -v)
  - bpT
  - bpD
  - bscalls
  - bsmap
  - bstabcmpct
  - bsxref
  - btarget

- Obsolete options: following is a list of the obsolete options. Upon encountering such an option, an informational message will be printed, and the option is ignored.

  - bi
  - binsert
  - bfilelist
  - bfl
  - bnostrcmpct
  - bstrcmpct
  - Q
  - x

- The **-Z** flag can be repeated, specifying a list of alternates to the default library directories /usr/lib and /lib.

- When the **-bgcbypass:<n>** flag is used, the count is reduced for each duplicate file occurring in the first <n> files specified.
- The **-bnobind** flag will cause the generated list of commands to be written, even if some input files cannot be found or opened. The command list is written to standard output, as well as to a file specified with the **-bbindcmds** flag.
- Better return code handling and message displaying.

6.2.1.3 System Loader Per-Process Shared Library Data Segment

Design changes to the system loader will be introduced in AIX Version 4.1. In this new design, shared library data will be put in a new ′per-process Shared Library data segment′. Previously, shared library data was put in the process private segment(2). It is now put in a new per-process shared library segment(15).

The virtual space for this new segment will be in the address range 0xF0000000 to 0xFFFFFFFF. In previous versions of AIX, shared library data was put in the ′process private segment′ and had addresses in the range 0x20000000 to 0x2FFFFFFF.

By putting shared library data in a separate segment, addressing conflicts for shared library data can be reduced. This results in the need for fewer pre-relocated instances of shared library data, which will result in less page space usage.

This change should have no impact on properly written applications. It is possible that this change will expose application coding errors, which were not exposed in previous versions of AIX. The most common coding error this would expose is an application which casts addresses to signed values. For example consider the following function:

```
char * foo(int i)
/* Function foo
*
* Return code:
*        Address of foobar[i] if i is less than 10
*        -1 otherwise
*/
{
if (i < 10)
    return &foobar[i]
else
    return (char *)-1;
 }
```

Now suppose a call is made to foo() like the following:

```
if ((int)foo(j) < 0)
    fatal_error();
```

This code is in error. AIX Version 4.1 would expose this error; however previous versions of AIX would not expose this error. The reason for the error is that foo() may return a valid address in the range 0xF0000000 to 0xFFFFFFFF. The correct way to write the above code would be the following:

```
if (foo(j) == (char *)-1)
    fatal_error();
```

Function has been added to the system loader that will privately load data for shared objects, if the ′o_modtype′ field of the aouthdr structure of the file is set to the characters ″PL″. This will provide a work around for code that has the above problems. Note, however, that privately loading shared module data will result in performance degradation during the load process. Therefore, code with this problem should be corrected as soon as possible. Shared objects and executable files can be patched so that shared data will be placed in segment two. This can be done by issuing the following command.

```
echo †PL† | dd of=<file_name> bs=1 count=2 seek=68 conv=notrunc
```

where <file_name> is the name of the shared object or executable you wish to patch. This can also be done by using the **-bM:PL** binder flag on cc or ld when creating the executable. (**-bM:sPL** when creating a shared object).

If a shared object is marked for private load, the data for the shared object and all objects that depend on the shared object will be privately loaded. If an executable file is marked for private load, the data for all shared objects required by the executable file will be privately loaded.

6.3 MP Implications on Binary Compatibility

In general, all applications that execute correctly on a Uniprocessor RISC System/6000 running AIX Version 4.1 will run as is on an MP AIX Version 4.1 system, the only exception being those applications which serialize the accessing of shared memory (global data) among multiple running processes or those that share memory with I/O. Typically those applications use semaphores such as ″compare and swap″ cs() to lock a particular piece of memory prior to using it and often use a simple assignment to release the lock. The sequence of events may look like this:

- Obtain the lock for the data; this is done by calling cs() on AIX Version 3.2
- Use the data for which the lock is obtained
- Release the lock; this is done by a simple assignment on AIX Version 3.2

This sequence is not sufficient for a PowerPC MP-based system due to the weakly ordered memory model which requires certain synchronization of buffers and queues. A typical sequence of events required on an MP system follows:

- Obtain the lock and synchronize the Instruction caches. This can be done on AIX Version 4.1 by calling a new system service (millicode) which resides in low memory.
- Use the data for which the lock is obtained
- Synchronize the buffers and release the lock

As can be seen from the above discussion, locking and unlocking for serializing access to shared memory does not require synchronization on a UP system, while it does on an MP system. In addition, new and more efficient user-mode functions that perform the locking and unlocking have been implemented on AIX Version 4.1. These functions ARE NOT available on AIX Version 3.2. In order for an application to perform the most efficient action for all versions of AIX and all hardware architectures (POWER or PowerPC) as well as all system configurations (UP or MP) the application must determine:

- What version of AIX
- What type of hardware
- Whether it is a UP or MP

Based on the above determination the following action can be taken:

Table 3. Locking/Unlocking Actions

	AIX Version 3	**AIX Version 4.1 UP**	**AIX Version 4.1 MP**
Locking	cs(&lock,0,1)	_check_lock(&lock,0,1)	_check_lock(&lock,0,1)
Unlocking	&lock=0	_check_lock(&lock,0,1)	_clear_lock(&lock,0)

```
_check_lock
       Absolute address:      0x3420
       registers killed:      r0, r3-r12, lr

_clear_lock
       Absolute address:      0x3400
       registers killed:      r0, r3-r11, lr
```

See following sample.

Sample Program for Determining AIX Version and Number of Processors

Following is an example of how you can determine which version of AIX is running and whether it is a UP or MP system and how to set up the proper calls for locking and unlocking.

```
#define _SC_NPROCESSORS_CONF                  71
#define _SC_NPROCESSORS_ONLN                  72

int lockword;
int AIX_MP;  /* This value is set to 0 if a UP system or to 1 if an MP */
int (*csptr) ();
int *latch;
int cs();
static struct {
    int w1;
    int w2;
    int w3;
} check_lock_des = {
    0x3420, 0, 0 };

main()
{
    int i;
    /*
*
* The call to sysconf() will return the following:
*       -1 if AIX Version 3.2
*        1 if AIX Version 4.1 and only one processor (UP)
*        n where 1 <= n <= M. M is the number of processors in an MP
*          configuration. M changes over time.
*
*/
    i= sysconf(_SC_NPROCESSORS_CONF);
    AIX_MP = 0;               /* Assume this is a UP system for now */
    if ( i== -1)
        csptr=cs;             /* This is AIX Version 3.2 system */
    else {
        csptr = (int (*)())(&check_lock_des);
                              /* This is AIX Version 4.1, which can be UP or MP */
        if (i > 1)
            AIX_MP = 1;       /* This is an MP system */
    }
    latch=&lockword;
    *latch = 0;
    get_lock(latch);
    free_lock (latch, 0);
}
```

6.3.1 Sample Coding for Using Semaphores

For an application to have a single source/binary that executes correctly and efficiently on AIX Version 3.2 and AIX Version 4.1 UP and MP it must use the proper locking/unlocking semaphores that work in all of these environments. This will most likely require source modifications, the extent of those modifications will largely depend on the number of individual references to these semaphores. If for example, all the locking/unlocking in the application is funneled through one or two functions then the amount of source modification is small since it is centralized in one place. Below are sample functions of **get_lock** and **free_lock** that can be called from the application to funnel all lock/unlock calls to system dependent services. These two functions assume that it has already been determined whether the application is running on AIX Version 3.2 or AIX Version 4.1 and if running on AIX Version 4.1 whether it is a UP or MP system (see "Sample Program for Determining AIX Version and Number of Processors" on page 31).

The _check_lock() Primitive

Import and export fences are special processor dependent synchronization instructions. They temporarily block other reads and writes to these locations. They protect against concurrent access by several processors, and against the read and write reordering performed by modern processors.

_check_lock() is an AIX Version 4.1 service, similar to the compare_and_swap() service cs() in AIX Version 3.2, with the addition of an appropriate import fence when the swap is successful. It differs from the cs() service in that it is restricted to word aligned accesses only.

Calling semantics:

```
#include <sys/atomic_op.h>
boolean_t _check_lock(atomic_p word, int old_value, int new_value)
```

The _clear_lock() Primitive

This service is basically an atomic store with the appropriate export fence.

Calling semantics:

```
#include <sys/atomic_op.h>
void _clear_lock(atomic_p word, int value)
```

For all UP systems, this routine is a simple store. For PowerPC architectures in real MP systems, this is preceded by the appropriate export fence. This routine does not need to execute disabled. The export fence guarantees that the data being protected is visible to all other processors prior to the lock word being released.

The _safe_fetch() Primitive Atomically reads a single word variable, issuing an import fence for multi-processor systems.

Calling semantics:

```
#include <sys/atomic_op.h>
int _safe_fetch (word_addr)
atomic_p word_addr;
```

Sample Program for Locking Semaphores

```
# This is a sample program that can be called from the application program
# through which all †locking† semaphores can be funneled.  It is useful
# for applications that prefer a single source for both AIX Version 3.2 and
# AIX Version 4.1.  The C prototype for this function is as follows:
#        int get_lock ( int *);
#
# Function: Call the cs or _check_lock routine.  It assumes the global
#           pointer variable csptr has been set to point to either cs
#           or _check_lock based on whether the application is running on
#           AIX Version 3 or AIX Version 4.1
# Input: Register R3 contains the address of the lock
#
# This routine assumes the lock word is at least WORD ALIGNED
# This function uses the Count Register (CTR) to branch to the appropriate
# function.
# It does not destroy the Link Register, therefore the function being called
# will not return to this routine, it will directly return to the caller
# of this routine.
# This function passes R3 as it receives it and sets up the second and third
# parameters to the values 0 and 1 and passes them in R4 and R5.
# The values of 0 and 1 are only a sample and can be substituted with any
# other value that the application requires.

        .file †get_lock.s†
        .csect get_lock[ds]
        .long  .get_lock[PR]
        .long TOC[tc0]
        .long 0
        .toc
T.get_lock:  .tc .get_lock[tc], get_lock[ds]
        .globl   .get_lock[PR]
        .csect   .get_lock[PR]
        .set   r0,0
        .set   r2,2
        .set   r3,3
        .set   r4,4
        .set   r5,5
        .set   r12,12
#
#
         l     r12,T_csptr(r2)      # Load address of descriptor address
         cal   r4,0(r0)             # Set second parameter to 0
         l     r12,0(r12)           # Load descriptor address
         l     r12,0(r12)           # Load code address of code from descriptor
         mtctr r12                  # Move code addr. to LR for branching
         cal   r5,1(r0)             # Set third parameter to 1
         bctr                       # Branch to the cs or _check_lock routine
#
#
         .toc
T_csptr: .tc csptr[tc], csptr[rw]
         .extern  csptr[rw]
```

Sample Program for UnLocking Semaphores

```
# This is a sample program that can be called from the application program
# through which all †unlocking† semaphores can be funneled.  The is †C†
# prototype for this function is as follows:
#        void free_lock (int *, int);
#
# Function: Free a lock
# Input: Address of lock (in Register 3)
# Output: Sets lock to a desired value if running on a Uni Processor
#         Or calls †_clear_lock† millicode routine which stores the value in
#         lock and synchronizes the data caches if running on an MP
# In this example the value being used is 0, but that can be any other value
# that the application may choose.
#
        .file †free_lock.s†
        .csect free_lock[ds]
        .long  .free_lock[PR]
        .long TOC[tc0]
        .long 0
        .toc
T.free_lock: .tc  .free_lock[tc], free_lock[ds]
        .globl  .free_lock[PR]
        .csect   .free_lock[PR]
        .set   r0,0
        .set   r2,2
        .set   r3,3
        .set   r4,4
        .set   r5,5
#
#
        l     r5,T_AIX_MP(r2)  # Get AIX information. This field should
                               # have been set to 1 if the system is MP
        cal   r4,0(r0)         # load zero in register 4
        cmpli 0,r5,1           # Check if system is an MP system
        beq   0,lab0           # branch if an MP system to lab0
        st    r4,0(r3)         # else zero the latch word
        blr                    # return
lab0:
        ba    0x00003400       # Branch to _clear_lock code in low memory
                               # The code for _clear_lock will return directly
                               # to the caller of this routine using the
                               # Link Register (LR) since it was not altered
                               # by this routine.
                               # At this point R3 has the address of the lock and
                               # R4 has the value to store which in this case is 0
#
            .align 2
            .toc
T_AIX_MP:   .tc AIX_MP[tc], AIX_MP[rw]
            .extern AIX_MP[rw]
```

6.4 Core File Format

The core file format has been modified to accomodate multi-threaded processes.

A core file may be dumped when a process is terminated due to a signal. This core file contains enough information to allow the cause of the process termination to be found. The information usually dumped includes the process's active machine state and user mode stack. An application can specify whether or not the full data area including heap is dumped into the core file. Full core dumps can also be made the default on a system wide basis with smit "System Environments".

In AIX Version 3.2, a core file always contains a core_dump structure which provides basic information. This structure has been changed for AIX Version 4.1.

Because of the changes to the data structures and pthreads interface necessary to support multi-threaded processes, the core_dump structure is not identical to that in AIX Version 3.2. However, the structure of the core file allows debuggers to retrieve register information and, for single-threaded applications, stack data. In the case where an application is multi-threaded, a full core dump is required to contain the stack data for all the user level threads.

Chapter 7. Programming Changes

7.1 STREAMS

7.1.1 OSF Thread-Safe STREAMS

An SVID 3 compliant implementation of System V STREAMS is included in AIX Version 4.1.

7.1.2 str_install Function

STREAMS-based device drivers use the str_install() function to initialize the STREAMS environment. The devswadd() function is invoked as part of this process to add the entry in the devsw table for the invoking device driver. No device driver specific entries are made. The config and dump pointers in the devsw table are set to nodev by the STREAMS frame work.

Functions that use devswqry() to find the config or the dump entry points of a device driver will fail causing unpredictable results.

A new function, devswchg(), has been added that allows device drivers to update the devsw table entries after the devswadd() or str_install().

7.2 Low-Function Terminal (LFT)

The low-function terminal (LFT) is the replacement of the high-function terminal (HFT) and is a STREAMS-based device driver. The LFT will interface with the STREAMS-based tty subsystem. LFT will provide many of the functions provided by HFT, with the following noteworthy exceptions:

1. No virtual terminal (or hotkey) support
2. No support for IBM-specific extensions to the ANSI 3.64 datastream. Applications using these IBM specific extensions will not work with the LFT but will work with **aixterm**.
3. No HFT-like monitor-mode support will be provided. Applications (such as, AIXwindows and diagnostics) that used the monitor mode support will need to use the interface provided by the LFT subsystem.
4. Removal of HFT specific commands
5. Full support for configuration-related commands, such as **mkdev** and **chdev**.
6. Only one color. Only one font and keyboard table are supported (both the font and the keyboard table can be changed, but require an IPL for the change to become effective).
7. In a multi-head system in the ASCII mode, LFT will display at the default display *only*. Users can use the **chdisp** command to change the default display.
8. The only input device supported by LFT is the keyboard. Other input devices, such as mouse, tablet, and dials, will be supported only through AIXwindows.

LFT is a low-function enablement feature. It is intended for use only during system startup, installation and stand-alone diagnostics. LFT provides a simple character oriented, tty-like, terminal emulation. Whether a user process is communicating with a tty device (such as VT100) or the LFT is transparent to the user application.

LFT is not intended to be used in a steady-state RISC System/6000 running AIX. Customers are expected to use the AIXwindows package to obtain the full-function console on the native display. This statement is based on the reasoning that a customer would not purchase a graphics adapter unless that customer intended to use it for graphics.

LFT supports all graphics adapters and keyboards supported by AIX. LFT does not support any other input device, such as a mouse or tablet. In a system configuration with multiple display adapters, the default display is set during system startup.

LFT supports multiple fonts as there is a requirement to support multiple resolution displays. However, fonts cannot be changed once the system is up. The logical screen size will always be 25 lines of 80 characters to a line.

LFT does not provide support for virtual terminals or hotkey. It also does not provide support for the HFT specific extensions to the ANSI 3.64 datastream specification. The support for these extensions are available via **aixterm**.

LFT does not provide any functions for exploiting the graphical capability of the underlying hardware. Nor does it provide any functions for switching from character mode to APA mode (no HFT-like monitor mode support is provided).

The following commands were supported by the HFT but will *not* be supported by the LFT:

- chcolor
- chcursor
- chhwkbd (this is supported by the keyboard device driver now; see 7.9.2, “Input Device Driver Changes” on page 46)
- chkeymap
- chnumvt
- chsound
- gm
- lscolor
- lsscreen
- mkkbd
- open
- swkbd

The following commands are changed:

- chdisp – Improved format
- chfont – Does not take effect until next IPL
- chkbd – Does not take effect until next IPL
- lsdisp – Improved format
- lsfont – Improved format
- lskbd – Improved format
- mkfont – Does not take effect until next IPL

Most of the HFT specific IOCTLs will not be supported. These are:

- HFCKEP
- HFCHGLOC
- HFCMON
- HFDSOUND
- HFESOUND
- HFESWKBD
- HFHANDLE
- HFQERROR
- HFQUERY
- HFRCONF
- HFSJKBD
- HFSKBD
- HFSKEP
- HFSMON
- HFTCSMGR
- HFTGETID
- HFTQDEV
- HFTQSMGR
- HFTSBREAK
- HFTSECHO

7.3 Undocumented Interfaces

Kernel extensions and applications (to a lesser extent) that use undocumented interfaces may not run on AIX Version 4.1. The process table or ublock are examples of undocumented interfaces that a kernel extension might use that would then prevent it from running on AIX Version 4.1. Using **/dev/kmem** is an example of an interface that an application extension might use that would prevent it from running on AIX Version 4.1.

7.4 2GB File System Support

uio structure change

As part of a design change to support file systems larger than 2GB, the uio structure has been changed. The uio_offset field of the uio structure is changed to a 64-bit integer. This allows the file system to send I/O requests to any device driver′s read and write entry points which have logical offsets beyond the current 2GB limit. The new uio structure is shown below.

```
struct uio {
    struct     iovec *uio_iov;
    struct     xmem  *uio_xmem;
    int        uio_iovcnt;
    int        uio_iovdcnt;
#ifdef _LONG_LONG
    offset_t   uio_offset;
#else
    int        uio_rsvd;
    off_t      uio_offset;
#endif
    int        uio_resid;
    short      uio_segflg;
    long       uio_fmode;
};
```

Note: The #ifdef _LONG_LONG is present because long long is not permitted in pure ANSI-C mode, langlvl=ansi.

Most device drivers and kernel extensions are unaffected by the change and will work correctly once recompiled. There is potential breakage in a few circumstances.

1. Function Linkage

   If a 64-bit integer is passed as an argument to a function, the receiving function must be properly declared as shown in the following:

   ```
   dd_read(dev_t dev, struct uio *uiop, chan_t chan, int ext)
   {
           foo(uiop->uio_offset,0);
   }

   foo(int a, b)   /* a only got msw of the 64-bit offset.
                      b got lsw of the 64-bit offset instead of
                      0 as the caller intended! */
   {;}


   foo(offset_t a, int b)
   {;}
   ```

   The caller may type cast the argument causing truncation of the most significant 32 bits. Also, the existence of an ANSI-style prototype will cause the compiler to do truncation automatically.

2. Loss of Precision

   Clearly, any attempt to store a 64-bit offset in a 32 bit variable will cause a loss of precision. Generally, this is only significant for device drivers which need to support devices larger than two gigabytes (for example, LVM).

   Care must also be taken when converting block offsets to byte offsets since the byte offset might overflow while the block offset doesn't.

   ```
   blkno = pageno * PSIZE / UBSIZE;
   ```

   In this case, the intermediate calculation overflows.

   ```
   blkno = pageno * (PSIZE / UBSIZE);
   ```

   Similar overflow conditions can occur with byte offsets.

```
int offset;
end_blkno = (offset + XSIZE - 1) >> UBSHIFT;
```

Generally, the calculation can be re-ordered to avoid the overflow.

7.5 32-LUN and 16-SCSI ID Support on IBM SCSI-2 FAST/WIDE Differential Adapter/A

The sc_buf structure in the **/usr/include/sys/scsi.h** has been changed. The resvd6 (reserved) field is replaced with scsi_buf.lun.

OEM device drivers must be recompiled if greater than 7 SCSI IDs or 7 LUNS are desired.

The IDLUN macro in scsi.h previously used 3 bits for the SCSI ID and 4 for the LUN. It now uses 4 for the SCSI ID and 5 for the LUN.

See ″Understanding the sc_buf Structure″ in InfoExplorer for more details.

7.6 AIX Version 4.1 Curses

7.6.1 Porting to AIX Version 4.1 Libcurses

AIX Version 3.2 curses applications will still execute in AIX Version 4.1; however, some modifications may be necessary before the application is re-compiled in AIX Version 4.1.

The AIX Version 3.2 and AIX Version 4.1 libcurses share many common functions. Most of these functions behave similarly, and there is no change in the interfaces to these functions. There are a few exceptions, and these are listed below:

- **touchline**: The AIX Version 4.1 **touchline** marks a range of lines to be refreshed. The AIX Version 3.2 **touchline** marks a range of columns in a specific line to be refreshed.
  - AIX Version 4.1: **touchline**(win,start,count) – Marks the lines start through (start + count -1) as changed.
  - AIX Version 3: **touchline**(win,y,firstcol,numcol) – Marks the firstcol through (firstcol + numcol -1) of line y as changed.
- **delwin**: AIX Version 4.1 returns an error if an attempt is made to delete the parent window before its children have been removed. AIX Version 3.2 warns of this, but does not return an error. (The difference in behavior can be attributed to the difference in the way subwindows are implemented between the two versions.)

  Since it is good programming practice to delete subwindows before deleting the parent, this should not be a major impact for most curses applications.

There are a set of functions in AIX Version 3.2 which are not available in AIX Version 4.1. The functions are listed below along with the replacement functions:

```
Obsolete          Replaced with
(AIX Version 3.2)           (AIX Version 4.1)
---------         ---------------
crmode            cbreak
fixterm           reset_prog_mode
getcap            tgetstr
nocrmode          nocbreak
resetterm         reset_shell_mode
saveterm          def_prog_mode
setterm           setupterm
```

The following functions are obsolete and have no direct replacements in AIX Version 4.1:

```
touchoverlap (use touchline)
flushok
_showstring
```

Note: Extended curses (libcur.a) applications will not have to port this release. Applications can still be re-compiled without modifications.

7.7 Library Changes

7.7.1 Shared Library Attribute

AIX Version 3.2 applications that override the shared attribute for a library member will not run on AIX Version 4.1. There are various private interfaces between library members such as shr.o in libc.a and the kernel. These interfaces may change from release to release. Statically bound binaries will *not* be portable.

7.7.2 librs2.a

The **/usr/lib/librs2.a** library provides statically linked, hardware-specific replacements for the sqrt and itrunc subroutines. These replacement subroutines make use of hardware-specific instructions on some RISC System/6000 models to increase performance.

7.8 Changes That Impact Programmers

The following changes are being made for standards compliance.

catopen: If the name parameter contains a slash, it is assumed to be a full pathname. Previously, only if the name parameter contained a slash in the leading character position would it be assumed to be a pathname.

The second argument to **catopen()** is no longer ignored. If it is 0, the LANG environment variable is used to locate the catalog without regard to the LC_MESSAGES category. If the oflag argument is NL_CAT_LOCALE, the LC_MESSAGES catagory is used to locate the message calalog.

cs: The **cs()** system call is only available to applications. The AIX Version 4.1 version of **cs()** does not function correctly when called from a kernel extension. The **cs()** system call was never documented as callable from a kernel extension. If called from a kernel extension, the **cs()** system call can return to the caller in problem state ″MSR(PR)=1″. The result could be an invalid opcode exception.

The PowerPC architecture defines weakly-ordered memory, which has certain implications on applications running on a multi-processor shared memory system if the application supports global data sharing among multiple processes. Typically, these applications use certain semaphores, such as **cs()**, as an atomic operation for latching (locking) data in order to provide data integrity by serializing the access to it.

execl, execle, execlp, exect, execv, execvp: If the file parameter contains a slash, it is assumed to be a full pathname. The PATH variable is still used if the parameter is not a full path.

Directory STREAMS are closed in the new process image.

For open file descriptors that remain open, all attributes of the open file description, including file locks, remain unchanged.

The state of conversion descriptor and message catalog descriptors in the new process image is undefined. For the new process, the equivalent of setlocale(LC_ALL, ″C″) is executed at startup.

After a successful call to any of the exec functions, any functions previously registered by the **atexit()** function are no longer registered.

The effective user ID and effective group ID of the new process image are saved (as the saved set-user-id and the saved set-group-id) for use by the **setuid()** function.

The number of bytes available for the new process′s combined argument and environment lists is ARG_MAX. ARG_MAX includes the null terminators on the strings; it does not include the pointers.

Any shared memory segments attached to the calling process image will not be attached to the new process image. Each mapped file and shared memory region created with the **mmap()** function is unmapped by a successful call to any of the exec functions, except those regions mapped with the MAP_INHERIT option. Regions mapped with the MAP_INHERIT option remain mapped in the new process image. Applications which rely on the shared memory segments being attached to a new child process image will have to be recoded. This is expected to be a rare occurrence.

lockf: In order for AIX Version 4.1 to be System V Interface Definition (SVID-2) compliant, the **lockf()** system call will now return the errno 9 (EBADF) if a file is opened without, at least, write permission and an application tries to lock that file with the lockf operations, F_TLOCK or F_LOCK.

Currently on AIX Version 3.2, the **lockf()** system call will allow an application to open a file without write permission and subsequently lock this file. The SVID-2 standard specifically states:

```
The file descriptor must have been opened with write-only (O_WRONLY)
or with read/write permission (O_RDWR) to establish a lock with this
function call.
```

Customers who upgrade to AIX Version 4.1 and have applications that call the **lockf()** system call, will have to make sure the file they are trying to lock was opened with write or read/write permissions.

nlist: **nlist()** has been changed to match new naming conventions used by the compiler/loader in the AIX Version 4.1 XCOFF format. These changes may cause applications using **nlist()** to have unusual or incorrect behavior.

The library routine **nlist()** in both libc and libbsd has been changed to return only the C_EXT (external) symbols. The reason for this change is because the AIX Version 4.1 binder puts TOC entries first in the symbol table, which causes **nlist()** to return the address of the TOC entry instead of the symbol. This breaks some commands.

Changing **nlist()** to return only C_EXT symbols may also break applications that use **nlist()** to look at other symbol′s class, but it′s the best solution since **nlist()** isn′t clearly defined as of what class of symbols can be examined. People who use **nlist()** to examine symbols other than C_EXTs will have to provide their own code.

7.8.1 /usr/include/sys/mount.h

The **/usr/include/sys/mount.h** will no longer be shipped. The mount structures are described in **/usr/include/sys/vmount.h**. The only action necessary to correct any problem created by the removal of **/usr/include/sys/mount.h** is to delete the lines of source that refer to it.

7.8.2 XPG4 Changes to Include (Header) Files

The major category of changes made to the include files for AIX Version 4.1 are namespace cleanup issues related to X/OPEN Portability Guide Issue 4 standard conformance. This standard is a superset of the ISO C language standard and the ISO 9945-1:1990 (IEEE POSIX 1003.1) standards, and includes their implementation namespace restrictions as well as adding its own.

Function Prototypes: In AIX Version 3.2, function prototypes are declared as follows:

```
foobar(int a, char *b, float c);
```

This violates the namespace rules in that the variables a, b, and c are all visible in the developer′s namespace. These were all fixed in the listed files to be as follows:

```
foobar(int, char *, float);
```

Macro Variables: Variables in macro definitions also violate the namespace rules, so they are changed from

```
#define FOO(a) bar(a, 0)
```

to become

```
#define FOO(__a) bar (__a, 0)
```

Elements of Structures: The standards describe certain structures, and certain elements within those structures, without restricting the implementation from having more elements in those described structures than required. However, the additional elements cannot impinge upon the developer′s namespace.

This means that for a described structure **Foo**, with described elements **foobar** and **feebar**:

```
struct Foo {
  int foobar;
  int feebar;
  int aix_extra_element;
}
```

becomes

```
struct Foo {
  int foobar;
  int feebar;
#ifdef _ALL_SOURCE
  int aix_extra_element;
#else
  int __aix_extra_element;
#endif
}
```

to keep the structure the same size, while ″hiding″ the additional element name.

Name Hiding: Some items will be moved to different sections of the same include file; for example, from the _POSIX_SOURCE section to the _ALL_SOURCE section.

Other items will be moved to different include files, because they will no longer be allowed for inclusion with a given file. An example is the declaration of ″char * va_list″, which was moved from **stdio.h** to its own include file, **va_list.h**.

List of Files Affected: This is the list of header files affected by the X/OPEN namespace issues.

Table 4. Header Files Affected By X/Open

File Name	**File Name**	**File Name**	**File Name**
assert.h	cpio.h	ctype.h	dirent.h
errno.h	fcntl.h	float.h	fnmatch.h
ftw.h	glob.h	grp.h	iconv.h
langinfo.h	limits.h	locale.h	math.h
monetary.h	nl_types.h	pwd.h	regex.h
regexp.h	search.h	setjmp.h	signal.h
stdarg.h	stddef.h	stdio.h	stdlib.h
string.h	sys/ipc.h	sys/msg.h	sys/sem.h
sys/shm.h	sys/stat.h	sys/times.h	sys/types.h
sys/utsname.h	sys/wait.h	tar.h	termios.h
time.h	ulimit.h	unistd.h	utime.h
.varargs.h	wchar.h	wordexp.h	

7.9 Kernel Extension Changes

7.9.1 Changes to Vnode Interface

A number of substantial changes have been made to the AIX vnode interface which is used by the system to send file system requests to underlying file system implementations. These changes will require existing AIX Version 3 file systems to be ported to the AIX Version 4.1 file system environment. The interfaces are fully documented in the *AIX Version 4.1 Technical Reference: Kernel and Subsystems:e book*; this section merely summarizes the changes.

1. Credentials

   Most of the vnode and vfs operations now take a pointer to a credential structure. This credential structure identifies the user ID and group list which should be used for permission checking by a file system implementation.

2. Large File Systems

   A number of file system data structures and interfaces have been changed to support file systems larger than 2 gigabytes and to facilitate the future introduction of files larger than 2 gigabytes.

   The f_offset field of the file structure is changed to a 64-bit integer.

   The va_size field of the vattr structure is changed to a 64-bit integer.

   A new file lock structure, named struct eflock, is defined. The size and offset fields are declared to be 64-bit integers.

   The vn_lockctl vnode operation now takes a 64-bit offset parameter and the lock data pointer is described by the eflock structure instead of the flock structure.

   The vn_fclear vnode operation now takes 64-bit offset and length parameters.

   The vn_ftruncate operation now takes a 64-bit offset parameter.

3. Read with attributes, lookup with attributes

   The lookup vnode operation and the read/write vnode operation are extended to allow the caller to request post attributes of the vnode which is the target of the operation. Calls through these interfaces which do not desire the attributes should pass NULL in the vattr parameter. Underlying file systems which do not implement this behavior must set the VA_NOTAVAIL in the va_flags field to indicate that attributes have not been returned.

4. Serialization

   In AIX Version 3, the underlying file system routines were always invoked while holding the kernel lock. This allowed the file systems to use the kernel lock for serialization. In AIX Version 4.1, this is no longer true; file systems will need to implement their own serialization scheme. Using the kernel lock at entry to each of the vnode operations would effectively simulate the AIX Version 3 behavior.

7.9.2 Input Device Driver Changes

The input devices consist of the keyboard/sound, mouse, tablet, dials and lighted programmable function keys (LPFK). In AIX Version 3.2, the input devices were not directly exposed to the user. Instead, I/O requests were handled by the HFT subsystem and passed to the input drivers when appropriate. With the introduction of the LFT, this is no longer true.

A programmer using I/O devices will see the following changes:

1. Device logical names (and special file names) are new. Device special files are opened directly. I/O is no longer done through HFT.
2. Interfaces are mapped by the **/usr/include/sys/inputdd.h** header file. The interfaces are 100 percent changed from AIX Version 3.2.
3. The HFQUERY IOCTL has been removed. This IOCTL was used to perform diagnostics. There is a separate package available to test the I/O devices.
4. Alarm IOCTLs are still queued if the device is busy, but the size of the queue has been substantially reduced. In addition, a signal is not provided when the sound request has been serviced. EBUSY is returned when a sound request is made and the queue is full. The AIX Version 3.2 driver put the requesting process to sleep till the request could be queued.
5. ODM entries for input devices have been substantially modified.

Device drivers that replace the input device drivers supplied by AIX Version 3.2 will have to have substantial changes made to their code for AIX Version 4.1.

7.9.3 Device and Ring Queues

AIX Version 4.1 will not support device and ring queues. Device queues were provided for compatibility with previous AIX device drivers.

Device queues in AIX were a method of queuing requests to a kernel process performing the device manager role. Kernel processes used device queues to communicate to other kernel processes.

Device queues were supported as an AIX kernel extension. There were two basic types of queue elements supported by device queues. A queue element sent from the client to the server was referred to as a request and was sent with the enqueue service. A queue element sent from the server to the client was referred to as an acknowledgment and was typically sent with the dequeue Queue Management service.

Four ring queue kernel services were also considered part of the device Queue Management services. They were used in the process environment only. The ring queue kernel services were a light-weight method of interprocess communication. They were typically used as part of a complex I/O subsystem and permitted a device driver to notify and pass data to a kernel process efficiently. Ring queue services were typically used in communications device handlers to notify a kernel process of the availability of received data. As part of the notification process, they also passed the address of the buffer containing the received data.

Only two products were known to use these services: SNA and OSI. Both of these products are IBM products. These products are discontinuing use of the services.

The following can be used to see if a kernel extension uses one of these services. Just replace $1 with the file name.

```
dump -Tvp $1 | untab | cut -c34-37,42-44,68-199 | fgrep IMP |
cut -c5-199 | fgrep -v SV | cut -c4-199 | sort
```

Services to be removed are:

- ackque - Sends an acknowledgement queue element.
- attchq - Creates a path to a device queue.
- canclq - Deletes all pending queue elements from a device queue.
- creatd - Assigns a global name to a device queue.
- creatq - Creates a device queue.
- deque - Performs completion processing for the active queue element.
- detchq - Invalidates the path to a device queue.
- dstryd - Deletes a global name from a device queue.
- dstryq - Destroys the specified device queue.
- enque - Sends a request queue element.
- peekq - Returns a pending queue element in the device queue.
- qryds - Returns information about the device manager.
- queryd - Returns the device identifier associated with the IODN (Input/Output Device Number).
- queryi - Provides information about device queues.
- queryp - Indicates whether a path exists to a device queue.
- readq - Returns the active queue element in the device queue.
- vec_clear - Removes a registered virtual interrupt handler.
- vec_init - Registers a virtual interrupt handler.
- waitq - Waits for a queue element.
- rqc - Creates a ring queue.
- rqd - Deletes a ring queue.
- rqgetw - Gets the next element (word) from the ring queue.
- rqputw - Puts a word into the next available element on the ring queue.

The following files will be removed:

- /lib/drivers/qm
- /lib/qm.exp
- /usr/include/sys/deviceq.h

7.10 Device Drivers Binary Incompatibility

Even though the PowerPC I/O architecture is different from that of POWER, there are few cases that cause the binary incompatibility of a device driver. Some of these cases can be corrected by just recompiling the driver source (if it is coded in a high level language); others may require source modification. Following are all known areas of incompatibilities:

- Address Check and Address Increment

  The PowerPC I/O Architecture removed the ability to disable the functions provided by the Address Check and Address Increment bits of the I/O segment register. The ′Address Check′ bit allowed the disabling of invalid address checking in order to disable the Card Select Feedback exception. Disabling the ′Address Increment′ bit allowed the use of dynamic bus sizing to allow a device driver to perform a 4 byte load/store to a 1 byte port address, where all 4 bytes would be loaded/stored one at a time without incrementing the address (in other words, a fast way to store multiple bytes to the same byte address).

Address Check and Address Increment were considered default operations, but required bits 12 and 13 of the I/O segment register to be set to '1'b. The PowerPC I/O architecture continues to make these operations the default, but provides no means for disabling them.

Device drivers that need the ability to disable Address Checking and/or Address Incrementing will require source modifications.

- IOCC Facilities

  IOCC facilities such as reading the Interrupt Status Register or the DMA Status, among others, can be accomplished by using the system provided Interrupt and DMA Services. This guideline has not always been followed. As a result, all device drivers which directly access these IOCC facilities will have to be modified.

  A device driver accessing any IOCC registers (for example, the IOCC configuration register, component reset register, etc.) must change their code since the addresses are different for the two architectures (POWER and PowerPC). AIX Version 3.2.5 provides macros to determine the architecture at run time. This way, the device driver can use the same source code for both architectures. The macros **__power_pc()** and **__power_rs()**, which tell whether the machine has PowerPC or POWER architecture, are defined in **/usr/include/sys/systemcfg.h** in AIX Version 3.2.5.

- RT Compatibility Bit Device Drivers for adapters that set the RT compatibility bit in the segment register will have to be recompiled. These drivers can be identified by their use of the Bus Memory Attach macro in the **/usr/include/sys/adspace.h** header file.

- Exception Error Status On POWER, error status following a PIO exception was available in Channel Status Register 15. In the PowerPC I/O Architecture, this status will be available in a separate register, the Direct Store Interrupt Error Register.

Chapter 8. User Commands

8.1 Changes That May Cause Problems

Most of the following changes were made for standards compliance.

8.1.1 acctdusg

Previously all files under a user's home directory were counted on the user's account regardless of ownership. Now only files that the user owns are counted on the user's account.

acctdusg reports in 512-byte blocks in AIX Version 3.2.5 and AIX Version 4.1. Previous to AIX Version 3.2.5, it reported in 1024-byte blocks.

Accounting will only be done for files on the local file system for local users. If the system administrator wants to count remote users (such as YP clients or diskless clients), then the -p flag should be used.

8.1.2 admin

The mrlist argument is now required with all uses of the -m flag.

8.1.3 awk

The usage message will be issued and a non-zero value will be returned when invalid flags are encountered. Previously, the invalid flag was ignored and **awk** continued processing.

awk will now count comment lines only once. Previously, comment lines were being counted twice.

In AIX Version 3.2, variable assignments made on the command line were available to the BEGIN procedure. In AIX Version 4.1, such assignments are not available in the BEGIN procedure, in conformance to XPG4 and POSIX requirements which state:

> An assignment before the first file argument will be executed after the BEGIN actions (if any), while an assignment after the last file argument will occur before the END actions (if any).

If variable assignments are required in the BEGIN procedure they must be made using the [-v Variable=Value] option on the command line before the [-f File|Commands] option is specified.

8.1.4 axeb

The **axeb** command is withdrawn. This command is now obsolete. In AIX Version 3.2, the command was not documented but it was provided for compatibility with AIX Version 3.1. The X/Open **iconv** command replaces the functionality provided by this command, so it will be removed for AIX Version 4.1. Applications which use this obsolete command will have to be rewritten.

8.1.5 backbyname

The -d and -s flags are no longer valid with **backbyname**.

8.1.6 bc

Support for the -c and -l flags to be used together has been added.

The <backslash> character immediately followed by a <newline> is ignored as part of a multi-line NUMBER token. The previous implementation treated this sequence as a terminator to the NUMBER.

8.1.7 bosboot

The **bosboot** command, which is used to create boot images, has been changed:

- To support new switches (-q, -U) and
- To remove the following old switches: -c, -f, -m, -o, -s, -u and -z

The reasons for adding new switches are:

- -q switch allows users to query the amount of disk space required to create the boot image, so that enough space can be allocated before running **bosboot**. This will prevent the **bosboot** command from failing due to insufficient disk space.
- -U switch allows the boot image to be created in uncompressed form. To reduce the disk space, the image is compressed by default. -U can be used on machines with memory larger than 8 megabytes and a boot logical volume larger than 4 megabytes to improve bootup time.

A combination of switches (-c, -f, -m, -o, -s, -u and -z) was specified to create the boot image in the **bosboot** command prior to AIX Version 4.1. However, they are no longer needed, since the -a switch will perform all necessary steps as required. Specifically:

- -c switch was used to create a compressed boot image. In AIX Version 4.1, it will be created compressed by default.
- -f switch was used to specify the name of the RAM file system. Since -s is not supported, this switch is no longer needed.
- -m switch was used to create a specified boot image. This switch is used in conjunction with -s and -f switches. This switch is no longer needed since -a switch already provides this function.
- -o switch is removed since it is not a flag but a necessary operation for certain boot devices such as hard disks. **bosboot** will run savebase as needed.
- -s switch was used to create a specified device RAM file system. This switch is of little or no use for the end user and has been removed.
- -u switch was used to specify an update to the list of bootable devices. Running the **bosboot** command had inadvertantly changed the list of boot devices in AIX versions prior to AIX Version 4.1. This side effect was considered undesirable and **bosboot** will no longer update the list of boot devices.

  Note: THE USER MUST RUN THE **bootlist** COMMAND TO UPDATE THE LIST OF BOOT DEVICES.

- -z switch was used to create a specified device compressed RAM file system image (while leaving the kernel uncompressed). This mode is no longer supported since the default (compressed boot image) mode provides similar function.

The **bosboot** command is run only at install/update time and has no impact on the runtime environment of vendors′ code. The changes in the **bosboot** command do not destroy or modify any preserved information.

There are two possible impacts on vendor installation:

1. If it runs the **bosboot** command during its installation using the unsupported switches, **bosboot** will fail. However, in almost all cases, the unsupported switches can be replaced with the switch -a or -aU.
2. If the vendor installation needs to modify the list of boot devices, the **bootlist** command must be run separately from **bosboot**.

8.1.8 bsh

There will be only one version of the Bourne shell distributed under /**usr**/**bin**/**bsh**; it will handle both single-byte and multi-byte data. In AIX Version 3.1 and AIX Version 3.2, both a single-byte and a multi-byte version were distributed under /**usr**/**bin**/**bsh** and /**usr**/**mbin**/**bsh** respectively.

In AIX Version 3.1 and AIX Version 3.2, the default shell /**usr**/**bin**/**sh** was linked to /**usr**/**bin**/**bsh**. In AIX Version 4.1, the default shell /**usr**/**bin**/**sh** is linked to /**usr**/**bin**/**ksh**.

The **bsh** built-in echo will not support the -n flag. Use ″\c″ to suppress the newline at the end of output.

8.1.9 catman

The -p, -n and -w flags are now mutually exclusive.

8.1.10 chuser

In AIX Version 3.2, when changing the pgrp (primary group) of a user, the user remained a member of the old pgrp. In AIX Version 4.1, changing the pgrp (primary group) of a user will result in the user being removed from the old pgrp. Use the groups attribute of the **chuser** command to retain the user in the old pgrp.

In the following examples, the /**usr**/**lib**/**security**/**mkuser.default** file is as follows:

```
user:
        pgrp = staff
        groups = staff
        shell = /usr/bin/ksh
        home = /home/$USER
admin:
        pgrp = system
        groups = system
        shell = /usr/bin/ksh
        home = /home/$USER
```

```
# mkuser yngwie

  No change from AIX Version 3.2.
  User †yngwie† is placed in group †staff†, with primary group of †staff†.


# mkuser -a ozzy

  No change from AIX Version 3.2.
  User †ozzy† is a member of only group †system†, with the primary group set
  to †system† since the †-a† flag was used to indicate admin user.


# mkuser pgrp=system sah

  No change from AIX Version 3.2.
  User †sah† is now a member of two groups, †system† and †staff†, with
  the primary group being †system†.  Group †staff† was added by the
  †groups† attribute in the mkuser.default.


# chuser pgrp=staff ozzy

  User †ozzy† is now taken out of group †system† and added to group †staff†.
  The primary group is now †staff†.
  In AIX Version 3.2, †ozzy† would have remained a member of group †system†.


# chuser pgrp=security groups=staff,security yngwie

  User †yngwie† now has group †security† as the primary group.  The groupset
  now contains both groups †staff† and †security†.
  In AIX Version 3.2, the †groups=staff,security† was not required because
  the user remained a member of group †staff† automatically.
```

8.1.11 cksum

The checksum algorithm has changed. The new algorithm is based on the polynomial used for CRC (cyclic redundancy check) error checking in the Ethernet standard. The checksum produced is completely different, but it is guaranteed to be calculated the same on all POSIX 1003.2 complaint systems. The AIX Version 3.2 **cksum** is not POSIX 1003.2 compliant and produces different values than the AIX Version 4.1 version of **cksum**.

The usage message will now be displayed when issuing **cksum -?**.

If no operand is specified, the pathname and its leading space will be omitted in the output format.

8.1.12 cmp

When no flags are issued with **cmp**, the output will now use the word 'char' instead of the word 'byte' in the C locale.

The -l and -s flags are now mutually exclusive.

8.1.13 cp

The following flags have been changed:

-h Tells **cp** to always act on the symlink itself and not follow or access the file the link references. This reverses the way that -h previously worked.

-R Copies the symlinks instead of recreating them.

The following new flag has been added:

-f If the target file exists but cannot be opened for writing, this flag causes the **cp** command to try to remove the target file before beginning the copy.

8.1.14 cpio

Specifying "." while renaming a file keeps that file's original name intact.

cpio supports archiving of special files. Previously, special files were neither archived nor extracted.

8.1.15 csh

There will be only one version of the C-shell distributed under **/usr/bin/csh**; it will handle both single-byte and multi-byte data. In AIX Version 3.2, both a single-byte and a multi-byte version were distributed under **/usr/bin/csh** and **/usr/mbin/csh** respectively.

The csh built-in command, **alloc**, returned values that were not consistent across system configurations. Portable applications, therefore, could not use the command, and the **alloc** built-in has been removed from the C shell.

8.1.16 ctags

When using the -a flag, the tags file will be sorted.

8.1.17 date

The default output in the C locale will now include the %Z argument as in:

```
†%a %b %e %H:%M:%S %Z %Y†
```

Date has been changed to accept only two input formats as follows:

```
mmddHHMM[.SSyy]        (old AIX default input)

mmddHHMM[yy]           (new POSIX input)
```

The %C, %u and %V field descriptors have been added.

The %E and %O modified field descriptors have been added.

8.1.18 delta

The mrlist argument is now required with all uses of -m.

8.1.19 df

Compliance to POSIX and XPG4 requires that AIX change its default units from 1024-byte units to 512-byte units. Any code that parses the **df** output for disk statistics needs to invoke **df -k** or be changed to use 512-byte units.

The following new flags have been added:

-P Specifies ″portable″ output.

-k Specifies 1024-byte units.

8.1.20 diff

The AIX Version 3.2 version of **diff** had an optional **lines** argument with the -c flag. This argument is no longer supported.

diff only supports text files as input.

8.1.21 diskusg

The output has been changed from displaying 1024-byte to 512-byte blocks.

8.1.22 du

The -a and -s flags are mutually exclusive.

du will always display output for non-directory operands.

The -r flag is the default behavior. The **du** command will always produce error messages about inaccessible directories and files whether the -r flag is specified or not.

du will no longer count each occurrence of a linked file. Files with multiple links (both hard and symbolic) will be counted and written for only one entry.

The following new flag has been added:

-x When evaluating file sizes, evaluate only those files that have the device as the file specified by the File operand.

8.1.23 ebxa

The **ebxa** command is withdrawn. This command is now obsolete. In AIX Version 3.2, the command was not documented but it was provided for compatibility with AIX Version 3.1. The X/Open **iconv** command replaces the functionality provided by this command, so it will be removed for AIX Version 4.1. Applications which use this obsolete command will have to be rewritten.

8.1.24 echo

The /**usr**/**bin**/**echo** command, the **bsh echo** built-in and the **ksh echo** built-in will not support the -n flag, which used to suppress the new line character at the end of the output. Use ″\c″ to suppress the newline at the end of output. Now, any argument following **echo** will be treated as a string.

8.1.25 ed

Input mode is terminated with ″.\n″. AIX Version 3.2 allowed a single period to be entered while in insert mode by preceding it with a backslash (for example, ″\.\n″). This is contrary to XPG4 and POSIX which states that ALL characters with the exception of the ″.\n″ are to be taken literally and inserted into the text. In AIX Version 4.1, the only way to enter a single period while in insert mode is to enter a character followed by a period and then use the substitute command to delete the extra character.

When using the (..,..)l command, long lines are folded at column position 72 and marked with a backslash character and a newline. XPG4 specifies that the end of listed lines must be marked with the dollar character ($).

8.1.26 expr

expr1 / expr2 and expr1 % expr2 will return an error message if expr2 is zero.

A value greater than 2 will be returned if an error occurs with **expr**.

8.1.27 fold

The **fold** utility does not insert a <newline> immediately before or after any <carriage-return>.

8.1.28 fsdb

This command will no longer eliminate words of zeros from the hex/octal/decimal dump facility. This refers to the display formats of **x**, **o** and **e** respectively.

fsdb will no longer be allowed on non-JFS file systems.

8.1.29 getconf

PathName is no longer optional. If a user enters path_var without PathName, it is considered an error. Likewise, if a user enters system_var with PathName, an error message will be printed.

AIX now supports the following path_vars:

- POSIX_CHOWN_RESTRICTED
- POSIX_NO_TRUNC
- POSIX_VDISABLE

All POSIX system-wide configuration variables and system path configuration variables are now supported by **getconf**.

8.1.30 head

This command no longer aborts when it cannot open a file. It continues processing the remaining files.

8.1.31 iconv

The following converters are withdrawn because they are obsolete:

- IBM-850_AIX.flatstr
- AIX.escstr_IBM-850
- IBM-850_AIX.escstr

These converter files are in the AIX Version 3.2 **/usr/lib/nls/loc/iconv** directory. Applications, utilities or commands that use these optional filters will have to change.

8.1.32 istat

The data block address stored in the inode will be printed in hex as opposed to decimal.

8.1.33 join

The -e flag is only valid when used with the -o flag.

Support for **-o 0** has been added.

8.1.34 ksh

There will be only one version of the Korn shell distributed under **/usr/bin/ksh**; it will handle both single-byte and multi-byte data. In release AIX Version 3.1 and AIX Version 3.2, both a single-byte and a multi-byte version were distributed under **/usr/bin/ksh** and **/usr/mbin/ksh** respectively.

The default shell, **/usr/bin/sh**, will be linked to **/usr/bin/ksh** in AIX Version 4.1 because **ksh** is the POSIX-compliant shell. In previous AIX releases, **sh** was linked to **bsh** (Bourne shell). This change may cause unexpected behavior from some user shell scripts.

There will also be a POSIX shell, **/usr/bin/psh**, which will be a link to **/usr/bin/ksh**.

POSIX requires that some of the shell built-in commands be provided as executable utilities in **/usr/bin**. Those executable utilities are: **alias**, **bg**, **cd**, **command**, **false**, **fc**, **fg**, **getopts**, **jobs**, **kill**, **newgrp**, **read**, **true**, **umask**, **unalias** and **wait**. The **false**, **kill**, **newgrp** and **true** utilities already exist. The others will be implemented as shell scripts which will invoke their respective **ksh** built-in.

Added support reserved word **!**. The format for a pipeline is:

```
[!] command1 [ | command2...].
```

If the reserved word **!** does not precede the pipeline, the exit status will be the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command.

The search sequence has been changed from "aliases, built-ins, functions, file system" to "aliases, special built-ins, functions, built-ins, file system". On AIX Version 4.1, **ksh** does not allow the user to create a function with the same name as a special built-in or regular built-in.

An even number of unescaped double-quotes or single-quotes, if any, must occur inside double-quotes, within the string of characters from an enclosed **${** to the matching **}**. A preceding backslash character must be used to escape a literal **{** or **}**.

Parameter 0 (zero) expands to the name of the shell or shell script. In previous AIX releases, parameter 0 was the name of the shell, shell script or function.

The following built-ins were changed to comply to POSIX and XPG4:

8.1.34.1 echo

The built-in **echo** will not support -n flag. Use ″\c″ to suppress the newline at the end of output.

8.1.34.2 export

The following new flag has been added:

-p Writes to standard output the names and values of all exported variables, in the following format:

`†export %s= %s\n†, <name>, <value>`

8.1.34.3 fc

Use stdout format: ″**\t%s\n**″, <**continued-command>** for the <command> consisting of more than one line.

The following new flag has been added:

-s This is the same as [-e -].

8.1.34.4 hash

The following new flag has been added:

-r Forgets all previously remembered utility locations.

8.1.34.5 jobs

The following are new stdout strings for the state field:

```
Stopped (SIGTSTP) replaces Stopped
Stopped (SIGSTOP) replaces Stopped (signal)
Stopped (SIGTTIN) replaces Stopped (tty input)
Stopped (SIGTTOU) replaces Stopped (tty output)
```

8.1.34.6 kill

The following new flag has been added:

-l [**exit_status**] If an exit_status operand is given (usually the ? shell special parameter), the signal name corresponding to the exit_status will be written. If there is no matching signal name, the exit_status will be written.

8.1.34.7 readonly

The following new flag has been added:

-p Writes to standard output the names and values of all readonly variables, in the following format:

`†readonly %s=%s\n†, <name>, <value>`

8.1.34.8 set

The following new flags have been added:

-C This is equivalent to **set -o noclobber**.

-b Causes the shell to notify the user asynchronously of background job completions. Equivalent to set -o notify.

It now supports **set -o notify**, which is equivalent to -b.

8.1.34.9 trap

The format for the **trap** command with no arguments is: ″**trap -- %s %s ...\n**″, **<action>, <condition>.**

Trap on 0 or EXIT: If **trap** is executed inside the body of a function, action is executed after the function completes in the environment that called the function. If **trap** is executed outside the body of a function, action is executed upon exit from **ksh**. In previous releases, no trap outside the body of a function was executed upon exit from the function.

Traps are not shared between the function and invoking scripts. In previous releases, traps other than EXIT and ERR were shared by the function and the invoking script.

8.1.34.10 unalias

The following new flag has been added:

-a Removes all alias definitions from the current shell execution environment.

8.1.34.11 umask

The following new flag has been added:

-S Produces symbolic_mode output.

8.1.34.12 unset

The following new flag has been added:

-v Refers to a variable name that the shell will unset and remove from the environment.

8.1.34.13 wait

The **ksh** now keeps the status of terminated jobs available until the status is requested.

8.1.35 lex

lex now supports extended regular expressions as defined in XPG4. For new code, users may want to take advantage of new enhancements. Also, multi-byte characters can be used in extended regular expressions with no exceptions. In the AIX Version 3.2 version of **lex**, there were some exceptions.

%x (%X) Definitions

Currently, %x (%X) in the definitions section of a **lex** source file is used to change the default number of multi-byte character class output slots. In AIX

Version 4.1, %z (%Z) will be used to change the default number of multi-byte character class output slots.

%x (%X) will be used in AIX Version 4.1 to specify an exclusive start condition. When the scanner generated by **lex** is in a %x state, patterns with no state specified will not be active as opposed to when in a %s state (regular start condition) where such patterns are active.

AIX Version 3.2 **lex** executable code will continue to run correctly on AIX Version 4.1. AIX Version 3.2 **lex** source code with the %x (%X) won't compile correctly on AIX Version 4.1 without this change.

AIX Version 4.1 **lex** executable code will run correctly on AIX Version 3.2 systems. AIX Version 4.1 **lex** source code with the %z (%Z) won't compile correctly on AIX Version 3.2 systems.

Character Ranges

Currently the characters that fall within a range are determined by character values. In AIX Version 4.1, ranges will be evaluated based on the current collating sequence, regardless of character values, and will include multi-character collating elements where present. If the collating sequence and the character sequence are the same for a given locale and range then this change will have no impact on the lexical analyser produced by **lex**.

This is an improvement because **lex** will treat ranges according to the current locale. Also multi-byte characters will now be allowed in ranges whereas they were not in AIX Version 3.2.

This will affect both single-byte and multi-byte language **lex** programs. For example, in the Fr_FR locale, [a-c] on AIX Version 3.2 would include only a, b and c. However, in AIX Version 4.1, [a-c] will include a, A, a-grave, A-grave, a-circumflex, A-circumflex, b, B, c.

Ending Range < Start Range

In AIX Version 4.1, **lex** will no longer accept ranges where the ending range point is lower than the starting range point (based on the current collating sequence). For example, in the En_US locale, [c-a], will cause **lex** to produce an error message and exit.

Existing vendor code with these ranges, which currently compiles on AIX Version 3.2, will fail to compile on AIX Version 4.1.

Special Delimiters

Currently **lex** will accept the following expressions in the source file:

```
[[:]    matching [ or :
[[.]    matching [ or .
[[=]    matching [ or =
```

However, on AIX Version 4.1:

[::], [..], and [==]

will become special delimiters within bracket expressions indicating a character class, a collating symbol, and an equivalence class respectively.

This change is an enhancement since **lex** will support character classes, collating symbols, and equivalence classes in AIX Version 4.1

The change will break expressions with these delimiters. In AIX Version 4.1, the above expressions will result in an error. These characters can still be matched, however, by reversing the sequence, for example:

```
[:[]
```

will not cause an error. **lex** source files which have any of the special delimiters listed above will have to be modified for AIX Version 4.1.

yytext Type

The default type of **yytext** will be changed from an unsigned character array to a character array.

This will only affect C++ users who may have their own functions which process **yytext**. For C users, unsigned char and char are the same. C++ users will either have to cast **yytext** before passing it to a function or modify the function.

8.1.36 locale

The C and POSIX locales are now included in **locale -a** output.

locale output was modified to include mb_cur_min in the CHARMAP category and alt_digits, era_d_fmt, era_year, t_fmt_ampm, era_t_fmt, era_d_t_fmt in the LC_TIME category.

Parameter processing now enforces proper use of flags:

- -a is mutually exclusive of all other flags and cannot be used with a parameter.
- -m is mutually exclusive of all other flags and cannot be used with a parameter.
- -c and -k require the use of a parameter.

8.1.37 localedef

The ability to escape a less-than sign < within a string is now supported (for example, ″test\<″).

The ability to escape an escape character within a symbol is now supported (that is, <\\>).

localedef has been modified to support XPG4 standards which explicity state that comments may only start at the begining of the line. Previously, **localedef** supported comments that started at any logical point on the line.

Ellipsis usage as defined in XPG4 is now fully supported.

Support for the alt_digits, era_t_fmt, era_d_t_fmt XPG4 keywords is now available. Also support for the copy directive and the charclass extension to the LC_CTYPE grammar.

8.1.38 lp

lp now returns a job id when a job is submitted.

The following new flag has been added:

-s Suppress message (job number) from lp.

The default is to return the job number.

8.1.39 lsfs

The previously undocumented -q flag is now documented and changed to report fragment size, block size, and number of bytes per i-node.

8.1.40 lsjfs

The output of **lsjfs** now reports the block size, fragment size, nbpi, logical volume size, and file system size.

8.1.41 man

The following flags have been added:

-c Does not pipe output through the more command.

-t Uses troff instead of nroff to format the reference page.

The following flag has been deleted:

- Pipes the command output using the **more** command with the **-s** flag. This has been deleted because the **-c** flag performs this function.

8.1.42 more

The following new flags have been added:

-e Exits immediately after writing the last line of the last file in the argument list.

-i Performs pattern matching in searches without regard to case.

+command Performs just like the -p command flag.

-t Tagstring Writes the screenful of the file containing the tag named by Tagstring.

:e Examines a new file.

The +g flag is no longer supported. A combination of pG and -w flags will perform this function.

The -l flag is no longer supported.

8.1.43 nice

The **nice** command exits with 126 if the utility specified by the utility operand was found but could not be invoked. The **nice** command exits with 127 if the utility specified by the utility operand was not found.

8.1.44 nm

/usr/bin/ccs/nm and **/usr/ucb/nm** (which called **/usr/bin/ccs/nm**) are combined into one command. The default output as specified by XPG4 is more similar to the BSD format than to the current AIX default. Consequently, the AIX default output format has changed to comply with that specified by XPG4. (Specifically, the symbol type must be represented by a single character and the default output must be sorted alphabetically by symbol name. Also, the XPG4 definition of the type field is a combination of the former **nm** class and section fields). The name field is padded to 20 characters, with longer names overflowing the field unless -T is specified, which will cause the name to be truncated.

The following flags have been added:

-A Writes the full pathname or library name of the object on each line.

-g Writes out only external (global) symbol information.

-P Writes the output in a standard portable output format. With this flag, the output must list, on each line: the library/object name (only if -A is also specified), the symbol name, the type, the value, and the size, separated by whitespace. Without the -P, the same information must be printed but the format is not specified, except that it be in alphabetical order and that the symbol type be a single character. If -P is specified and -t is not, the default numeric format will be hexadecimal.

-t Format Outputs numeric values in either decimal, octal, or hexadecimal, depending on the Format value. Format is one of d (decimal - the default), o (octal), or x (hexadecimal).

-B Prints output in BSD format of <value> <type> <name>. For a degree of regression usability, the BSD output format will still be available to the user. The AT&T formatting will no longer be available.

Previously, -v sorted only external symbols by value and required the -d flag also be specified. This has changed so that whatever the output, it is sorted by value instead of alphabetically (which is now the default for XPG4).

8.1.45 nohup

The **nohup** utility now takes the standard action for all signals except for SIGHUP, which is ignored. **nohup** previously ignored SIGQUIT in addition to SIGHUP.

8.1.46 od

The -s (lowercase) flag which searches for strings of characters ending with a null byte has been changed to -S (uppercase). The -S flag only works with single-byte characters.

The new -s (lowercase) flag interprets words (two-byte units) in signed decimal. This is equivalent to -t d2.

od now attempts to process all input files before terminating. Previously, it terminated on the first file that it could not access.

When the -c flag is selected, other non-printable characters are to be written as one three-digit octal number for each byte in the character.

8.1.47 pack

The following message is now printed to stdout if the file is not packed:

†pack: %s: no saving – file unchanged,† <file>

8.1.48 paste

Entering **paste** without any arguments returns a usage statement. On previous versions it simply returned without taking any action.

8.1.49 pax

The following new flag has been added:

-o Provides information to the implementation to modify the algorithm for extracting or writing files specific to the file format specified by -x.

When either or both of the -r flag and the -w flag are specified as well as the -v flag, **pax** will write the pathnames to stderr in the following format:

†%s\n†,<pathname>

Previously, the format for files that were hard links were written as follows:

†%s\n†,<pathname>,<linkname>

When neither -r nor -w are specified, and -v is not specified, **pax** will write the pathnames to stdout in the following format:

†%s\n†,<pathname>

Previously, the format for files that were hard links were written as follows:

†%s\n†,<pathname>,<linkname>

8.1.50 renice

The syntax of the **renice** command has changed to the following:

```
renice [-n increment] [-g | -p | -u] ID...
```

The increment argument of the -n flag in the new syntax is different than the nice_value of the old syntax. The increment argument will increment the system scheduling priority by the amount specified. Previously nice_value set the system scheduling priority to the number specified.

8.1.51 sed

[**2 addr**]**l** - Write the pattern space to standard output in an unambiguous form. Nonprintable characters shall be written as one three-digit octal number, except for the characters \\,\a,\b \f, \n, \r, \t,and \v, which will be written as the corresponding escape sequence. AIX previously printed them as one three-digit octal number.

Long lines will be folded, with the point of folding indicated by writing <backslash> <newline>; the length at which folding occurs is at column 72 for AIX. The end of each line will be marked with a $.

8.1.52 sort

The following exit values are now returned:

0 All input files were output successfully, or -c was specified and the input file was correctly sorted

1 Under the -c flag, the file was not ordered as specified, or if the -c and -u flags were both specified, two input lines were found with equal keys

>1 An error occurred

8.1.53 strings

The following flags have been added:

-t Format Lists each string preceded by its offset from the start of the file.

-n Number Specifies a minimum string length other than the default of 4 characters. The maximum value of a string length is 4096.

strings now attempts to process all input files before terminating instead of terminating on the first file it cannot access.

8.1.54 tctl

The following flag has been added:

-l reset Sends a bus device reset (BDR) to the tape device. The BDR will only be sent if the device cannot be opened and is not busy

The -e flag is no longer supported.

8.1.55 tee

If a write to any successfully opened file operand fails, writes to other successfully opened file operands and standard output continue, but the exit status is non-zero.

8.1.56 touch

If no -r flag is specified, no -t flag is specified, at least two operands are specified, and the first operand is an eight- or ten-digit decimal integer, the first operand shall be assumed to be a date_time operand. Otherwise, the first operand shall be assumed to be a file operand.

AIX Version 3.2 exited with a non-zero value when using **touch -c** if a file parameter did not exist. AIX Version 4.1 will exit zero when using **touch -c** if the file parameter does not exist; this change is to comply to XPG4 and POSIX.2. When the -c flag is specified, it is not considered an error if the file parameter does not exist.

D_FMT and T_FMT are no longer used to determine the order of month and day in the date specification and the order of hour and minute in the time. The date/time specification must always be specified in the form MMDDhhmm[yy]. This is a requirement of the XPG4 standards.

8.1.57 uniq

The c, d and u flags are now mutually exclusive.

The standard input will be used only if no input_file operand is specified or if input_file is a dash (-). Previously, a dash (-) was read as a filename.

The input file must be a text file. The **uniq** command will display an error message when the input file is a non-text file.

8.1.58 wc

The **wc** command output format changed to comply to XPG4 standards. The counts are now shown in the order of lines, words and bytes regardless of the order the following flags are specified on the command line:

- -w (words)
- -l (lines)
- -c (bytes)

The following new flag has been added:

-m Writes to the standard output the number of characters in each input file.

The -c and -m flags are mutually exclusive.

8.1.59 what

The standard output consists of the following for each file operand:

″%s:\n\t%s\n″,<pathname>,<identification string>

Previously, this format was output if there were multiple files, but did not output the pathname if there was only one file.

8.1.60 yacc

Simultaneous execution of multiple **yacc**s and multiple parsers in the same file are now possible.

The flag to specify an alternate parser to **yacc** is changed from -p(-P) to -y(-Y).

yacc -p Option

The -p flag is currently used with **yacc** to specify an alternate parser instead of **/usr/ccs/lib/yaccpar**. In AIX Version 4.1, -y will be used to specify an alternate parser.

The -p flag will be used to change the prefix of all external names produced by **yacc** from yy to the prefix specified as an argument with the flag.

Shell scripts or make files which use the -p flag will have to be changed.

Chapter 9. Multi-Processing

AIX Version 4.1 is enabled for Symmetric Multi-Processing (SMP) capabilities. The following section is provided as an overview for readers who are unfamiliar with multi-processor concepts.

9.1 Multi-Processor Definitions

Physical and Logical CPU Number

Processors are identified using either physical numbers or logical numbers. Processor numbering always starts with 0 (zero). Physical numbers identify all processors on a system, regardless of their state, whereas logical numbers only identify enabled processors. The Object Data Manager (ODM) names for processors are based on physical numbers with the prefix /**proc**. The table below illustrates these naming schemes for a four-processor system:

Table 5. Processor Naming Conventions

ODM Name	Physical Number	Logical Number	Processor State
/proc0	0	0	Enabled
/proc1	1		Disabled
/proc2	2		Unavailable
/proc3	3	1	Enabled

Generally, all operating system commands and library subroutines use logical numbers to identify processors. The **cpu_state** command is an exception and uses physical processor numbers. It can be used to list system processors, or control which processors are enabled or disabled. In addition to these software controlled states, the unavailable state indicates a hardware failure detected by the system.

Master Processor

The master processor is defined by the value of MP_MASTER in the **/usr/include/sys/processor.h** file and is currently 0. It is not a master processor in the sense of master/slave processing – the term is used only to designate which processor will be the default processor.

The notion of a master processor came out of the design for funneling. In order to run some UP drivers unchanged, their execution had to be "funnelled" (see below) through one specific processor, which is called the MP master.

Funneling:

Funnelled code runs only on the master processor; therefore, the current uni-processor serialization is sufficient. Interrupts for a funnelled device driver will be routed to the MP master CPU. Funnelling is intended to support third party device drivers and low-throughput device drivers. The base kernel will provide binary compatibility for these device drivers.

Funneling only works if all references to the device driver are through the device switch table.

Funneling also works for device driver iodone routines, interrupt handlers, EPOW handlers, off-level interrupt handlers, watchdog timers, and timeout routines.

MP Safe

The device driver runs on any processor. The code for the device driver has been modified to add a code lock to serialize the device drivers execution. This type of device driver is intended for medium-throughput device drivers.

MP Efficient

The device driver runs on any processor. The code for the device driver has been modified to add data locks to serialize the device driver's accesses to devices and data. This type of device driver is intended for high-throughput device drivers.

9.2 MP Scheduler/Dispatcher

Scheduler: AIX scheduling uses a time-sharing, priority-based scheduling algorithm. This algorithm favors processes that do not consume large amounts of the processor, as the amount of processor time used by the process is included in the priority recalculation equation. Fixed priority processes are allowed. The priority of these processes do not change regardless of the amount of processor time used.

There is one global priority-based, multi-level run queue (runq). All threads that are runnable are linked into one of the runq entries. There are currently 128 priorities. The scheduler periodically scans the list of all active processes and recalculates process priority based on the amount of processor time used.

The AIX Version 4.1 scheduler is based on the AIX Version 3.2 scheduler, with the following changes:

- Move from a process-based scheme to a thread-based model. The schedulable entity is the thread for AIX Version 4.1. This requires the runq (run queue) to be thread based. Suspension due to memory thrashing shall remain on a process basis, rather than on a thread basis.
- Ability to bind a process/thread to a processor (Hard Cache Affinity). Threads can be bound to a processor ID. Only the logical CPU numbers are used by the scheduler subsystem; the physical CPU number is never used by the scheduler subsystem.
- Attempt to run process/thread on the same processor (Soft Cache Affinity).
- Support funneling threads.

Dispatcher: The dispatcher determines which process is to be made the current running process.

The AIX Version 4.1 dispatcher has also been changed. AIX Version 4.1 does not guarantee that the thread with the best priority is the one to select. If the thread is bound to another processor, the dispatcher must select the next available thread. This design may require skipping threads on the runq, or even going down to less favored priority levels.

User Interfaces: User interfaces are routines that modify process priority for one reason or another. These include system calls like nice(), setpriority(), getpriority(), setpri(), getpri() and yield() and kernel services like lockl(), which does priority promotion.

The interfaces nice() and setpriority()/getpriority() will remain process based. setpri() will fix the priority of all the threads in the target process and getpri() will return the priority of the first thread in a process.

The yield() subroutine forces the current running process or thread to relinquish use of the processor. If the run queue is empty when the yield() subroutine is called, the calling process or kernel thread is immediately rescheduled. If the calling process has multiple threads, only the calling thread is affected. The process or thread resumes execution after all processes or threads of equal or greater priority are scheduled to run.

The following new user interfaces have been added to **libc** in AIX Version 4.1:

- bindprocessor() – Binds a process or thread to a processor.
- sched_setparam() – Sets the scheduling parameters for a process.
- sched_getparam() – Gets the scheduling parameters for a process.
- sched_setscheduler() – Sets the scheduling policy and parameters for a process.
- sched_getscheduler()- Gets the scheduling policy and parameters for a process.
- sched_yield() – Forces the calling process to relinquish the CPU until it becomes the head of its process list again.
- sched_get_priority_max() – Returns the appropriate maximum priority for the specified scheduling policy.
- sched_get_priority_min() – Returns the appropriate minimum priority for the specified scheduling policy.
- sched_rr_get_interval() – Returns the current execution time quantum for the process specified.

The following new kernel service is available:

- bindprocessor() – Binds a process or kernel thread to a processor.

MP Commands: The following new commands have been added for MP support:

- **cpu_state** – Controls and lists which processors will be active when the system is next started
- **bindprocessor** – Binds a process to a processor.

Chapter 10. Multi-Processor Synchronization

10.1 Introduction to Synchronization on Multi-Processors

AIX Version 4.1 introduces the capabilities of running on a shared memory multi-processor. This introduces new problems of maintaining data consistency.

Consider the classical example of a doubly linked list. Removing an element from the list implies updating two pointers:

- the forward link of the element preceding the one to remove
- the backward link of the element following it.

When only one of the pointers has been updated, the queue is in an inconsistent state. It is easy to figure out that if two processors in a multi-processor system start adding or removing elements from the same queue at the same time, the result can be totally unpredictable.

The code section updating the two pointers is called a *critical section*. Both updates must appear to have been done *atomically*. This is achieved by means of locks.

10.1.1 Uniprocessor Synchronization

The problem of concurrent access to global data is not specific to multi-processors. On a uni-processor, the concurrency is achieved through preemption. A process can be preempted either by another process or by an interrupt routine:

- An application composed of several processes communicating through shared memory must use semaphores to protect itself from *concurrent* modifications of global shared data.
- Kernel programmers familiar with device drivers know that they need to protect any access to data shared between the base level and the interrupt routine. On a uni-processor, this is done by disabling interrupts.
- Unlike traditional UNIX systems, the AIX Version 3 kernel is preemptible. Without special precautions, a process could be preempted in the kernel in the middle of a critical section, and the new running process could enter the same critical section. Protection in AIX Version 4.1 uses the kernel functions lockl()/unlockl().

In AIX Version 4.1, there are three basic types of critical sections:

thread-thread Critical sections shared between threads (no disabling required).

thread-interrupt Critical sections shared between threads and interrupt handlers (disabling required).

back-tracking This type of critical section occurs only in the page fault handler, because the VMM data structures are pageable. The page fault handler can page fault while trying to resolve a page fault.

10.1.2 Multi-Processor Synchronization

Masking interrupts or disabling the processor via i_disable() to serialize with an interrupt handler no longer works in a symmetric multi-processor environment, except for funnelled code. Since the I/O is symmetric and interrupts can be routed to any of the processors in the complex, masking interrupts will not prevent the interrupt routine from executing simultaneously on another processor. Serialization in a symmetric multi-processor system requires the use of locks in addition to i_disable().

10.1.2.1 Introduction to MP Locks

Basically, a lock can be viewed as a bit in memory. If the bit is 0, the lock is free. If the bit is 1, the lock is busy. For a processor to take a lock, the simplest code sequence is:

```
test the lock bit
if lock is free
     then set it to busy
else
     wait for it to become free
```

Since taking a lock requires several operations (read, test, and set the lock bit) this operation would be itself a critical section. That is why multi-processor hardware must provide a way to perform this test-and-set operation atomically with respect to the other processors. This kind of atomic operation is the basic block upon which all the locking primitives are built.

The PowerPC architecture defines the load and reserve (**larx**) and the store conditional (**stcx**) instructions as a means for providing an atomic memory update. This is accomplished by setting a reservation on the load, and checking that the reservation is still valid before the store is performed; a processor will lose the reservation when issuing another larx, when issuing a stcx, when another processor stores a new value in the cache granule at which the reservation was taken or when an interrupt or a context switch occurs.

For all other POWER architectures (being uni-processor) the atomicity of the above sequence is granted by disabling the machine at the beginning, executing the test-and-set operation and then enabling the machine.

```
begin:
     load lock
     if (lock is free)
          try to set it busy
          if (successful)
               we¢ve got the lock
          else
               go to begin [retry]
     else [lock busy]
          wait for the lock to be free
```

Being shared entities, locks cannot be allocated on the kernel stack, but need to be global to all entities actually sharing them.

10.1.2.2 Different Types of MP Locks

When the calling process must wait for the lock to be free, it will be **blocked**. There are two widely used ways to block the calling process:

- Make it loop until another process releases the lock; the caller is said to be **spinning**. This kind of lock is called spin lock. This way of waiting is sometimes called busy wait.
- The caller is put to **sleep**, and the process starts another thread of execution. The caller will be reactivated when the lock is released. This kind of lock is often called wait lock.

From now on, the words block(ing), spin(ning) and sleep(ing) will be used as defined above.

The basic type of lock is used for mutual exclusion: only one process at a time can access the data protected by the lock.

Imagine that data shared among several processes is accessed sometimes for reading, and sometimes for update. Two (or more) processes can read this data concurrently without any consistency problem. But to modify the data, any process must gain exclusive access; otherwise, even a read concurrent with the write may read inconsistent data.

For these cases, where generally reads are much more frequent than updates, a kind of lock allowing shared accesses (for the read) and exclusive access (for the write) improves the parallelism and allows for better MP performance. Such a lock will be called **read shared/write exclusive** lock, or **RW** lock.

10.2 Lock Model Overview

10.2.1 Lock Model

The OSF/1 1.1 locking methodology was used as a model for the AIX Version 4.1 multi-processor lock function. This comprises of two types of locks: simple and complex. The main characteristics of these two types of locks are the following:

Simple Lock Spin, exclusive, nonrecursive lock.

Complex Lock RW, sleep, recursive (on request) lock.

On the basis of these two types of locks, the kernel can protect data both exclusively and nonexclusively.

10.2.2 Simple Locks

Simple locks in AIX Version 4.1 are spin locks that will sleep under certain conditions. This is to prevent a thread from spinning indefinitely. A thread trying to acquire a busy simple lock will behave as follows:

1. Sleep if the owner of the lock is not currently running.
2. Spin indefinitely until the lock is free if the requestor is running, is an interrupt handler, or is running disabled.
3. Spin for a predefined period of time, and then, if lock has not been freed, the thread is put to sleep.

AIX Version 4.1 simple locks are preemptible by another higher priority thread. Therefore, if a thread is holding a simple lock and is preempted, any thread trying to acquire that lock while the owner is not running will be put to sleep. Priority promotion will be employed while a lock is being waited on.

AIX Version 4.1 simple locks are intended to protect thread-thread and thread interrupt critical sections. Thread-thread critical sections may contain access to pageable data, while thread-interrupt critical sections should not. Otherwise, the interrupt routine could be kept spinning for the whole duration of the page fault resolution (tens of milliseconds when disk I/O is required).

In order to minimize the impact due to MP locking, all interrupt-interrupt and thread-interrupt critical sections should never directly use the simple lock primitives, but should rather use the disable_lock() (to lock) and the unlock_enable() (to unlock) primitives; since in this kind of critical sections, disabling is always needed, while locking is required only on a multi-processor, these two primitives have been optimized to handle this. These two primitives will be discussed in later sections.

10.2.3 Complex Locks

AIX Version 4.1 complex locks are intended to protect thread-thread critical sections. They are preemptible, and priority promotion will be used like simple locks.

Complex locks support recursivity. By default, complex locks are not recursive. They can become recursive through the use of the lock_set_recursive() kernel service.

10.2.4 AIX Version 3 lockl Support

The locking routine in AIX Version 3, lockl(), is a sleeping, mutually exclusive lock. This type of lock is recursive and can only be used on process level. Sleep on an AIX Version 3 lock can be interrupted by signals.

This type of lock is supported in order to maintain compatibility for existing code. Newly written code should not use lockl(). This primitive can still be used for old MP-safe subsystems requiring very coarse-grained parallelization.

10.2.5 Allocation/Deallocation of Locks

In multi-processor systems, locking causes bottlenecks. Therefore, facilities are put in place to provide lock instrumentation. Due to these new facilities, lock words now have to be allocated before they can be used.

Lock instrumentation is turned on and off by the **bosboot** command, and is system wide. If lock instrumentation is turned off, then the lock word is the AIX Version 3 lockl() lock word. If instrumentation is turned on, then the lock word is a pointer that points into a kernel structure that has the real lock word and statistics on lock usage. These statistics are maintained by the lock and unlock system calls. This data can be read by the **lockstat** command.

Therefore, before you can use a simple or complex lock to secure, a data structure needs to be created/allocated and initialized. If instrumentation is turned off, the allocation and deallocation routines are no ops. The allocation routines only performs if instrumentation is turned on.

The two new kernel services that handle allocation and deallocation are lock_alloc() and lock_free().

10.2.6 Spinning on Locks

Interrupt handlers will be allowed to spin forever when trying to grab a simple lock. It makes no sense to sleep in the interrupt environment, since interrupt handlers don′t have an execution state indicating they are running or sleeping, nor are they scheduled by the dispatcher. Interrupt handlers are not allowed to sleep for any reason.

When trying to acquire a lock busy, threads will be allowed to spin:

- Forever, if running disabled.
- Until the owner of the lock is not currently executing on any processor. The rationale here is that it is going to be a relatively long time before the owner runs again and can free the lock.
- Until a threshold on number of spinning cycles has been crossed. This is meant to guarantee a more stable behavior to the system, preventing threads from spinning for an unpredictable time and keeping a processor unnecessarily busy.

Complex locks will behave like simple locks: a thread won′t be allowed to spin forever when blocked on a complex lock. If the lock is held in write mode, the behavior is the same as above. If the lock is held in read mode, the state of the owner won′t be tested, since the identity of the owner is not known; the thread will go to sleep after crossing the threshold of spinning cycles allowed.

A thread trying to acquire an AIX Version 3 lockl() won′t spin if the lock is busy, but will go immediately to sleep. No interrupt handler is allowed to acquire an AIX Version 3 lock, since it is used only at process level.

10.3 Device Drivers

The device driver model for multi-processors is different from the device driver model of a uni-processor. Device drivers, as a kernel extension, have the same multi-processor requirements as any kernel extensions. New features in AIX Version 4.1 provide compatibilty with old AIX Version 3 uni-processor device drivers. These new features allow the programmer to tell the system, when the device driver is registered, the type of device driver. There are three types of device drivers:

- **Funnelled** - Device driver only runs on the *master processor*. Serialization used in a uni-processor device driver is sufficient for this type of device driver. It is intended to support low-throughput devices and provide compatibility with AIX Version 3 device drivers.
- **MP Safe** - This type of device driver runs on any processor. The code for the device driver has been designed to serialize device driver execution via code locking. That is, a single lock is used by the program to ensure that only one instance of the program is actually running at a time. Many times, performance for this type of device driver on a multi-processor will only equal uni-processor performance. This type of device driver is intended for medium-throughput devices.

- **MP Efficient** - This type of device driver runs on any processor and has been designed to run on a multi-processor. The device driver serializes access to devices and data. This type of device driver is intended for high-throughput devices.

Another way to look at these three types is: **Funnelled** is old style uni-processor syncronization that only runs on the master processor, **MP Safe** is old style device driver with added locking to only allow the device driver to run on *any* single processor at a time, and **MP Efficient** is a redesigned device driver that is optimized to run on a multi-processor.

10.3.1 Top-Half Considerations

There are a few things that can be done to the top half of a device driver to make it run more efficiently in a multi-processor environment. However, none of them are required.

lockl() calls are fully supported in the top half of a device driver. For performance reasons, you may want to convert your lockl() calls to simple locks. The major difference between lockl() and simple locks is that simple locks do not support nesting. The recommendation is that simple locks should be used when lockl() locks are replaced, except when the work to remove nesting is too expensive. Another option is to use complex locks that support nesting.

The device switch table now has a field to indicate whether a device driver is to be funnelled or is MP Safe/MP Efficient. This is done via the devswadd() kernel service. The base kernel will assume the device driver is to be funnelled, unless the devsw structure′s **d_opts** field is set to DEV_MPSAFE prior to calling devswadd(). If you port an AIX Version 3 device driver and make no changes to the top half, it will be a funnelled device driver.

10.3.2 Bottom-Half Considerations

There are many more considerations in writing an MP-Safe or MP-Efficient bottom half of your device driver. In uni-processor device drivers, disabling interrupts was sufficient to serialize a device driver′s bottom half. Disabling interrupts is not sufficient for an MP-safe or MP-Efficient device driver. Locking must be employed in cooperation with disabling interrupts. The process of converting a uni-processor locking methodology to an MP variety can be quite complicated.

10.3.2.1 Serialization

Device drivers use i_disable() to serialize thread-interrupt critical sections on a uni-processor. This is not sufficient for a multi-processor, because i_disable() just disables interrupts for the processor that it is called from. Therefore, you need to use disable_lock(), which disables interrupts and sets a simple lock atomically.

The work in porting a uni-processor device driver to AIX Version 4.1 involves the following:

1. Replacing i_disable() and i_enable() calls with atomic lock and interrupt calls.
2. Ensuring that a thread-interrupt lock is never held across a sleep.
3. Adding serialization when a routine is called with interrupts disabled.

4. Removing lock nesting.

You may have to convert your locking methods to a data based locking method in order to make the driver MP-Efficient.

Device drivers disable interrupts to protect data structure accesses. Some drivers have carefully ordered the methods that data structures are accessed and updated, so the structures are always in a consistent state. This will not work in an MP environment.

An example would be where entries are added and removed from a singly linked list only in a process environment. The update of the list can be done in a careful fashion such that the list is always consistent. Therefore, an interrupt handler could safely scan the list on a uni-processor. This depends on process level code never running at the same time as interrupt level code. This is a correct assumption for a uni-processor. It is an incorrect assumption for a multi-processor.

10.3.2.2 Multi-processor vs. Uniprocessor Lock Behavior

The base kernel is aware whether it is running on a uni-processor or multi-processor. The kernel will not perform certain locking functions on a uni-processor to save path length. This allows an application developer to code only for the multi-processor case and be sure the extra overhead of locking will be eliminated if the code is run on a uni-processor. This will cause you, when porting your device driver, to add locking code that will only have an effect when running on a multi-processor.

disable_lock() and unlock_enable() should always be used when setting simple thread-interrupt locks. These kinds of locks are only needed in an MP implementation. By using the disable_lock() and unlock_enable() calls, the kernel knows that the lock is being used in a thread-interrupt context. Therefore, the kernel will not set and release the simple lock if running on a uni-processor. This saves path length.

The e_sleep_thread() call was created for the same reason. As discussed in the next section, you do not want to hold locks across sleeps. If you use disable_lock(), and then you need to sleep, you will want to use e_sleep_thread(). e_sleep_thread() specifies a lock word and a function parameter in addition to the event you are sleeping on. When e_sleep_thread() is called, the kernel will not operate on the lock if it is running on a uni-processor.

10.3.2.3 Thread-Interrupt Locks

Interrupts are automatically enabled when a process is put to sleep and then are automatically disabled when the process is dispatched after wakeup. This is not true for multi-processor spin locks. Therefore, both MP-Safe and MP-Efficient device drivers must be changed to make sure they never hold a thread-interrupt lock across a sleep. In general, this consists of replacing the device driver's calls to e_sleep() with calls to e_sleep_thread(). The following is a typical example where:

```
oldpri = i_disable(my_intpri);
while ( io_not_done )
     e_sleep(&my_event);
i_enable(oldpri);
```

Would be changed to:

```
oldpri = disable_lock(my_intpri, &my_lock);
while ( io_not_done )
     e_sleep_thread(&my_event, &my_lock, LOCK_HANDLER);
unlock_enable(oldpri, &my_lock);
```

The LOCK_HANDLER parameter tells e_sleep_thread() this simple lock is a thread-interrupt spin lock and not a thread-thread wait lock. This parameter must always be specified for a thread-interrupt spin lock. Additionally, MP-Safe and MP-Efficient device drivers must be very careful to release their multi-processor spin locks when they call outside routines that might sleep. This is not a concern for the bottom half routines that only run on an interrupt level. It is a concern for the routines that only run at process level or run at either process or interrupt level.

Excessive releasing of locks and reacquiring them can add path length, so care should be taken when doing this. The locks do not have to be released and reacquired when calling a base kernel service that can be called from an interrupt environment.

Note: This discussion of explicit sleep calls also applies to implicit sleeps, such as page faults. An MP-Safe or MP-Efficient device driver cannot touch data that is not pinned while holding a thread-interrupt lock.

10.3.2.4 Serializing When Called Disabled

There are several device driver bottom half routines that are called disabled by the base kernel. Therefore, these routines might not have calls to i_disable() and i_enable() in them. So, calls to disable_lock() and unlock_enable() must be added to these routines if they require serialization. The following table describes the device driver routines that are called disabled by the base kernel.

Routine	Location	Called at Interrupt Priority
EPOW Handler	intr.handler	INTEPOW
interrupt handler	intr.handler	INTCLASS0-INTCLASS3 (intr,priority)
timeout routine	trb.func	INTCLASS0-INTCLASS3 (trb.ipri)
watchdog	watchdog.func	INTTIMER
offlevel	intr.handler	INTOFFL0-INTOFFL3 (intr,priority)
iodone routine	buf.b_iodone	INTIODONE

10.3.2.5 Lock Nesting

The i_disable() service fully supports nesting. Most device drivers have nested calls to i_disable(). The disable_lock() service does not support nesting. Therefore, all nested locking must be removed from a device driver to make it either MP Safe or MP Efficient. The best way to do this is to restructure the device driver so that nesting does not occur. A good technique that has been used to successfully remove nesting is to have two names for each routine that can be called from either outside of a critical section or from inside a critical section. The first name is of a small routine that is called from outside of a critical section. This routine acquires the lock, calls the other routine, and then releases the lock. The other routine assumes the lock is already acquired.

Note: This technique can often result in shorter path lengths for both a uni-processor and a multi-processor.

Some device drivers may be keeping a variable to prevent nested calls to i_disable(), to save path length. This technique cannot be used on a multi-processor, even after the i_disable() and i_enable() calls have been changed to disable_lock() and unlock_enable() calls. The reason is that it does not prevent execution at the same time on more than one processor. The base kernel does not provide an equivalent test that works for a multi-processor.

10.3.2.6 Device Driver Interface Changes

The AIX Version 4.1 kernel is designed to support binary compatibility for funnelled device drivers. The device driver must tell the kernel when funneling is required. This is done through a set of flags that have been added to various system services.

A device driver′s **strategy routine** is defined to the base kernel via the devswadd() kernel service, along with the device driver′s top half routines. The devstrat() service will assume the device driver is a funnelled device driver, unless the devsw structure′s d_opts field is modified to specify the DEV_MPSAFE flag prior to calling devswadd(). The device driver will be treated as an MP-Safe or MP-Efficient device driver if DEV_MPSAFE is specified. Strategy routine calls for funnelled device drivers will be queued up to be run on the master processor, if the strategy routine is for a funnelled device driver and this is not the master processor. Otherwise, the strategy routine will just be called. The strategy routine may or may not have been called when devstrat() returns.

A device driver′s **iodone routine** is defined to the base kernel via the buf structure′s b_iodone field. The iodone() service will assume the device driver is a funnelled device driver, unless the dev structure′s b_flags field specifies B_MPSAFE. This flag must be set in the buf structure by an MP-Safe or MP-Efficient device driver prior to calling devstrat(). Otherwise, iodone() will only run the device driver′s b_iodone() routine on the master processor.

An **interrupt handler** is defined to the base kernel via the i_init() kernel service. This service will assume the device driver is a funnelled device driver, unless the intr structure′s flags field is modified to specify the INTR_MPSAFE flag prior to calling i_init(). The device driver will be treated as an MP-Safe or MP-Efficient device driver if INTR_MPSAFE is specified.

The INTR_MPSAFE flag should also be specified for EPOW interrupt handlers and offlevel interrupt handlers. The initial multi-processor base kernel does not require that these flags be set to work properly; setting them just allows for future base kernel design changes. Similarly, the T_MPSAFE flag should be set in the trb structure′s flags field for MP-Safe and MP-Efficient device drivers.

There is no flags field in the watchdog handler′s watchdog structure. Therefore, watchdog handlers are architected to run only on the master processor or run only on the processor that w_init() is called on.

The i_init() and i_clear() kernel services are called to add or remove an interrupt handler from the list of interrupt handlers the base kernel knows about. The base kernel can always succeed at adding or removing the specified interrupt handler on a uni-processor. On a multi-processor, it may not be able to, if interrupts are being processed on another processor. Therefore, neither i_init()

nor i_clear() can be called while holding a thread-interrupt lock. Calling either of these services while holding an thread-interrupt lock can result in deadlock.

The tstop() kernel service is called to remove a **timeout request** from the list of timeout requests the base kernel knows about. The base kernel can always succeed at removing the specified timeout request on a uni-processor. On a multi-processor, it may not be able to, if timeout requests are being processed on another processor. Therefore, a return value has been added to tstop(). The value is 0 if the timeout request is successfully removed; otherwise, it is -1. So an MP-Safe or MP-Efficient device driver must change its tstop() calls to be:

```
while(tstop(&trb)) release_then_reacquire_dd_lock;
```

The above assumes that a thread-interrupt lock was held when tstop() was called. The tstop() service may fail even if a thread-interrupt lock was not held when it was called. In this case, though, the caller would not have to release and then reacquire the lock. They would simply retry.

Note: Some delay may be required between releasing the thread-interrupt lock and reacquiring it.

The w_init() and w_clear() kernel services are called to add or remove a **watchdog timer handler** from the list of watchdog timer handlers the base kernel knows about. The base kernel can always succeed at adding or removing the specified watchdog timer handler on a uni-processor. It may not be able to on a multi-processor, if watchdog timers are being processed on another processor. Therefore, a return value has been added to w_init() and w_clear(). The value is 0 if the watchdog timer handler is successfully removed; otherwise, it is -1. So an MP-Safe or MP-Efficient device driver must change its w_init() and w_clear() calls to be:

```
while(w_init(&watchdog)) release_then_reacquire_dd_lock;
while(w_clear(&watchdog)) release_then_reacquire_dd_lock;
```

The above assumes that a thread-interrupt lock was held when w_init() or w_clear() was called. The w_init() or w_clear() services may fail even if a thread-interrupt lock was not held when either is called. In this case, though, the caller would not have to release and then reacquire the locks. They would simply retry.

Note: Some delay may be required between releasing the thread-interrupt lock and reacquiring it.

The base kernel currently sets B_DONE prior to calling the device driver's b_iodone() routine. It will continue to do that for funnelled device drivers. It will not for MP-Safe or MP-Efficient device drivers (that is, b_iodone() routines for buf structures with B_MPSAFE set in the buf structure's b_flags field). The device driver's b_iodone() routine must set B_DONE within its critical section with its thread-interrupt lock. Otherwise, the setting of B_DONE and the wakeup of anyone waiting on this I/O are not serialized correctly.

Note: This also implies that MP-Safe and MP-Efficient device drivers cannot call iowait(). They must provide their own routine that loops sleeping on the buf's b_event field while B_DONE is not set with their thread-interrupt lock held.

10.3.2.7 Considerations for Dump Devices

The dump routine for an MP-Safe or MP-Efficient device driver should be coded to assume the worst case. The dump routine should assume that one of the device driver′s critical sections was active on another processor when the dump was initiated and, therefore, the device driver′s hardware may be in an indeterminate state. The dump routine should first establish a well-defined state at the start of the first dump write.

The dump routine may call parts of the device driver or kernel that have thread-interrupt lock calls. The kernel will ensure that no deadlock occurs due to these calls.

10.3.3 Lock Calls

All type declarations for lock control words will be found in **<sys/lock_def.h>;**.

10.3.3.1 Simple Lock

simple_lock_init
: This function initializes a simple lock and, if instrumentation is enabled, clears all the statistics associated to the lock.

simple_lock
: This function locks a simple lock. A simple lock may only be used for exclusive locks, protecting *interrupt-thread* and *thread-thread* critical sections.

simple_lock_try
: This function tries to lock a simple lock without blocking if the lock is busy.

simple_unlock
: This function unlocks a simple lock.

10.3.3.2 Complex Lock

lock_init
: This function resets the internal counters associated with the lock word. The *can_sleep* parameter will have no effect; it′s supported only for OSF/1 1.1 compatibility.

lock_islocked
: This function tests whether the specified complex lock is locked.

lock_done
: This function releases a complex lock. If the lock is held recursively, decrements the recursion_depth counter; if the lock is held in read mode, the read_count is decremented; if the lock is held in write mode, the WW bit is cleared. This function will panic if the lock is actually free.

lock_read
: This function locks a complex lock for read access. It increments the read_count.

lock_read_to_write
: This function upgrades a read-only lock to one with write permission. If another reader has already requested an upgrade to a write lock, no lock is held upon return. This function panics if the lock is held in write mode.

lock_try_read
: This function tries to get a complex lock in read only mode.

lock_try_read_to_write
This function upgrades a read-only complex lock to one with write permission. If another reader has already requested an upgrade to a write lock, the read lock is still held upon return.

lock_try_write
This function tries to get a complex lock for write only access.

lock_write
This function locks a complex lock for write exclusive access. It sets the WW bit.

lock_write_to_read
This function decreases the complex lock′s owner privilege from write only to read shared access.

lock_set_recursive
This function sets the recursive flag in the lock structure; only a thread holding the lock in write will be allowed to set the recursive flag

lock_clear_recursive
This function prevents a complex lock from being acquired recursively.

10.3.4 Allocation/Deallocation Services

lock_alloc
This function allocates a secondary structure for the lock. It must be called before a lock can be used. After this function is called, the lock is ready to be initialized by the appropriate init service. This function is called for both simple and complex locks. This function is not needed for AIX Version 3 locks.

lock_free
This function frees an instance of a lock. This function must be called before the memory backing the lock can be reused for another purpose. Its counterpart is the function *lock_alloc*.

10.3.5 Other Interfaces

The following kernel services will be provided:

disable_lock
The kernel service disable_lock() disables the specified interrupt priority and then locks the specified simple lock.

unlock_enable
The kernel service unlock_enable() unlocks the specified simple lock and then enables interrupts to the specified priority level.

lock_mine
A new kernel service will be provided just for debugging purposes, to verify that the caller is actually holding the lock passed as a parameter:

10.3.6 Atomic Operations

The following new services will be implemented and exported; they will be available at user level as well.

fetch_and_add
This primitive increments a location atomically. This primitive is particularly useful when updating reference count, statistics.

fetch_and_or
This primitive sets bits in a location atomically.

compare_and_swap
This primitive takes as parameters a pointer to an atomic location, an *old_val*, and a *new_val*; it reads the current value of the atomic location: if it equals the value of *old_val*, *new_val* is stored and TRUE is returned; if it differs from the value of *old_val*, *old_val* is modified to have the current value of the atomic location and FALSE is returned. This primitive is particularly useful when issuing operations on a singly linked list.

Glossary

AIC. AIXwindows Interface Composer. An X Windows based GUI program for creating GUI programs on AIX.

applet. A little application; for example, a sample program or a tool.

BFF. Binary Format File. One file which is an installable image.

BiDi. Bi-Directional. The ablility to handle bi-directional text, such as Hebrew and Arabic, that is sometimes read left to right (for numerals) and other times are read right to left (most text).

DMA. Direct Memory Access. Generally, this term refers to the transfer of data between an adapter and system memory by a device that ″steals″ memory cycles from the CPU for the purpose of reading or writing data directly from or to memory. First-party DMA refers to DMA operations that are completely controlled by an adapter (also referred to as BUS MASTER DMA). Third-party DMA refers to DMA operations that are controlled by onboard (planar) hardware which works with the adapter to accomplish the data transfer (also referred to as SLAVE DMA).

GDLC. Generic Data Link Control. A software API to a kernel extension.

GPR. General Purpose Register. This refers to a CPU register.

I/O. Input/Output. The remote input and output system I/O space supports two classes of ″devices″ referred to as system I/O and standard I/O. System I/O is defined as facilities in the I/O space intrinsic to the system but not normally considered as I/O devices, such as NVRAM, clock/calendar and system registers. Standard I/O supports those devices that are defined intrinsic to a basic workstation, such as keyboard, mouse, diskette adapter, and ports.

IOCC. (1) Input/Output Controller Chip (2) Input/Output Channel Converter. A processor I/O bus attached space providing support for load and store, interrupt, and I/O control such as DMA to an I/O bus, (for example, the IBM Micro Channel Bus).

IPL. Initial Program Load. Refers to the process of finding, loading into system memory, and then passing machine control to a program that originally resided on some type of media such as a diskette, hardfile, CD-ROM, or local area network. This is the primary function performed by the IPL ROM code.

JPEG. Joint Photographic Experts Group. This is an image compression standard.

LLC. Logical Link Control. In LANs, the top sublayer of layer 2, IEEE 802.2.

LSA. LAN and SCSI Adapter. This is a dual-function Micro Channel architecture 40MB/s adapter. It combines a single-channel SCSI-2, fast (10MB/s), wide (8- or 16-bit), differential with an Ethernet, big coax (10BASE5), and twisted pair (10BASET).

MCA. (1) Micro Channel Adapter (2) Micro Channel Architecture

NTSC. National Television Standards Committee (USA). This is a video composite format which is interlaced, 30 frames per second, 525 lines per frame. It is used in the U.S., Japan and Korea. See also PAL.

PAD. Packet Assembler/Disassembler for X.25

PAL. Phase Alternating Line (television). This is a video composite format that is interlaced, 25 frames per second, 625 lines per frame. It is used in Europe, UK, Australia and South Africa. See also NTSC.

POR. (1) Power-on-reset. Implies the reset is caused by a cycling of the power from an inactive state to an active state. (2) Plan of Record.

RIOS. A pseudonym for the IBM RISC System/6000 workstation family.

ROM. Read Only Memory. An alternate name for ROS. In this document, it is a general reference to an EPROM which contains the IPL ROM code. IPL ROM is a generic term used to refer to the program code designed for the explicit purpose of causing an IPL.

ROS. Read Only Storage. An alternate name for ROM.

SNA. Systems Network Architecture, an IBM communications protocol.

TLB. Translation Look-aside Buffer. Hardware that contains the virtual-to-real address mapping.

UI. (1) UNIX International, a UNIX vendor organization. (2) User Interface

UU. Universal UNI, the COSE API.

WABI. An environment to execute MS Windows shrink-wrapped applications on UNIX. Wabi is available as an Licensed Program Product on AIX Version 4.1

YUV. A color image encoding scheme that separates luminance (Y) & two color signals: red minus Y (U) and blue minus Y (V)

List of Abbreviations

ABI	Application Binary Interface
AIX	Advanced Interactive eXecutive
ANSI	American National Standards Institute
APAR	Authorized Program Analysis Report
API	Application Program Interface
ASCII	American National Standard Code for Information Interchange
BDR	Bus Device Request
BOOTP	Boot Protocol
BOS	Base Operation System
BSD	Berkely Software Distribution
CD-ROM	Compact Disk Read Only Memory
CGE	Common Graphics Environment
CMVC	Configuration Management Version Control
COSE	Common Open Systems Environment
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DASD	Direct Access Storage Device
DE	Device End
DLPI	Data Link Protocol Interface
DME	Distributed Management Environment
dpi	dots per inch
EA	Effective Address
EBCDIC	Extended Binary Coded Decimal Interchange Code
EIA	Electronics Industries Association (USA)
ESCON	Enterprise Systems Connection
EUC	Extended Unix Code
FCS	Fiber Channel Standard
FDDI	Fiber Distributed Data Interface
FFDC	First Failure Data Capture
FFST	First Failure Support Technology
GA	General Availability
GB	Gigabytes
GL	Graphics Language
GUI	Graphical User Interface
HCON	3270 Host Connection program.
HFT	High Function Terminal
IETF	Internet Engineering Task Force
ILS	International Language Support
IP	Internet Protocol
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ISV	Independent Software Vendor
JFS	Journalled File System
LAN	Local Area Network
LED	Light-Emitting Diode
LFT	Low Function Terminal
LPP	Licensed Program Product
LV	Logical Volume
LVM	Logical Volume Manager
MB	Megabytes
MBCS	Multi-Byte Character Set
MCA	Micro-Channel Adapter
MHz	Mega-Hertz
MMU	Memory Management Unit
MP	Multi-processor
MPQP	Multi-Protocol Quad Port
MSFR	Minimum Security Function Requirements
NCR	Networked Computing Resource Company
NFS	Network File System
NIO	Native I/O
NIM	Network Installation Management
NIS	Network Information System
NLS	National Language Support
NVRAM	Non-Volatile Random Access Memory

OCS	On Card Sequencer
ODM	Object Data Manager
OEM	Original Equipment Manufacturer
ORI	Online Retrieval Interface
OS	Operating System
OSF	Open Software Foundation
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
PCA	Parallel Channel Attach
PCI	Peripheral Component Interconnect - an Intel bus.
PII	Program Integrated Information
PIO	Programmable Input/Output
PMR	Problem Management Report
POS	Programmable Option Select
POSIX	Portable Operating System Interface for Computer Environments
PRPQ	Programming Request for Price Quotation (IBM custom-built program products)
PTF	Program Temporary Fix
RAM	Random Access Memory
RAS	Reliability, Availability, Serviceability
RDTO	Receive Data Transfer Offset
RFC	Request for Comments
RGB	Red/green/blue
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RSC	RISC System/6000 Single Chip
SCSI	Small Computer System Interface
SDLC	Synchronous Data Link Control
SE	Single End
SLIP	Serial Link Internet Protocol
SMIT	System Management Interface Tool
SMP	Symmetric Multi-processor
SMT	Surface Mounted Technology
SNMP	Simple Network Management Protocol
SOM	System Object Model
SPOT	Shared Product Object Tree
SW	Software
SWVPD	Software Vital Product Data
TBD	To be determined/done/decided
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TOC	Table of Contents
TOD	Time of Day
UCS	Unicode Conversion Support
UP	Uniprocessor
USL	Unix System Laboratories (Tokyo, Japan)
UTF	Unified Test System
VMM	Virtual Memory Manager
VPD	Vital Product Data
VSM	Visual System Management
XPG	X/Open Portability Guide
XTI	X/Open Transport Library Interface