SunSoft
A Sun Microsystems, Inc. Business

Corporate Headquarters
2550 Garcia Avenue
Mountain View, CA 94043-1100 USA
(415) 960-3200
FAX (415) 961-6070

Sales Offices

Atlanta, GA: (404) 901-6900
Austin, TX: (512) 502-3829
Billerica, MA: (508) 442-0001
Chicago, IL: (708) 285-8700
Los Angeles, CA: (310) 348-8649
Mountain View, CA: (415) 960-3200
New York City, NY: (212) 635-2800

SunSoft Federal Systems
2100 Reston Parkway, Suite 204
Reston, VA 22091
(703) 758-0900
FAX (703) 620-1244

OpenWindows

When Solaris was ported to the PowerPC platform, the windowing code comprised the MIT X11R5 distribution (server, libraries, and sample applications) and OPEN LOOK® (XView™, ToolTalk®, and DeskSet™ and other sample applications). Altogether there were approximately 2,250,000 lines of code. The problems encountered were related to approximately 200 lines of non-standard-conforming use of varargs.

Common Desktop Environment

The Common Desktop Environment (CDE) consists of approximately 2,700,000 lines of code. Only six files had to be changed, most of which were "Makefiles." Only one C source file was changed, a one-line replacement of an assignment with `va_copy`.

Localizations

As an experiment, the German and Korean locales were ported to Solaris *PowerPC Edition* by someone with no prior exposure to PowerPC systems. The code consisted of locale definition files, message files, code conversion utilities, an input method server, and a language engine—a total of 55,000 lines of code. There were no problems with code portability.

Conclusion

The cost of porting applications developed on other platforms to Solaris *PowerPC Edition* is likely to be very low. This paper described the issues most likely to cause problems, and experience has shown that they rarely do. Most applications developed for Solaris x86 or Solaris for SPARC, or with reasonable attention paid to language standards, will port quickly and without significant change.

Porting Experience

In the not-so-distant past, programmers wrote code for a particular environment on a particular machine and nonportable code was common. Programs written largely or exclusively in assembler language were the rule, and there was no hope that such programs would be portable to computers using a different processor. The widespread use of high-level languages, along with language standardization that recognizes the need to support different processor architectures, has made it possible and even easy to write portable code. The rapid evolution of processors and the need for applications to run across a range of processors either within a family (such as x86) or across multiple families (such as SPARC and x86) has meant that most application vendors have already dealt with most portability issues—this is certainly true for Solaris itself. This section presents some indications of the portability of code from Solaris on SPARC and x86 to Solaris *PowerPC Edition*, based on experience during the early development of the Solaris *PowerPC Edition* prototype.

Drivers

The initial PowerPC platform to which Solaris was ported shared a number of peripherals with x86. x86 drivers for the direct memory access (DMA) controller, diskette, parallel port, serial ports, a mouse port, and an Ethernet card were all ported to this platform. Since drivers tend to have the most processor-specific code, the needed drivers comprised approximately 13,000 lines of code. The only problems encountered were:

- Approximately 100 lines of assembler code had to be rewritten for the PowerPC instruction set.
- There was one problem with alignment; the Ethernet driver attempted to output 4 nonaligned bytes as a `long`.

Networking

The Solaris networking code consists of libraries, commands, and kernel modules, totaling approximately 450,000 lines of code. The only problems encountered, coincidentally, were the same as for drivers, one assembler language module and one instance of an alignment problem.

Example 10 Nonportable Assignment of va_lists

```
#include <stdarg.h>
...
int func(int fixedarg, ...)
{
    va_list args, bookmark;

    va_start(args, fixedarg);
    /* Scan some arguments using va_arg(args) */
    ...
    /* "Mark" the place in the argument list */
    bookmark = args;

    /* Scan more arguments using va_arg(args) */
    ...
    /* Rescan arguments using va_arg(bookmark) */
    ...
}
```

Example 10 assumes that va_lists (that is, args and bookmark) are of a type that can be assigned (bookmark = args). However, the type of a va_list is implementation dependent and must therefore be treated as opaque and hence not necessarily assignable. Although the above code works on x86 and SPARC, it does not work on PowerPC implementations because, on PowerPC, a va_list is an array type, and C does not permit assignment of arrays.

To solve this problem, Solaris has added a macro, va_copy, to stdarg.h and varargs.h, which an application may use to make the above code portable among Solaris platforms and, by defining va_copy appropriately, on other platforms as well. Simply replace the assignment bookmark = args with va_copy(bookmark,args).

One possible ANSI-compliant alternative to the above that does not require the use of va_copy is to keep track of the number and type of each argument scanned before marking the place in the argument list, then to use a second va_start and repeated va_args to scan the arguments up to that place again before rescanning the later ones. Although this technique would be ANSI-compliant, it may not be practical; keeping track of the earlier arguments might require considerable bookkeeping and might not even be confined to a single function.

Example 9 Nonportable Assumption Regarding va_list

```
#include <stdarg.h>
int execl(char *name, ...)
{
    va_list args;
    extern char **environ;

    va_start(args, name);
    return (execve(name, (char **)args, environ));
}
```

Example 9 assumes that the va_list, properly constructed by va_start, can be cast to a pointer to (an array of) pointers to characters, which is the type of argument that execve expects. If the argument had not been cast, lint would have complained, but this improper cast results instead in an error at run time in which the wrong arguments are passed to execve. This problem was fixed by having execl scan its variable argument list as in Example 7 and insert each argument, up to and including a NULL pointer, into an array.

There is another error in the above code which bears pointing out, though it is harmless on all current Solaris implementations. The call to va_start should have been matched by a corresponding call to va_end.

The "Bookmark" Problem

Some programs have been written to "mark" a place in a variable argument list, scan some of the arguments from that point, then return to the marked place and rescan the arguments. Unfortunately, there is no ANSI-compliant way to do this. We have seen code that works on some implementations by declaring an object of type va_list, assigning the va_list created by va_start to the object, then scanning the arguments once with the original va_list and again with the copy assigned to the object, as shown in Example 10.

Example 8 Incorrect Handling of Varargs

```
#define MAXARGS 31
void f1(int n_ptrs, char *arg1)
{
    char **argp = &arg1;
    char *array[MAXARGS];
    int ptr_no = 0;

    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    while (ptr_no < n_ptrs)
        array[ptr_no++] = *argp++;
    f2(n_ptrs, array);
}
```

Example 8 assigns the address of its second argument to a pointer and, in the `while` loop, increments the pointer, assuming that it will point to successive arguments. This kind of code is hard to detect without examining code directly—that is, since assigning the address of an argument to a pointer and incrementing a pointer are each valid and useful operations, no compiler flags the potential error.

Making Assumptions About the Type or Contents of a `va_list`

Code that takes the address of an argument and passes it on to a function that expects a variable argument list (`va_list`) as an argument makes an implicit assumption that a `va_list` is simply a pointer to the first argument in an array of arguments, which is not the case for the PowerPC ABI. The `lint` program readily points out this particular error as an argument type mismatch between caller and callee, in which the callee expects a `va_list` but the caller passes a pointer to some type of argument. The proper way to construct a `va_list` argument is through proper use of `stdarg.h`, as shown in Example 7.

Some programs may use the proper method to construct a `va_list` to pass to a function that it calls, but the callee may expect some other type of argument. For example, before being ported to Solaris *PowerPC Edition*, the implementation of the `execl` function was (roughly) as shown in Example 9.

Example 7 Correct Handling of Varargs

```
#include <stdarg.h>
#define MAXARGS 31

void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_no = 0;

    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
        array[ptr_no++] = va_arg(ap, char *);
    va_end(ap);
    f2(n_ptrs, array);
}
```

Incorrect Handling of Variable Argument Lists

The following sections illustrate various ways in which some legacy code may mishandle variable argument lists.

Failure to Use `stdarg.h` *or* `varargs.h`

On x86 all arguments are passed on the stack, in order. On SPARC floating-point arguments are passed in integer registers, which can be (and sometimes are) stored blindly on the stack, resulting in the same ordered array of arguments. Thus, code such as that in Example 8, which takes the address of an argument and increments it expecting to get the address of the next argument, will work on x86, and will *sometimes* work on SPARC, but it will *not* work on PowerPC systems.

Variable Argument Lists

Background

To improve performance when floating-point arguments are passed, the PowerPC ABI specifies that the first eight floating-point arguments to a function are passed in floating-point registers. This scheme poses a problem for functions that accept “variable argument lists," that is, functions that may be called with different numbers of arguments and/or different types of arguments. In this section such functions will be called “varargs” functions.

The best-known varargs functions are `printf` and `scanf` and their variants. ANSI C requires all varargs functions to have prototypes declaring them as such in every caller—for example, by including `<stdio.h>`, which contains the declaration:

```
extern int printf(const char *, ...);
```

The ellipsis (“`...`”) syntax declares `printf` as a varargs function.

There is, however, a substantial body of “legacy” code that contains calls to these functions without prototypes. Therefore, the function calling sequence in the PowerPC ABI was designed to accommodate the legacy code by making the calling sequence for a function called with a particular set of arguments independent of whether the function is a varargs function. Since arguments are passed in both floating-point registers and integer registers (and on the stack as well when there are more arguments than can fit into the registers available for argument passing), it is up to the varargs function itself to determine how to locate successive arguments.

Correct Handling of Variable Argument Lists

The ANSI-compliant way to code varargs functions is to use the interface specified in `stdarg(5)`. Solaris also supports the alternative, but not ANSI-compliant, mechanism specified in `varargs(5)`.

Example 7 uses the recommended `stdarg` mechanism. The function `f1` stores up to `MAXARGS` of its arguments (which are pointers to `chars`) into an array, then passes the number of arguments and the resulting array to function `f2`. The number of pointers is specified by the first argument to `f1`.

Bit-Field Layout and Padding

Another implementation choice that can affect application portability is the layout and padding of bit fields. The rules dictated by the PowerPC ABI are the same as on x86; bit fields are allocated from right to left (the least significant to the most significant end of a word) and always reside in a storage units appropriate for their declared types. For example, a bit field declared `short` (for example, `short b:3;`) is not allowed to cross a `short` (that is, 16-bit) boundary. Note, however, that for maximum portability, applications should not specify bit fields with any type other than `int`, `signed int`, or `unsigned int`, as these are the only types guaranteed by the ANSI C specification.

Bit fields whose types are not explicitly declared `signed` are treated as `unsigned` in the PowerPC ABI. This is the same as on both SPARC and x86.

As is the case with alignment, problems with bit fields are very rare and generally occur only in the case of imported or exported data structures. The recommended way to handle bit-field problems when they do arise is to `#ifdef` structure definitions containing bit fields on the symbols `_BIT_FIELDS_LTOH` and `_BIT_FIELDS_HTOL` in `<sys/isa_defs.h>`. The former symbol is defined on systems with low-to-high bit-field order (such as PowerPC and x86) and the latter symbol on systems with high-to-low bit-field order (such as SPARC). Alternatively, one can eschew the use of bit fields and rely on explicit masking and shifting. This does not make good use of the language and may hamper the ability of some non-optimizing compilers to use special instructions for inserting and extracting bit fields efficiently.

Passing of Aggregates as Arguments

The PowerPC ABI specifies that a structure or union argument be passed as a pointer to the structure being passed, or to a copy of the structure if necessary to assure C's call-by-value semantics. With respect to the called function, a structure argument is indistinguishable from a pointer to a structure. This is the same technique as is used on SPARC. On x86, structure arguments are copied onto the stack. Although this was the source of a small number of nonportabilities between SPARC and x86 (nonportabilities due to argument type mismatch that the `lint` program diagnoses), it should not cause any problems in going from x86 to SPARC or PowerPC.

A Note for "Language Lawyers"

The following examples illustrate that the 0 to 255 range for `char` is not exactly the same thing as `char` being `unsigned`.

K&R C (described in the earlier-cited book by Kernighan and Ritchie) treats expressions involving any `unsigned` quantity as `unsigned`, while ANSI C extends "narrow" `unsigned` quantities to their signed counterparts if in doing so the value can be preserved.

Example 5 Nonportable Assumption Regarding `char` Range

```
char c1 = 0xFF;
int i = -1000;
if ((c1 + i) < 0) ...
```

In Example 5, `c1` is a signed quantity with value 255 on PowerPC implementations and value -1 on implementations such as SPARC and x86, on which the default `char` range is -128 to 127. With both ANSI and K&R C, the expression `(c1 + i)` is a (signed) integer expression, but on PowerPC it has value -745, and on SPARC and x86 it has value -1001.

Example 6 Difference Between ANSI and K&R C

```
unsigned char c2 = 0xFF;
int i = -1000;
if ((c2 + i) < 0) ...
```

The Sun compiler invoked *without* one of the "conform to ANSI" flags (`-Xa` or `-Xc`) produces two warning messages for the last line in Example 6: `"Semantics of '<' change in ANSI C; use explicit cast"` and `"Unsigned compare with zero always false."` The second warning arises because, under K&R C, the expression `(c2 + i)` is `unsigned`, so is never less than 0. With ANSI C semantics, the compiler produces only the first warning, a reminder that the behavior may not be what one might expect and that one should cast the expression to `int` or `unsigned int` to obtain consistent behavior with both ANSI and K&R C rules.

Example 3 Nonportable Assumption Regarding `char` Range

`char c = -1;`	produces a warning that the initializer -1 does not fit
`if (c < 0)`	warns "Unsigned compare with zero always false."

The code shown in Example 3 can easily be fixed by declaring `c` to be of type `signed char` rather than simply `char`. Fixing the code in this way will make the program behave just as it does on SPARC and x86. However, this is not always the correct way to fix the code, as the following example illustrates.

Example 4 Improper Use of `char` to Hold an `int` Value

```
#include <stdio.h>
char c;
...
if ((c = getchar()) != EOF)      produces a warning: "Result of comparison
                                 never varies."
```

Example 4 contains a bug in all implementations, though the compiler produces a warning only where the character range is 0 to 255, as in PowerPC implementations. In the example, `getchar()` returns an integer value which, if it succeeded in reading a character from the standard input, is in the range of 0 to 255, or, if it encountered an error or end of file, has the value `EOF` (the constant -1). By storing the return value in the `char c`, the program removes the distinction between the character with value 0xFF and the end of file indication, -1. The program should declare `c` as an `int` so that it can hold the value returned by `getchar()`. On SPARC or x86, a program containing the above sequence will incorrectly treat the byte 0xFF in the input as an `EOF`.

The compiler warnings produced for this case and for similar code using the `getopt(3C)` library function exposed several latent bugs in the Solaris software.

Some compilers may provide an option for treating unqualified `char` declarations as if they were `signed char`. Note, however, that code compiled with such an option may not work as expected with library functions or other code compiled without the option. Also note that assuming `signed char` is not exactly the same as changing the range of `char` (see below).

When alignment errors occur, they generally produce “bus errors.” Even though they are somewhat difficult to locate from code inspection, they are relatively easy to locate with debuggers at run time.

Integer Division by Zero

On PowerPC systems, integer division by zero does not produce an exception as it does on many other processor architectures. In practice, properly written programs take care not to divide by zero, so this is unlikely to affect application portability.

Assumptions Regarding Language Implementation Choices

Language specifications, such as the ANSI C standard[1] and the earlier book by Kernighan and Ritchie[2], leave some aspects of the binding of the language to a processor or system to be “implementation-defined.” Some of these bindings, such as those discussed earlier, are primarily determined by the processor architecture. Others are somewhat less constrained. This and the following section highlight implementation choices that may affect portability of code among Solaris implementations for PowerPC, SPARC, and x86.

`char` *Data Type*

On both SPARC and x86, the range of C `char` data declared without the `signed` or `unsigned` qualifiers is -128 to +127. In the PowerPC ABI, unqualified `char` values range from 0 to 255, a natural choice because the architecture has an instruction for loading a byte into a register and clearing the high order bits, but no instruction for loading a byte with “sign extension.” Although this difference raises some portability issues, experience in porting Solaris has shown that:

- Most of the portability problems are detected and reported by the compiler, and
- The reported problems are often latent bugs present on the other platforms.

---

1. *American National Standard for Information Systems - Programming Language C*, X3.159-1989.

2. Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc., 1978.

Alignment

The file <sys/isa_defs.h> also defines a set of symbols that specifies the alignment of various basic data types. The alignments for PowerPC systems are dictated partly by the processor architecture, which supports only aligned accesses to halfword (16-bit) and word (32-bit) data, and partly by the PowerPC Supplement to the System V Application Binary Interface (ABI), which requires that each basic data type be aligned on a multiple of its size. Thus, alignment symbols for PowerPC implementations defined in the header file are as follows:

```
#define    _CHAR_ALIGNMENT          1
#define    _SHORT_ALIGNMENT         2
#define    _INT_ALIGNMENT           4
#define    _LONG_ALIGNMENT          4
#define    _LONG_LONG_ALIGNMENT     8
#define    _DOUBLE_ALIGNMENT        8
#define    _LONG_DOUBLE_ALIGNMENT   16
#define    _POINTER_ALIGNMENT       4
#define    _MAX_ALIGNMENT           16
```

This differs from SPARC only in that `long double` data must be aligned on 16-byte boundaries on PowerPC systems and only on 8-byte boundaries on SPARC. It differs from x86 in that, on x86, `long long`, `double`, and `long double` alignments (and therefore `_MAX_ALIGNMENT`) are all 4 bytes. In addition, the size of a `long double` on x86 is 12 bytes, not 16 bytes as on SPARC and PowerPC systems.

Nonportabilities due to alignment problems are exceedingly rare, and there is no systematic, mechanical way to locate them in source files. The few cases that arose in porting Solaris were due to data structures created on and imported from x86, with its less strict alignment requirements, so one should be especially careful to examine data structure declarations for imported data. When possible, one should use XDR (eXternal Data Representation, as supported by the `xdr(3N)` functions) to make data structures portable between architectures. When historical precedents require one to deal with data structures created on processors with different alignment requirements, the only choice is to convert them to “native” form, preferably only at input and output time rather than on each use of a datum in the structure.

into a pointer to a `char`. Example 2 shows how this code can be written in a much more straightforward and portable form, using the features of the C language.

Example 2 Portable, Endian-Independent Code

```
switch(size) {
    case 1:
        *(char *)vcp = (char)(*lp);
        break;
    case 2:
        *(short *)vcp = (short)(*lp);
        break;
    case 4:
        *(long *)vcp = *lp;
        break;
}
```

Example 2 shows the best way to deal with endian issues—namely, by writing endian-independent software. When this is not possible, avoid `#ifdefs` on processor type as in the previous section. Instead of adding a branch for PowerPC systems, the code should be changed to be feature-specific (in this case, endian-specific), rather than processor-specific, using:

```
#if defined(_BIG_ENDIAN)
...
#elif defined(_LITTLE_ENDIAN)
...
#else
#error Neither _BIG_ENDIAN nor _LITTLE_ENDIAN is defined
#endif
```

The “`_ENDIAN`” symbols are defined in the Solaris header file `<sys/isa_defs.h>`, which contains Instruction Set Architecture (ISA) dependent definitions. Although PowerPC systems are capable of running in either big or little endian mode, Solaris *PowerPC Edition* runs in little endian mode, so the header file defines `_LITTLE_ENDIAN` for PowerPC.

Processor Feature Dependent Code

Endianness

Some code that appears to be processor-specific is really dependent on features that are common to many processors. By far the most common example is code that depends on the "endianness" of a processor, that is, whether a multibyte value appears in memory with its least significant byte or most significant byte at the lowest memory address, for "little endian" and "big endian" processors, respectively.

The best way to avoid endian-related problems is to write programs so that they are endian-independent. Code that uses casts or unions to "fool the compiler" into treating one data type as another is frequently a symptom of a programmer making an assumption about the processor on which he or she happens to be working, but which will be unwarranted on other processors. The most egregious endian dependence is typified by Example 1, in which the programmer wanted to assign a value (pointed to by the `long` pointer `lp`) to a variable (pointed to by the `void` pointer `vcp`) based on the size of the variable: 1 for a `char`, 2 for a `short`, and 4 for a `long`.

Example 1 Nonportable, Endian-Dependent Code

```
unsigned char *cp = (unsigned char *)lp;
switch (size) {
    case 1:
        *(char *)vcp = cp[3];
        break;
    case 2:
        *(short *)vcp = cp[2] << 8 | cp[3];
        break;
    case 4:
        *(long *)vcp = (((cp[0] << 8 | cp[1]) << 8) |
                cp[2] << 8) | cp[3];
    break;
}
```

The programmer was writing code for a big endian machine and constructed the values accordingly from the individual bytes of the `long`. The tip-off to this poor technique is contained in the first line, where the pointer to a `long` is cast

Porting such code to PowerPC systems would usually require adding another `#elif branch:`

```
#if defined(sparc)
...
#elif defined(i386)
...
#elif defined(__ppc)
...
#else
#error Unknown processor type
#endif
```

Note that the symbol, which your PowerPC compilation environment should define automatically, has two leading underscore characters to comply with ANSI specifications for reserved identifiers.

Also note the addition of the `#else` clause to cause the compiler to flag problems that will arise when the code is ported to another processor for which no alternative has been provided.

Self-Modifying Code

Programs that write instructions into memory and then execute them are a special case of instruction set dependent code. Such programs must not only generate the proper instruction sequences but must also guarantee that the processor will “see” them. On some processors with separate instruction and data caches, including some PowerPC and SPARC models, this may require the execution of special instruction sequences—for example, to flush the data cache and invalidate the instruction cache. In Solaris 2.5 and later, the function `sync_instruction_memory(`*`addr`*`,`*`len`*`)`, in the C library on all processor architectures, assures that any modified instructions between *addr* and *addr+len-1* inclusive are "visible" for execution.

only after measurement (such as profiling) has shown that they would be worthwhile and after the rest of the application has been ported and is running reliably.

An application might also use assembler language to gain access to facilities that are not available through a high-level language. For example, in writing multithreaded code, a programmer might need to implement critical sections to protect data from simultaneous writing. Facilities that require such processor-specific implementations are rare, and Solaris encapsulates them for application developers in library functions (`mutex_lock(3T)`, for example).

Instruction Set Dependent Code

Some code is inherently dependent on the processor instruction set. For example, parts of compilers, assemblers, disassemblers, and debuggers tend to be fundamentally instruction set dependent. While there are means of reducing such dependencies, like table-driven techniques, applications of this kind usually require significant porting effort.

Instruction set dependent applications often contain overtly processor dependent code. For example, one sometimes finds code such as:

```
#if defined(sparc)
...
#elif defined(i386)
...
#endif
```

The great majority of the code was written in C, which receives most of the attention here. We emphasize factors that may present a problem on PowerPC systems but that didn't cause the application to fail on either SPARC or x86.

The following sections detail several general areas in which one might encounter portability problems:

- Processor dependent code
- Processor feature dependent code
- Assumptions regarding language implementation choices
- Variable argument lists

Each section describes the problems that may arise, means of locating the problems, and ways of fixing them.

Because this paper focuses on factors that make code nonportable, the reader might form the mistaken impression that porting code is a minefield full of traps for the unwary. However, our experience indicates that applications that run on Solaris for SPARC or Solaris x86 are very likely to run on Solaris *PowerPC Edition* with little, if any, change, and that locating nonportabilities is usually quite easy. See the section "Porting Experience" on page 17 for the encouraging statistics.

Processor Dependent Code

Assembler Language

The most obvious form of processor dependent code is code written in assembler language. Obviously, code written in assembler language must be rewritten either in a high-level language or in PowerPC assembler language. Assembler language code is by its nature easy to locate; either it appears in separate source files or in distinctive compiler "in-line" directives.

The principal use of assembler language in applications is to improve performance through the use of instructions or instruction sequences that clever programmers can generate but which compilers cannot or do not. Such sequences are usually isolated, tight loops that constitute only a small part of an application. For example, in porting Solaris to PowerPC systems, some of the most frequently used library functions, such as the string and memory functions in the C library, were coded in assembler language to improve their performance. Generally, such optimizations should be made in applications

Solaris PowerPC Edition Porting Guide

Introduction

Solaris is Solaris is Solaris

Solaris™ *PowerPC™ Edition* presents the same Application Programming Interfaces (APIs) as Solaris for SPARC® and Solaris x86. There are not three Solaris operating environments, but three implementations of one environment, all derived from the same source base. Therefore, if your application is coded in a high-level language and it compiles and runs on Solaris for SPARC or Solaris x86, it will probably compile and run on Solaris *PowerPC Edition* without change, provided that it conforms with applicable language standards and does not make processor-specific or implementation-specific assumptions.

Portability

This paper describes and gives examples of various processor-specific and implementation-specific assumptions that programmers sometimes make, usually inadvertently, that lead to nonportable code. While code that has run on more than one processor or under more than one operating system is more likely to be portable than code that has run only in one environment, it may still be nonportable. Assumptions that happened to be true for the earlier environments may not be universally true.

Overview

This paper discusses the causes of nonportability that might pose a problem when porting an application to PowerPC-microprocessor-based systems, including but not limited to those that were encountered while porting several million lines of code to Solaris *PowerPC Edition*. Most of that code was Solaris itself: the operating system, commands, and libraries; all of the networking code; and the OpenWindows™ environment, including the X Window System.

Solaris™ *PowerPC™ Edition* Porting Guide

by
Steven Zucker

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 802-2575-11
Revision A, September 1995