Endianness in Solaris

Steven Zucker

SunSoft

901 San Antonio Road, Palo Alto, California 94303-4900, U.S.A.

Please
Recycle

Endianness in Solaris

> *"It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; ... . It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the Big-Endians have been long forbidden, ..."*
>
> —Jonathan Swift's *Gulliver's Travels*

Introduction

This paper focuses on the programming-related aspects of endianness. Its primary goal is to present mechanisms for producing endian-independent source code that can be compiled for either big endian or little endian environments from one source.

Readers unfamiliar with endianness as applied to programming (as opposed to eggs) will find some needed background in the next section. Then the *Overview* section outlines the remainder of the paper.

What is Endianness?

Nearly all computers today address their memory in units of eight bit "bytes" . They address larger storage units, typically but not always called halfwords (sixteen bits), words (thirty-two bits), and doublewords (sixty-four bits), by giving the address of the byte at one end or the other of the storage unit.

Unfortunately, some computers, the “little endians”, use the address of the numerically least significant byte, while others, the “big endians”, use the address of the numerically most significant byte for the address of the multibyte storage unit.[1] The Intel x86 family and Digital Equipment Corporation architectures (PDP-11, VAX, Alpha) are representative little endians, while the Sun SPARC, IBM 360/370, and Motorola 68000 and 88000 architectures are big endians. Still other architectures (PowerPC, MIPS, and Intel’s IA-64[2]) are capable of operating in either big endian or little endian mode.

Consider a little endian and a big endian computer, each containing in its first sixteen bytes of memory the number representing the address of the byte. Figure 1 illustrates this situation. Regarded as an array of bytes, the situation is the same on both computers.

Figure 1 Memory as an array of bytes

Byte Addr	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**11**	**12**	**13**	**14**	**15**
Contents	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15

If instead of considering the memory as an array of bytes we consider the same memory contents as four thirty-two bit *words*, the view that the two processors have is different, as shown in Figure 2 and Figure 3. In these figures, the bytes are grouped into four byte words, which are shown in the normal arabic form with the most significant byte (“digit”) on the left. In Figure 2 we put the word address column on the right (“little end”) because the computer uses the address of the least significant byte, the byte on the right, to address the word; in Figure 3, the address column is on the left (“big end”), showing that the computer addresses the most significant byte in word operations. As a result,

1. The opening quotation was written by Swift in 1726, so the “endian” term predates the computer era by more than two centuries.

2. IA-64 is Intel’s 64-bit successor to the x86 (now called IA-32) architecture. “Merced” is the better known name of the first implementation of the IA-64 architecture.

the little endian processor loading the thirty-two bit word at word address 0 would obtain a different value (03.02.01.00) than the big endian processor (00.01.02.03).[1],

Figure 2 Little endian memory as an array of words

Contents				Word Addr
03	02	01	00	**00**
07	06	05	04	**04**
11	10	09	08	**08**
15	14	13	12	**12**

.

Figure 3 Big endian memory as an array of words

Word Addr	Contents			
00	00	01	02	03
04	04	05	06	07
08	08	09	10	11
12	12	13	14	15

Overview

There are two fundamental sources of endian-related problems:

1. Programmer assumptions about the endianness of the processor on which a program will run, and
2. Endianness collision: a program running on a machine of one endianness having to deal with structured data (i.e., not simply an array of bytes) exported from an environment of the other endianness.

Most programmers will never encounter endianness problems. Some will never even write programs that will have to run on machines of different endianness. ~~This paper is intended to help t~~those who may have to deal with endianness

1. The periods denote concatenation of the byte values so that 00.01.02.03 represents $00 \times 2^{24} + 01 \times 2^{16} + 02 \times 2^{8} + 03 \times 2^{0} = 66051$

issues either in developing a new product or porting an existing product. It offers advice on how to avoid the problems where possible and how to recognize and overcome the more difficult ones. Although most of the material is not operating system specific, it highlights the solutions that Solaris offers. It also points out areas in which hardware assistance may be brought to bear in dealing with endianness problems. The paper is organized into sections as follows:

Avoiding Endianness Assumptions. The most common, and yet easiest to solve, endian-related problems arise from programmers being unaware of endianness issues. The next section describes some typical cases and shows how to recognize them and to write endian-independent code. Both applications and systems programmers may find this section useful.

Application Problems. This section deals with endianness problems encountered in applications programming.

Device Drivers and Endianness. This section covers endianness issues in systems programming such as those facing device driver writers in mixed-endian environments.

“Practical Notes” appearing throughout this paper highlight the practical application of the techniques and mechanisms discussed.

Finally, the *Appendix* discusses issues involved in choosing the endianness for an operating system on a bi-endian system, one whose hardware can support either endianness.

Avoiding Endianness Assumptions

Figure 4 shows a program that makes endian-dependent assumptions. 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 byte for a char, 2 for a short, and 4 for an int.

Figure 4 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:
            *(int *)vcp = ((cp[0] << 8 | cp[1]) << 8 |
                  cp[2]) << 8 | cp[3];
            break;
}
```

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 the code will run. In the above case, the programmer assumed that the code would run on 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 to a pointer to a char, suggesting that the programmer is going to treat the pointed-to long as an array of bytes. As the earlier figures make clear, the values obtained for the bytes at various offsets from that pointer will depend on the endianness of the processor.

Practical Note – Run lint or other programming aids that point out dubious uses of C constructs. Look for questionable casts, especially from pointers to multibyte integers into pointers to char. Any dereferencing of the resulting char pointer is highly suspect.

Practical Note – Code that is not explicitly intended to deal with endian dependence should never look at the constituent parts of numerical data.

C unions are well suited to defining variants of data structures. However, the use of unions to "alias" the members of data structures so that the *same* data can be accessed in multiple ways is another indication of nonportable code in general, and of potential endian dependence in particular.

Practical Note – Examine all union declarations. Suspect endian dependence if their use is for aliasing.

Figure 5 shows how this code can be written in a much more straightforward and portable form, using rather than abusing the features of the C language. The casts on the right sides of the assignment statements in Figure 5, while not strictly necessary, tell the compiler that the programmer knows that the target of the assignment is "narrower" than the value that the long might contain and is explicitly truncating the value.

Figure 5 Portable, endian-independent code

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

The order of bit-fields within a word is not specified in the C language, but in practice compiler writers invariably choose to assign them from low-order to high-order bits on little endian implementations and from high-order to low-order bits on big endian implementations. Looking at the bit-fields within a word (e.g., by using a union to access the word containing the bit-fields) is subject to the same portability problems as discussed above.

Practical Note – Beware of code containing bit-fields, especially when a structure containing bit-fields appears in a union with other data types.

Application Problems

Imported Data

This section describes several issues that require endian awareness on the part of applications programmers. All of them arise from the need to “import” structured data from a source which may not have the same endianness as the target machine on which the code will run. To be portable between machines of different endianness, an application must deal with these issues explicitly. The *Software Solutions* given in the next section may always be used. However, the *Hardware Solutions* in the last section generally provide better performance.

Structured Files

The most common and obvious form of data import is in the form of files. A spreadsheet or word processing program written for an Intel x86 environment without consideration for endianness issues might save user files in a form that mirrors its internal representation. That is, the data in the files might be direct copies of the internal, endian-dependent data (little endian in this case). Reading or writing these files on a big endian processor must take into account the endianness of the data.

Practical Note – Examine all input and output file formats for endian dependence.

Canonical Forms

Some applications must deal with data that has a prescribed form, regardless of the endianness of the machine on which the application runs. For example:

1. There are standards for sixteen-bit audio data and various image file formats that contain multibyte numerical data. Applications that deal with such data must take their standard-defined endianness into account.
2. Networking protocols, such as Remote Procedure Call (RPC) protocols, define standard data formats which must be used by clients of either endianness.
3. Some applications write data to or read data from devices as byte streams that are not interpreted and translated by device drivers. For example, an application program that reads a Universal Serial Bus descriptor must be prepared to translate multibyte numerical data in the descriptor from its (probably little endian) form to one usable by the program. Such translations can be done only by the consumer of the data because only the consumer knows which data in the “stream” is multibyte numerical data, which must be translated, and which is (e.g.) character strings or other byte stream data, which must not be translated.

Practical Note – Import and export of endian-dependent data is not always through well-defined *files*. Examine all external communication paths for endian dependence.

Shared Memory

A less obvious example of “imported data” is memory shared between processors. To avoid transferring data to high speed devices through an operating system, with the attendant system call overhead, one sometimes arranges for memory on a peripheral processor to be mapped directly into an application’s address space. The most common example of shared memory mapped into an application’s address space is the case of a graphics adapter with a frame buffer shared between the host and the graphics processor.

Practical Note – Sharing memory with opposite-endian devices or processors constitutes data import and export. Access to shared data must take endianness issues into account.

Some peripheral busses lend themselves to shared memory approaches to interprocessor communication. In particular, the VME bus is often used in shared memory configurations in embedded systems to attach special purpose peripherals, such as signal processors or high speed page printers, to a general purpose host computer. If the processors are not of the same endianness, some means of translating the data must be provided.

Software solutions

Structured Files

One simple solution to the imported file problem is to provide an application-dependent file format converter. This is not an unreasonable method for “one-way” conversions, in which files are imported once and never exported back to the environment from which they came. In this case, one essentially provides two “canonical forms” for application files, and translates between them. After translation, which may be done in a front end conversion program, the application sees the data only in its native endianness.

Practical Note – File format converters provide an easy way to avoid endian issues within applications by isolating the conversion problem to the explicit import and export portions of the application.

Some applications support two file formats, a portable, endian-independent, format for data interchange and a nonportable, endian-dependent format. For example, FrameMaker provides a portable “Maker Interchange Format” (“.mif” files) for cross-platform transfers, but normally uses “.doc” files, which are one-third to one-half as large and do not incur translation overhead.

Canonical Forms

Maker Interchange Format, although limited to FrameMaker, is one example of a canonical form of data. It represents the data in the form of an ASCII character stream, the *lingua franca* of at least the English-speaking part of the computer world. It has the advantage of being human readable (and potentially editable), and is, of course, an endian-independent representation. However, the endian-independence comes at the expense of large size and conversion cost.

When one recognizes endian-independence as an "up-front" requirement, one can do considerably better than the file conversion or multiple format approach. For a new application, one can adopt an efficient and compact, and at the same time endian-independent, external representation for data. SunSoft provides and recommends the use of XDR (eXternal Data Representation), which it uses for its Remote Procedure Call (RPC) protocol, for this purpose. See xdr_create(3N) in the Solaris documentation for details. XDR provides much more than endian-independence (e.g., it provides canonical floating point and string representations), so it may be overkill for applications that only need to deal with endianness issues.

Practical Note – XDR provides a canonical form for data interchange that is well-suited to reading and writing files. Consider using it when file formats are not already specified or as an alternative interchange format.

Some canonical forms, such as audio data in the form of arrays of sixteen-bit integers, lend themselves to simple, direct translation. For example, Sun provides the functions htonl(3N) and htons(3N) for translating from native ("host") to network longs and shorts (thirty-two bit and sixteen bit integers), and ntohl(3N), and ntohs(3N) for the reverse translations.

Since the "network" representation (Internet Protocol) is big endian, the host-to-network family of functions are implemented as identity macros on big endian processors. Macros or in-line functions generally provide better performance than function calls for simple endian inversion.

Practical Note – When endianness of integer data is the only data interchange issue, use the “network” form as a canonical representation. Solaris’s htonl(3N) and related functions provide efficient and easy to use endianness conversion.

Endianness conversion may be applied either when data is used or when it is imported or exported. Converting at import and export time is possible and sometimes preferable when the data is “context free”, as in the case of audio data. It also has the advantage that endianness issues are only a consideration for the import and export portions of the program; the remainder of the application does not need to concern itself with endianness issues.

Practical Note – An issue that sometimes arises along with endianness is that of *alignment*. Some processors can access multibyte data only at addresses that are a multiple of the data length. The obvious approach of doing endian translation “in-place” may not be usable when there are alignment restrictions. Furthermore, alignment restrictions may prevent direct use of functions such as htonl(3N), since picking up a “long” argument from an unaligned byte stream involves more than simply dereferencing a pointer to a long.

Shared Memory

Shared memory situations such as memory-mapped graphics adapters or VME devices are best handled with hardware solutions discussed below. However, when hardware solutions are not available, the translation function or macro approach discussed above can be applied. That is, the multibyte values can be explicitly byte-swapped before being stored to, or after being loaded from, the opposite-endian device.

Hardware Solutions to the Shared Memory Problem

Device Apertures

The computer industry trend away from proprietary peripheral busses and towards commodity busses has brought with it an increase in the use of devices on diverse platforms. For example, the PCI bus, originally designed for the little endian Intel architecture, is used on both big endian and little endian platforms: SPARC (in some recent Sun systems) and PowerPC-in-big-endian-mode on the one hand and Intel and PowerPC-in-little-endian-mode on the other.

To make it easier and more efficient to use their memory-mapped devices on platforms of either endianness, some peripheral vendors provide “apertures” for each endianness. For example, a PCI-based graphics adapter that internally uses a little endian processor can provide two apertures (ranges of addresses) for its frame buffer. Accesses to the little endian aperture store the data as presented on the bus directly into the frame buffer; accesses to the big endian aperture swap the data bytes before storing them. Thus, an application running on a big endian processor can simply map the big endian aperture and store its big endian data just as if it were running on a little endian processor. The device takes care of swapping the data as necessary.

Host-based Mapping Hardware

Just as a device can provide endian translation apertures, so can a host processor. UltraSPARC processors have several mechanisms for dealing with endianness, one of which allows independent assignment of endianness to each mapped page. For example, if an audio file containing little endian data were mapped into memory in little endian mode on an UltraSPARC, the data would be transparently swapped as it was loaded from or stored to the memory into which the file was mapped. This mechanism also solves many of the problems associated with shared memory, so long as data is properly aligned. However, it does not eliminate the need for applications to be endian-aware. If the same memory contains both structured data (i.e., data that represents numbers whose

values must be preserved) and byte-stream data (whose byte order must be preserved), then applications must map the same memory twice and make the accesses to data accordingly.

Device Drivers and Endianness

The Issue

Sun's use of the PCI bus in some of its systems made it desirable for Sun to produce drivers for PCI devices for both SPARC and Intel x86 platforms from common sources. Achieving platform independence required addressing the problem of endianness, both in accesses to device registers and in the construction and reading of device data in memory, such as SCSI command and status packets. Doing so in a way that takes advantage of hardware assistance when it is available (e.g., on UltraSPARC), but also works when only software swapping is available, added to the challenge.

The Solaris Solution

The solution, introduced in Solaris 2.5, involves abstracting device register access and reading and writing of device data in memory.

Device Register Access

When mapping the registers of a device (using ddi_regs_map_setup(9F)), the driver writer provides as one of the arguments a device access attributes structure (ddi_device_acc_attr(9S)), which specifies the characteristics of the data that the device expects. The ddi_regs_map_setup function returns an "access handle" (ddi_acc_handle_t) for use in subsequent data access function calls. Solaris provides a full range of functions for accessing 8-, 16-, 32-, and 64-bit data, individually or in blocks. The access functions use the handle to transform the data as required. The driver does not need to know whether or not byte swapping is required, and if it is, whether it is accomplished by hardware or by software.

There are special functions for access to PCI configuration space. These functions know that the configuration space is little endian, and also know the platform-specific means of addressing configuration space.

Memory Data

A device driver writer allocates memory for DMA to and from devices using the ddi_dma_mem_alloc(9F) function. This function, like ddi_regs_map_setup, also takes a device access attributes structure as an argument and return a handle for use in subsequent data accesses.

Practical Note – Use the Solaris DDI when writing or porting device drivers to enable cross-platform, cross-endianness portability.

Appendix: Operating Systems and Bi-endian Architectures

There is little or no reason to prefer one endianness over the other and it is truly unfortunate for program portability that both are used. However, some recent computer architectures support *both* big endian and little endian modes. Why?

The reason for a processor architecture to support both big and little endian operation invariably derives from a requirement to support legacy environments—operating systems and applications—of both endiannesses. MIPS was originally a big endian architecture; MIPS added little endian support to induce DEC, with a little endian legacy, to adopt MIPS processors for its desktop systems. IBM had a big endian legacy on its workstation and server systems and a little endian legacy on its Intel-based personal computers and wanted to support both with the PowerPC architecture. The IA-64 architecture resulted from a collaboration between Intel, with a little endian legacy, and Hewlett Packard with a big endian legacy on its workstations, so it, too, supports both.

While a *processor architecture* may support bi-endian operation, it would be very difficult to make a case for a an *operating system* to support both big and little endian applications on a given architecture, whether with separate versions or within one version. The development, packaging, distribution, and support costs involved in providing separate versions of the operating system or applications would have no compensating benefits for anyone: the OS vendor, independent software vendors, or the end user. If one version of the operating system supported applications of either endianness, applications of different endianness would not be able to interoperate. Data (e.g., a file or UNIX pipe) written by one application would not be readable by another of opposite endianness, and applications of opposite endianness would not be able to share memory. This would be problematic for any operating system, but especially so for UNIX operating systems. UNIX applications are often composed of modular components interacting through shared data, in contrast with the self-contained, monolithic applications found on other operating systems.

Thus, when considering a port of Solaris to a new architecture, Sun must choose whether the port should be big endian or little endian. Sun supports both big endian (on SPARC) and little endian (on x86) environments, and does so from a code base that is almost entirely common. With attention to the guidelines provided in this paper, Solaris applications, and even system level code (such as device drivers) can be written portably, so that they can be compiled for either environment without change. Thus, the choice of endianness for a Solaris port to a bi-endian processor rests primarily not on technical considerations or ease of porting the OS itself, but on the software environment desired by the potential customers for Solaris on the new architecture, which depends on the legacy applications that must be supported.