Endianness White Paper

November 15, 2004

Abstract

This paper describes software considerations related to microprocessor Endian architecture and discusses guidelines for developing Endian-neutral code.

Disclaimers

Legal Notices

Contents

Introduction

Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system. Unfortunately not all computer systems are designed with the same Endian-architecture. The difference in Endian-architecture is an issue when software or data is shared between computer systems. An analysis of the computer system and its interfaces will determine the requirements of the Endian implementation of the software.

Software is sometimes designed with one specific Endian-architecture in mind, limiting the portability of the code to other processor architectures. This type of implementation is considered to be Endian-specific. However, Endian-neutral software can be developed, allowing the code to be ported easily between processors of different Endian-architectures, and without rewriting any code. Endian-neutral software is developed by identifying system memory and external data interfaces, and using Endian-neutral coding practices to implement the interfaces.

Platform migration requires consideration of the Endian-architecture of the current and target platforms, as well as the Endian-architecture of the code. Best known methods describe the software interface information that should be considered and how to convert Endian-specific code to Endian-neutral code.

This white paper establishes a set of fundamental guidelines for software developers who wish to develop Endian-neutral code or convert Endian-specific code through the use of some or all of the coding techniques documented in this paper.

Note: The examples in this paper are based on 32-bit processor architecture.

Analysis

There are two main areas where Endianness must be considered. One area pertains to code portability. The second area pertains to sharing data between platforms.

Code Portability

It is not uncommon for software to be designed and implemented for the Endian-architecture of a specific processor platform, without allowing for ease of portability to other platforms.

Endian-neutral code provides flexibility for software implementations to be compiled for and operate seamlessly on processors of different Endian-architectures.

Shared Data

Computer systems are made up of multiple components, including computers, interfaces, data storage and shared memory. Any time file data or memory is shared between computers, there is a potential for an Endian-architecture conflict. Data can be stored in ways that are not tied to endian-architecture and also in ways that define the Endianness of the data.

Best Known Methods

Guidelines and Best Known Methods (BKM's) have been established to cope with Endianness-architecture differences. Following the guidelines and BKM's described in this document will greatly increase the usability and portability of embedded software between hardware platforms.

Definition of Endianness

Endianness is the format to how multi-byte data is stored in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory. Endianness is dictated by the CPU architecture implementation of the system. The operating system does not dictate the endian model implemented, but rather the endian model of the CPU architecture dictates how the operating system is implemented.

Representing these two storage formats are two types of Endianness-architecture, Big-Endian[1] and Little-Endian. There are benefits to both of these endian architectures. See section "Merits of Endian Architectures." Big-Endian stores the MSB at the lowest memory address. Little-Endian stores the LSB at the lowest memory address. The lowest memory address of multi-byte data is considered the starting address of the data. In Figure 1, the 32-bit hex value 0x12345678 is stored in memory as follows for each Endian-architecture. The lowest memory address is represented in the leftmost position, Byte 00.

Endian Order	**Byte 00**	**Byte 01**	**Byte 02**	**Byte 03**
Big Endian	12	34	56	78 (LSB)
Little Endian	78 (LSB)	56	34	12

Figure 1- Example of Memory Addressing for Big and Little Endian

As you can see in Figure 1, the value of the stored multi-byte data field is the same for both types of Endianness as long as the data is referenced in its native data type, in this case a long value. If this data field is referenced as individual bytes, the Endianness of the data must be known. An unexpected difference in Endianness will cause a computer system to interpret the data in the opposite direction, resulting in the wrong value. The difference can be correctly handled by implementing code that is aware of the Endian-architecture of the computer system as well as the Endianness of the stored data. See section "Byte Swapping."

Merits of Endian Architectures

You may see a lot of discussion about the relative merits of the two formats, mostly based on the relative merits of the PC. Both formats have their advantages and disadvantages.

In "Little-Endian" form, assembly language instructions for picking up a 1, 2, 4, or longer byte number proceed in exactly the same way for all formats: first pick up the lowest order byte at offset 0. Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write.

In "Big-Endian" form, by having the high-order byte come first, you can always test whether the number is positive or negative by looking at the byte at offset zero. You don't have to know how long the number is, nor do you have to skip over any bytes to find the byte containing the sign information. The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

Most embedded communication processors and custom solutions associated with the data plane are Big-Endian (i.e. PowerPC, SPARC, etc.). Because of this, legacy code on these processors is often written specifically for network byte order (Big-Endian).

Table 1 lists several popular computer systems and their Endian-architectures. Note that some CPUs can be either big or little endian (Bi-Endian) by setting a processor register to the desired endian-architecture.

[1] The terms big-Endian and little-Endian are derived from the Lilliputians of Gulliver's Travels, whose major political issue was whether soft-boiled eggs should be opened on the big side or the little side. Likewise, the big/little-Endian computer debate has much more to do with political issues than technological merits.

Platform	Endian Architecture
ARM*	Bi-Endian
DEC Alpha*	Little-Endian
HP PA-RISC 8000*	Bi-Endian
IBM PowerPC*	Bi-Endian
Intel® 80x86	Little-Endian
Intel® IXP network processors	Bi-Endian
Intel® Itanium® processor family	Bi-Endian
Java Virtual Machine*	Big-Endian
MIPS*	Bi-Endian
Motorola 68k*	Big-Endian
Sun SPARC*	Big-Endian

Table 1 – Computer System Endianness

Relevance of Endian Order

Endian order means that any time a computer accesses a stream (a network tap, a local file, or an audio, video or multimedia stream), the computer has to know how the file is constructed. For example: if you write out a graphics file (such as a .BMP file, which is Little-Endian format) on a Big-Endian machine, you must first reverse the byte order of each integer you write or another "standard" program will not be able to read the file.

Table 2 contains examples of some common file formats and the Endian order that they use:

Little-Endian Format	Big-Endian Format	Variable or Bi-Endian Format
BMP (Windows* & OS/2)	**PSD** (Adobe Photoshop*)	**DXF** (AutoCAD*)
GIF	**IMG** (GEM Raster*)	**PS** (Postscript*, 8 bit interpreted text, no Endian issue)
FLI (Autodesk Animator*)	**JPEG, JPG**	**POV** (Persistence of Visionraytracer*)
PCX (PC Paintbrush*)	**MacPaint**	**RIFF** (WAV & AVI*)
QTM (MAC Quicktime*)	**SGI** (Silicon Graphics*)	**TIFF**
RTF (Rich Text Format)	**Sun Raster**	**XWD** (X Window Dump*)
	WPG (WordPerfect*)	
Bus Protocols	**Network Protocols**	**Bus Protocols**
Infiniband	**TCP/IP**	**GMII** (8 bit wide bus, no Endian issue)
PCI Express	**UDP**	
PCI-32/PCI-64		
USB		

Table 2- Common file formats

How can the opposing Endian data be efficiently processed? A hardware solution doesn't allow for variability in data since it expects either Big-Endian or Little-Endian formats. Also, "Hard-wired" Endian swapping typically won’t suffice for a large range of networks and protocols and many of these file formats are fixed Endian. Software byte swapping seems the only viable method. Several different methods are available:

Byte Swapping

Basically, anytime multi-byte data is imported or exported between computer systems, the format of the data must be standardized. If the data format is binary, the Endianness of the data must be known by both nodes. With this knowledge, the computer systems can decide, based on their own endian-architecture, whether byte swapping must be performed on the data. Byte swap methods are developed to standardize the access to the data. The byte swap methods of Endian-neutral code use byte swap controls to determine whether a byte swap must be performed.

Byte Swapping Methods

There are several methods available for byte swapping. These methods perform the actual byte swapping of the given data.

- Byte swapping macros provided by an operating system’s networking libraries, (**ntohl** and **ntohs**, short for network-to-host long and network-to-host short)
- Optimized custom byte swap macros
- Inline “bswap” macros
- Assembly language instructions such as ror (**r**otate **o**perand **r**ight) or rol (**r**otate **o**perand **l**eft)
- Standard C library function “swab” can be used to swap two adjacent bytes
- A generic assembly language function implementing the same algorithm as the ntohl and ntohs macros

Network I/O Macros

All communication protocols must define the Endianness of the protocol so that there is a predefined agreement on how nodes at opposite ends know how to communicate. In the TCP/IP stack, each network host is identified by its 32-bit IP address, which is ordinarily displayed in the 4 numeric octets referred to as “network byte order.” TCP/IP defines the network byte order as Big-Endian and the IP Header of a TCP/IP packet contains several multi-byte fields. Computers having Little-Endian architecture must reorder the bytes in the TCP/IP header information into Big-Endian format before transmitting the data and likewise, need to reorder the TCP/IP information received into Little-Endian format.

Computers having Big-Endian architecture need to do nothing since their endian-architecture is the same as the TCP/IP protocol.

Network I/O Macros are standardized popular macros commonly available in network libraries and are commonly used to import / export TCP/IP packet header data, which is described below, in an Endian-neutral manner.

• Length	2 bytes
• ID	2 bytes
• Offset	2 bytes
• Source	4 bytes
• Destination	4 bytes

The network I/O macros are described in Table 3. The word “host” is used to refer to the processor’s endian-architecture and the word “network” is used to refer to the TCP/IP endian-architecture. Using these macros allow the same code to work on a Big-Endian or Little-Endian processor.

Macro Name	Translation (Can be read as…)	Meaning
htons()	"host to network short"	Converts the unsigned short integer *hostshort* from host byte order to network byte order.
htonl()	"host to network long"	Converts the unsigned integer *hostlong* from host byte order to network byte order.
ntohs()	"network to host short"	Converts the unsigned short integer *netshort* from network byte order to host byte order.
ntohl()	"network to host long"	Converts the unsigned integer *netlong* from network byte order to host byte order.

Table 3 – Network I/O Macros

The byte swap performed for TCP/IP communication on Little-Endian processors adds a performance overhead. However, this overhead can be recovered as the processor speed increases. See Recovering Byte Swap Overhead.

Custom Byte Swap Macros

Custom byte swap macros are used to wrap and standardize the code for accessing each data type. Table 4 shows and example of byte swap macros for each data size.

Access Size	Example Macro Name	Macro Code

16 bits	SwapTwoBytes	`#include <stdio.h>`
		
		`#define SwapTwoBytes(data) \`
		`( (((data) >> 8) & 0x00FF) | (((data) << 8) & 0xFF00) )`

32 bits	SwapFourBytes	`#include <stdio.h>`
		
		`#define SwapFourBytes(data)   \`
		`( (((data) >> 24) & 0x000000FF) | (((data) >>  8) & 0x0000FF00) | \`
		`  (((data) <<  8) & 0x00FF0000) | (((data) << 24) & 0xFF000000) )`

64 bits	SwapEightBytes	`#include <stdio.h>`
		
		`#define SwapEightBytes(data)   \`
		`( (((data) >> 56) & 0x00000000000000FF) | (((data) >> 40) &`
		`0x000000000000FF00) | \`
		`  (((data) >> 24) & 0x0000000000FF0000) | (((data) >>  8) &`
		`0x00000000FF000000) | \`
		`  (((data) <<  8) & 0x000000FF00000000) | (((data) << 24) &`
		`0x0000FF0000000000) | \`
		`  (((data) << 40) & 0x00FF000000000000) | (((data) << 56) &`
		`0xFF00000000000000) )`

Table 4 – Custom Byte Swap Macros

Byte Swap Controls

Byte swap controls are used within byte swap methods to determine when byte swapping should be performed. In normal usage, the controls add byte swap code if byte swapping is required. If byte swapping is not required the control adds no code and thus, does nothing. Byte swapping can be controlled with the following mechanisms:

- Compile-time controls
- Using run-time controls

Compile Time Controls

Table 5 is an example of how the compiler preprocessor is used within data access wrappers to control whether or not byte swapping should be performed. Note that different code is compiled in based on the preprocessor definition. This example is defined by the compiler to work for both Little-Endian and Big-Endian processors.

Access Size	Example Macro Names	Macro Code

16 bit Big-Endian data	MY_RD_BE_SHORT
	MY_WRT_BE_SHORT
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`/* Do nothing */`
		`#else`
		`SwapTwoBytes` `(data)`
		`#endif`

16 bit Little-Endian data	MY_RD_LE_SHORT
	MY_WRT_LE_SHORT
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`SwapTwoBytes` `(data)`
		`#else`
		`/* Do nothing */`
		`#endif`

32 bits Big-Endian data	MY_RD_BE_LONG
	MY_WRT_BE_LONG
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`/* Do nothing */`
		`#else`
		`SwapFourBytes` `(data)`
		`#endif`

32 bits Little-Endian data	MY_RD_LE_LONG
	MY_WRT_LE_LONG
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`SwapFourBytes` `(data)`
		`#else`
		`/* Do nothing */`
		`#endif`

64 bits Big-Endian data	MY_RD_BE_DOUBLE
	MY_WRT_BE_DOUBLE
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`/* Do nothing */`
		`#else`
		`SwapEightBytes` `(data)`
		`#endif`

64 bits Little-Endian data	MY_RD_LE_DOUBLE
	MY_WRT_LE_DOUBLE
		`#if CPU_ARCHITECTURE == BIG_ENDIAN`
		`SwapEightBytes` `(data)`
		`#else`
		`/* Do nothing */`
		`#endif`

Table 5 – Preprocessor Control

Run Time Controls

It is possible to detect the endian-architecture of a processor using run-time code. Figure 2 shows an example of code that does a run-time test to check (at run time) if the code is running on a little or big-Endian system. This allows run-time code to dynamically perform Endianness processing.

```
union
{
    char Array[4];
    long Chars;
} TestUnion;

char c = 'a';

/* Test platform Endianness */
for(x = 0; x < 4; x++)
    TestUnion.Array[x] = c++;

if (TestUnion.Chars == 0x61626364
    /* It’s big endian */
```

Figure 2 – Run-Time Byte Order Test

Recovering Byte Swap Overhead

Overhead associated with byte swapping the fields in the IP Header are miniscule compared to the actual propagation delay associated with the network transmission speeds available today. The byte-swapping overhead, though it undeniably exists, can be readily recovered when there is a significant amount of packet processing to be done, especially with the higher frequency processors. Figure 3 depicts a time exaggerated example of the overhead required for byte swapping, as well as the reduced processing time of more powerful processors that recover the time.

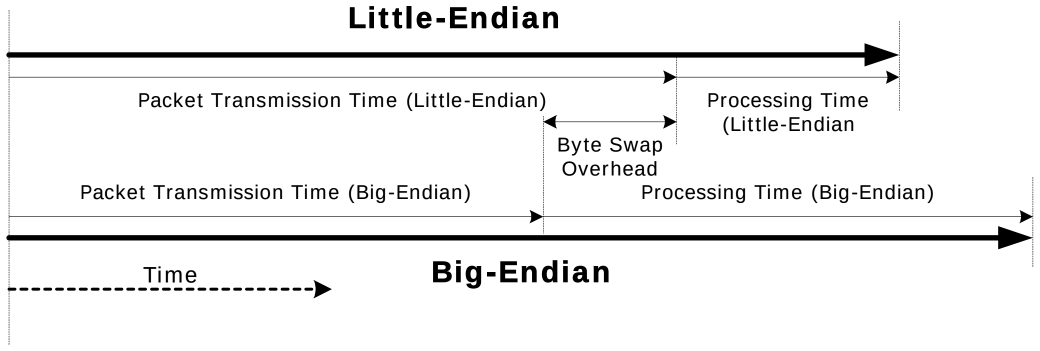
Little-Endian
Packet Transmission Time (Little-Endian)
Processing Time (Little-Endian
Byte Swap Overhead
Packet Transmission Time (Big-Endian)
Processing Time (Big-Endian)
Time
Big-Endian

Figure 3 - Example of Byte Swap Overhead and Recovery

From this example we can see that there is some overhead associated with swapping the bytes in the network headers. However, given a substantial increase in processor performance, the byte swap required on the Little-Endian processor is recovered.

Platform Porting Considerations

If the target application is currently running on a Big Endian platform and the goal is to port to a Little Endian platform, or visa-versa, byte ordering may become an issue. For the most part, the byte ordering within a system is self contained and therefore not affected by Endianness. However, a few cases can result in porting problems, including the use of:

- Data Storage and Shared Memory
- Data Transfer
- Data types: unions, byte arrays, bit fields and bit masks, pointer casts

Data Storage and Shared Memory

File system data and shared memory create a unique problem because this type of data is (depending on the system design) accessible between platforms.

Problem - The endian-architecture of the platforms that access the data could be different. The format that data is written to a file or shared memory must be understood by the reading application or the content will be misinterpreted by opposite endian-architecture platforms.

Example: The big endian system writes the value "0x11223344". The little endian system reads the value as "0x44332211."

Solution A - Store the data in an Endian-neutral format. For example: use text files with string data format, or the External Data Representation (XDR) protocol. XDR is a protocol governed by the standards and formalizes a platform-independent way for computers to send data to each other.

Solution B - Specify one endian format for the stored data and always write the data in that format. Then wrap the data access with macros that are aware of the endian format of the stored data as well as the native endian format of the host processor. The macros will perform byte swapping based on a difference in formats.

Data Transfer

Data transfer is the movement of data from one system to another across a specified transmission medium.

Problem - When transferring multi-byte data between big and little endian systems, the data has to be manipulated to ensure the preservation of the "true meaning" of the data on both systems. When transferring multi-byte data from a big endian machine, the most significant byte will be in the leftmost position. When a little endian system receives the data, however, the most significant byte will be in the rightmost position unless the bytes are "swapped".

Example: The big endian system transmits the value "0x11223344". The little endian system receives the value as "0x44332211."

Solution - When multi-byte data is transferred between big and little endian systems, the bytes must be swapped in order to preserve the "true meaning" of the values. Use functions that swap the bytes like the network I/O macros to ensure the preservation of data in its true form on both big and little endian systems.

Data Types

Unions

A union is a variable that may hold objects of different types and sizes, with the compiler keeping track of the size and alignment requirements. Objects of dissimilar types and sizes can only be held at different times. A union provides a way to manipulate different kinds of data in a single area of storage.

Problem – Unions work fine for using the same memory to access different data. The key is to know what type of data exists in the memory before it is accessed. Accessing the same data with different types is not a valid use of unions and can cause endian issues.

Problem – If data types longer than 8 bits are united with a byte array, the data becomes byte order dependent.

Solution A – Don’t access the same data in memory as different data types.

Byte Arrays

Byte Arrays – A character array that is used to hold a specified number of bytes. The size of array is always equal to the number of bytes to hold.

Problem – If data in the byte array is accessed outside of its native data type, the data becomes byte order dependent.

Example: An array that is initialized with a list of characters will be read as different values between little endian and big endian platforms. The following example shows a byte array initialized to a,b,c,d. Accessing this array as a long data type on a little endian platform will result in the value 0x64636261. On a big endian platform it results in the value 0x61626364.

Solution – Avoid accessing byte arrays outside of the byte data type.

Bit Fields and Bit Masks

Bit operations are endian-sensitive. Even a bit field defined within a single byte is endian-sensitive. Code that defines bit fields is subject to Endianness conflicts when porting the code to an opposite endian platform.

Problem – In the following example, the network protocol IP header contains a bit field defined within a single byte. There are two fields within the definition, each 4 bits long (nibbles). Code that sets the value of these nibbles, iphdr.ver = 4, and iphdr.ihl = 7, will get different results if the bit field data is accessed as a byte. The result of the data read as a byte on the Big-Endian machine is 0x47, whereas the result of the data read as a byte on the Little-Endian machine is 0x74.

Also, if the data is set as a byte value, say 0x74 on the Little-Endian machine, the result of the data read as nibbles on the Little-Endian machine is a value of 4 for iphdr.ver field, and a value of 7 for iphdr.ihl field. On the Big-Endian machine the results would be a value of 7 for the iphdr.verf field, and a value of 4 for the iphdr.ihl field.

Example:

```
struct
{
     char ver:4,
     ihl:4;
} iphdr;

/*
 * A packet header may utilize bit fields.  Bit order within
 * a byte is determined by the byte order of the processor.
 * In this example we modify two nibbles of an IP header and
 * then access later as a byte.
 */
char ipbyte;
iphdr.ver = 0x4;
iphdr.ihl = 0x7;
ipbyte = *(char *)&iphdr;

if (ipbyte == 0x47)
{
   printf (“Big Endian\n”);
}
else if (ipbyte == 0x74)
{
   printf (“Little Endian\n”);
}
```

Figure 4 – IP Header Bit Fields

Solution – Instead of using the bit field structure, access the entire 8 bit value in its native data type (byte) and use a mask for the bits of each field. Masks for the 4 bits of the version field (V) and a mask for the 4 bits of the header length field (L) are represented below.

Format -	VVVV LLLL
Version field (V) bit mask -	1111 0000
Version bit mask hex value -	0xF0
Header Length field (L) bit mask -	0000 1111
Header Length bit mask hex value -	0x0F

Pointer Casts

Casting pointers changes the native meaning of the original data. Doing so will affect which data is addressed.

Problem – If the native data pointer is a 32-bit pointer and is cast to a byte pointer, depending on the Endian-architecture of the host, either the first byte or the last byte will be pointed to.

Example: Casting a pointer that stores the 32-bit value 0x11223344 to a byte pointer, the big-Endian system points to 0x11. The little-Endian system points to 0x44.

Solution – Never change the native type of a pointer. Instead, get the data in its native data type format and use byte swapping macros to access the bytes individually.

Native Data Types

When ever data is accessed outside of its native data type, conflicts can occur. It is important to note that this is true whether the size accessed is smaller or larger than the native data type.

If data is read/written outside of its native format, then the Endian format of the shared data must be known and static. For example: If a Big-Endian computer stores data to a file in Big-Endian format, a Little-Endian computer must account for the Endian difference and perform byte-swapping in order to read the data correctly. On the flip side, whenever the Little-Endian computer writes data to that same file, it must perform byte-swapping to convert the data back to Big-Endian format

Table 6 shows the conversion action that is required when accessing data outside of its native data type and on opposite endian-architectures:

Native Data Type Size	Size Accessed	Conversion
short	char	Swap both bytes
long	short	Swap both shorts end for end
long	char	• Swap bytes 0 and 3 • Swap bytes 1 and 2
double	long	Swap both longs end for end
double	short	• Swap bytes 0,1 with 7,6 • Swap bytes 2,3 with 5,4
char	short	Never. Although this may be efficient for copies, it is not a good programming practice.
short	long	“
long	double	“

Table 6 – Data Type Conversion Actions

Endian-Neutral Code

The goal of Endian-neutral code is to provide one software source-set of files that will work correctly no matter which processor Endian-architecture the code is executed on, eliminating the need to rewrite the code. The way to effectively achieve this goal is by identifying the memory and external data interfaces of the system and then implementing the use of processor independent macros to perform the interface operations. These macros automatically compile the appropriate code for the respective Endian-architecture.

Endian-neutral code makes no assumptions of the underlying platform in its implementation. Instead, it funnels all data and memory accesses through wrappers that decide how the accesses should be made. The decision is based on information that is defined during code compilation and specifies which Endian-architecture the code is being compiled to support.

Guidelines for Implementing Endian-neutral Code

Endian-neutral code can be achieved by identifying the external software interfaces and following these guidelines to access the interfaces.

Endian-neutral Coding Practices

1. Data Storage and Shared Memory – Store data in a format that is not tied to endian-architecture.

   Example:

   a. Use a format that works for all architectures, such as text files and strings, or XDR protocol.

   b. An alternative is to specify one endian format for the stored data and always write the data in that format, or use a header that specifies the endian format.

   c. Wrap the data access with macros that understand the endian format of the stored data as well as the native endian format of the host processor. The macros will perform byte swapping based on a difference in formats.

2. Byte Swap Macros – Use macros that serve as wrappers around all binary multi-byte data interfaces.

3. Data Transfer – Use network I/O macros to read/write data from the network. The macros will determine when byte swapping should occur based on whether the format of the transferred data is in the native endian format of the processor.

4. Data Types – Access data in its native data type. For example: Always read/write an “int” as an “int” type as opposed to reading/writing four bytes. An alternative is to use custom endian-neutral macros to access specific bytes within a multi-byte data type. Lack of conformance to this guideline will cause code compatibility problems between endian-architectures.

   Examples:

   a. Unions – Never use unions to access the same data with dissimilar types. See Platform Porting considerations.

   b. Byte Arrays – Never access multi-byte data as a byte array. See Platform Porting considerations.

   c. Pointer Casts – Never cast pointers. See Platform Porting considerations.

5. Bit Fields – Never define bit fields across byte boundaries or smaller than 8 bits. If it is necessary to access bit data that is not a full byte or on byte boundaries, access the entire bit field in its native data type and use a bit mask for the bits of each fields.

6. Bit Shifts – Use the “C” language << and >> constructs to move byte positions of binary multi-byte data.

7. Pointer Casts – Never cast pointers to change the size of the data pointed to.

8. Compiler Directives – Be careful when using compiler directives, such as those affecting storage (align, pack). Directives are not always portable between compilers. “C” defined directives such as #include and #define are okay. Use the #define directive to define the platform endian-architecture of the compiled code compilers.

Code Analysis

Analysis of code to determine its endian portability can result in variations of portability. I’ll refer to these variations as the Good, Bad, and Ugly. The guidelines to code analysis are as follows:

1. Review data definitions for use of unions. Unions should be considered suspect. If the unions include the use of accessing the same data with different data types, the code should be updated to remove the use of the union.

2. Review code for casting of data types. Remove all uses of accessing data outside of its native data type and replace with macros that access the data per compiler defined Endianness.

3. Review code for use of Network I/O macros. Big-Endian architectures do not require byte swapping on the TCP/IP header data so it is possible that no special code is added for this interface. However, in order to make the code portable to Little-Endian architecture byte swapping is required. The use of Network I/O macros for this interface will determine whether to byte swap or not.
4. Identify all import/export interfaces of shared data. Verify whether the interface follows the recommendations for handling shared data. If not, decide which solution should be used for each interface to make it endian-neutral.
5. Identify all memory interfaces. Verify whether the interfaces follow the recommendations for accessing data in its native data type.
6. Always use the compile bit shift directives to swap positions of bytes within data.

The Good

The code is already Endian-neutral. The code analysis did not result in any required Endianness changes.

The Bad

The code is only partially Endian-neutral. The code analysis results determined that the code uses some endian-neutral code practices, such as the use of network macros, but does not adhere to all guidelines for all imported and exported data. The code conforms to at least 50 percent of the Endian-neutral coding guidelines.

The Ugly

The code analysis results determined that there is little to no endian-neutrality designed in. The code explicitly assumes Endianness, contains use of unions and type casting pointers to change the size of the data access, or does not use Endian-neutral macros to access binary multi-byte data, or is in violation to more than 50 percent of the Endian-neutral coding guidelines.

Converting Endian-specific to Endian-neutral Code

To convert Endian-specific code to Endian-neutral code, all memory and external data interfaces are identified and analyzed. All interfaces that use endian-specific code are re-implemented by following the endian-neutral coding guidelines.

Be aware of data interfaces that use the native endian-architecture of the source host. No operations need to be performed on data that exist in the same endian format as the host. Therefore, byte swapping code will need to be added to support the new target endian-architecture.

Example: Big-Endian platforms do nothing extra to receive and transmit the TCP/IP header of network data. So, it is very likely that the network I/O macros are absent. These macros will need to be added when porting the Big-Endian code to a Little-Endian host.

The end result will be code that contains macros that act as wrappers around the data interfaces, hiding the Endianness of the code, and allowing the code to work correctly on either endian-architecture.

Reversing Endian-specific Architecture of Code

To reverse the Endian-architecture of the implemented code to the opposite endian-architecture, all memory and external data interfaces are identified. All interfaces that use endian-specific code are re-implemented by hard-coding the implementation to reverse the order of the data accesses.

As previously mentioned in “Converting Endian-specific to Endian-neutral Code”, be aware of data interfaces that use the native endian-architecture of the source host. No operations need to be performed on data that exist in the same endian format as the host. Therefore, to support the new target endian-architecture, byte swapping code may need to be added where it might not have existed before .

Example: Big-Endian platforms do nothing extra to receive and transmit the TCP/IP header of network data. So, it is very likely that the network I/O macros are absent. These macros will need to be added when porting the Big-Endian code to a Little-Endian host.

Reversing the endian-specific architecture of code requires slightly less effort than re-implementing the code for endian neutrality, since endian-neutral code requires the addition of wrappers around external data. However, it might make more sense to convert the code to be Endian-neutral in order to guarantee flexibility in the future.

Conclusion

In all, it is absolutely important to understand the format of all external data and endian architecture of the source and target processors before porting. In order to make external data formats compatible with the host processor endian-architecture, byte-swapping is sometimes required to accommodate the differences in formats. The best way to neutralize this difference is with the use of byte-swapping macros. This paper described Endianness and its affect on code portability. Following the guidelines in this paper will allow the same source code to work correctly on host processors of differing Endian-architectures, easing the effort of platform migration.

Appendix – Definitions

.1 Bi-Endian Architecture

A CPU that is capable of being configured to operate as either Big Endian Architecture or Little Endian Architecture.

.2 Big Endian Architecture

Big endian is an order in which the “big end” (most significant value in the sequence) is stored first, at the lowest storage address. The most significant byte is stored in the leftmost position. CPU architectures, such as the Sun SPARC* system, IBM’s PowerPC*, and Hewlett-Packard Precision Architecture* systems use a ``big endian’’ model, where the most significant byte is at the lowest address in memory. See Forward Byte Ordering.

.3 Byte Array

A byte array is a character array that is used to hold a specified number of bytes. Size of array is always equal to the number of bytes to hold.

.4 Data Transfer

Data transfer is the movement of data from one system to another across a specified transmission medium.

.5 Data type

Data types are used to access data in different formats. These formats specify the size of the data as well as the location. For example: char, short, int, and long all specify the size of machine defined data sizes. A structure will define a custom data type that can contain members of various sizes and residing at specific locations within the structure. A Union defines a grouping of data types that can be used to access the same data in different formats.

.6 Endian-architecture

This term is used to refer to the endian architecture of a system, either Big Endian architecture or Little Endian architecture.

.7 Endian-neutral

The code does not assume endian-architecture. All endian sensitive data interfaces are encapsulated by wrappers, such as macros, that access data in a manner respective to the endian-architecture.

.8 Endian-specific

The code is written explicitly to be either Big Endian or Little Endian architecture. Endian-specific code will not run correctly on CPUs with the opposite Endian-architecture of the implemented code.

.9 External Data Representation (XDR)

“XDR is a standard (RFC 1832) for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC*, and Cray*. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing."[2]

.10 Forward Byte Ordering

An addressing scheme in which the lowest address (or leftmost) in a multi-byte value is the most significant byte. Forward byte ordering is equivalent to big endian byte ordering.

.11 Least Significant Byte

The Least Significant Byte (LSB) is the byte in a multi-byte value that represents the smallest quantity or weight. On a big endian architecture, the right most byte (high byte) is the LSB. On a little endian architecture the leftmost byte (low byte) is the LSB. It is not possible to distinguish the least significant byte without taking byte ordering or Endianness into consideration.

Example: The 32-bit hex value 0x12345678 is stored in memory as follows:

Endian Order	**Byte 00**	**Byte 01**	**Byte 02**	**Byte 03**
Big Endian	12	34	56	78 (LSB)
Little Endian	78 (LSB)	56	34	12

.12 Little Endian Architecture

Little endian is an order in which the "little end" (least significant value in the sequence) is stored first. The most significant byte in is stored in the rightmost position. Intel, Alpha, and VAX architectures use the ``little endian'' model, where the least significant byte is at the lowest address in memory.

.13 Most Significant Byte

The Most Significant Byte (MSB) is the byte in a multi-byte value that represents the largest quantity or weight. On a big endian architecture, the left most byte (low byte) is the MSB. On a little endian architecture, the rightmost byte (high byte) is the MSB. It is not possible to distinguish the most significant byte without taking byte ordering or Endianness into consideration.

Example: The 32-bit hex value 0x12345678 is stored in memory as follows:

Endian Order	**Byte 00**	**Byte 01**	**Byte 02**	**Byte 03**
Big Endian	12(MSB)	34	56	78
Little Endian	78	56	34	12(MSB)

[2] Description of XDR from Sun Microsystems, "RFC 1832 - External Data Representation Standard", August 1995, p. 24 ftp://ftp.isi.edu/in-notes/rfc1832.txt

.14 Multi-byte Value

Multi-byte Value is a value containing more than one byte. This results in having to consider how the set of bytes is to be ordered.

Example: The value 1 is stored in a computer in binary and can be represented in hexadecimal notation as follows:

Endian Order	**Single-byte Value (8-bits)**	**Multi-byte Value (16-bits)**
Big Endian	01	00 01
Little Endian	01	01 00

.15 Union

A union is a variable that may hold objects of different types and sizes, with the compiler keeping track of the size and alignment requirements. Objects of different types and sizes can only be held at different times. A union provides a way to manipulate different kinds of data in a single area of storage.

Appendix – Abbreviations and Acronyms

Abbreviation	Description
ARM	Advanced RISC Machines Ltd.
BKM	Best Known Method
CPU	Central Processing Unit
DEC	Digital Equipment Corporation
IA	Intel Architecture
I/O	Input / Output
IBM	International Business Machines Corporation
LSB	Least Significant Byte
MSB	Most Significant Byte
MIPS	MIPS Technologies, Inc.
RISC	Reduced Instruction Set Computer

Appendix – References

Srinivasan, R. Sun Microsystems, “RFC 1832 - External Data Representation Standard”, August 1995, p. 24
ftp://ftp.isi.edu/in-notes/rfc1832.txt