

## *Microprocessor Endian Architecture*

**ABSTRACT:** 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. This article explains endianness and its effect on code portability.

### Endian-Neutral Software

Software is sometimes designed with a 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.

Endian-specific code assumes the endianness of the underlying hardware. In a nutshell, the code is endian-specific if it contains the 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.

For architecture migrations to Intel Atom processors, no endian updates are required if the source code involved in the architecture migration is designed as little-endian or endian-neutral. In fact, designing software that abstracts the OS and hardware is common practice in modern code bases. On the other hand, for the edge cases where the software design doesn't provide the abstraction layers and is hardcoded as big-endian the code will need some amount of endian updates.

This remainder of this section establishes a set of fundamental guidelines for software developers who wish to develop endian-neutral code or convert endian-specific code. These guidelines describe the software interface information that should be considered and how to convert endian-specific code to endian-neutral code.

Note: The examples in this section 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, the potential for an endian architecture conflict exists. Data can be stored in ways that are not tied to endian architecture and also in ways that define the endianness of the data.

## Definition of Endianness

Endianness is the byte order in which 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; the endian model of the CPU architecture instead dictates how the operating system is implemented.

Representing these two storage formats are two types of endian architecture, big-endian and little-endian. There are benefits to both of these endian architectures. See the section “Merits of Endian Architectures.” Big-endian architecture stores the MSB at the lowest memory address. Little-endian architecture stores the LSB at the lowest memory address. The lowest memory address of multi-byte data is considered the starting address

of the data. Table 1 depicts how the 32-bit hex value 0x12345678 is stored in memory for big-endian and little-endian architectures. The lowest memory address is represented in the leftmost position, byte 00.

**Table 1** Example of Memory Addressing for Big- and Little-endian

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

As you can see in Table 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. The details of handling the endian difference in code are thoroughly discussed in the 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 the address off set 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 off set 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.

In the past embedded communication processors and custom solutions associated with the data plane have been designed on big-endian architectures. Because of this, legacy code on these processors is often written specifically for network byte order, which is big-endian format.

Table 2 lists several popular computer systems and their endian architectures. Note that some CPUs can be either big- or little-endian, referred to as bi-endian, by setting a processor register to the desired endian architecture.

**Table 2** Computer System Endianness

Platform	Endian Architecture
ARM†	Bi-endian
DEC Alpha†	Bi-endian
HP PA-RISC 8000†	Bi-endian
IBM PowerPC†	Bi-endian
Intel® architecture	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
Sun SPARC V9†	Bi-endian

### *Relevance of Endian Order*

Endian order means that any time a computer accesses a stream, such as a network tap, local file, audio, video, or multimedia stream, the software needs to understand how the file is constructed. For example: if a graphics file, such as a .BMP file, which is little-endian format, is written out to a big-endian machine, the byte order of each integer must first be reversed. Otherwise another standardized program will not be able to read the file.

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 endian specific. Software byte swapping seems to be a viable method. Several different methods are available, and are described in the following sections.

## 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 computer systems. 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*

Several methods are 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 include `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 rotate operand right (`ror`) or rotate operand left (`rol`).
- 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 Input/Output 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 Transmission Control Protocol/Internet Protocol (TCP/IP) stack, each network host is identified by its 32-bit IP address, which is ordinarily displayed in the four 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 TCP/IP.

Network input/output (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
- Off set: 2 bytes

- Source: 4 bytes
- Destination: 4 bytes

Table 3 describes network I/O macros. The term *host* is used to refer to the processor's endian architecture and the term *network* is used to refer to the TCP/IP endian architecture. Using these macros allows for the same code to work on a big-endian or little-endian processor.

**Table 3** Network I/O Macros

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

The byte swap performed for TCP/IP communication on little-endian processors adds performance overhead. However, this overhead can be recovered as the processor speed increases. See the section "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 examples of byte swap macros for each data size.

**Table 4** Custom Byte Swap Macros

Access Size	Example Macro Name	Macro Code
16 bits	SwapTwoBytes	<pre>#include &lt;stdio.h&gt;  #define SwapTwoBytes(data) \ ( (((data) &gt;&gt; 8) &amp; 0x00FF)   (((data) &lt;&lt; 8) &amp; 0xFF00) )</pre>
32 bits	SwapFourBytes	<pre>#include &lt;stdio.h&gt;  #define SwapFourBytes(data) \ ( (((data) &gt;&gt; 24) &amp; 0x000000FF)   (((data) &gt;&gt; 8) &amp; 0x0000FF00)   \   (((data) &lt;&lt; 8) &amp; 0x00FF0000)   (((data) &lt;&lt; 24) &amp; 0xFF000000) )</pre>
64 bits	SwapEightBytes	<pre>#include &lt;stdio.h&gt;  #define SwapEightBytes(data) \ ( (((data) &gt;&gt; 56) &amp; 0x0000000000000000FF)   \   (((data) &gt;&gt; 40) &amp; 0x00000000000000FF00)   \   (((data) &gt;&gt; 24) &amp; 0x000000000000FF0000)   \   (((data) &gt;&gt; 8) &amp; 0x00000000FF000000)   \   (((data) &lt;&lt; 8) &amp; 0x000000FF00000000)   \   (((data) &lt;&lt; 24) &amp; 0x0000FF0000000000)   \   (((data) &lt;&lt; 40) &amp; 0x00FF000000000000)   \   (((data) &lt;&lt; 56) &amp; 0xFF00000000000000) )</pre>

## 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
- Runtime 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.

**Table 5** Preprocessor Control

Access Size	Example Macro Names	Macro Code
16-bit big-endian data	MY_RD_BE_SHORT MY_WRT_BE_SHORT	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN /* Do nothing */ #else <b>SwapTwoBytes</b> (data) #endif </pre>
16-bit little-endian data	MY_RD_LE_SHORT MY_WRT_LE_SHORT	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN <b>SwapTwoBytes</b> (data) #else /* Do nothing */ #endif </pre>
32-bit big-endian data	MY_RD_BE_LONG MY_WRT_BE_LONG	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN /* Do nothing */ #else <b>SwapFourBytes</b> (data) #endif </pre>
32-bit little-endian data	MY_RD_LE_LONG MY_WRT_LE_LONG	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN <b>SwapFourBytes</b> (data) #else /* Do nothing */ #endif </pre>
64-bit big-endian data	MY_RD_BE_DOUBLE MY_WRT_BE_DOUBLE	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN /* Do nothing */ #else <b>SwapEightBytes</b> (data) #endif </pre>
64-bit little-endian data	MY_RD_LE_DOUBLE MY_WRT_LE_DOUBLE	<pre> #if CPU_ARCHITECTURE == BIG_ENDIAN <b>SwapEightBytes</b> (data) #else /* Do nothing */ #endif </pre>

## Runtime Controls

It is possible to detect the endian architecture of a processor using runtime code. Figure 1 shows an example of code that performs a runtime test that checks whether the code is running on a little- or big-endian system. This allows runtime 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++;

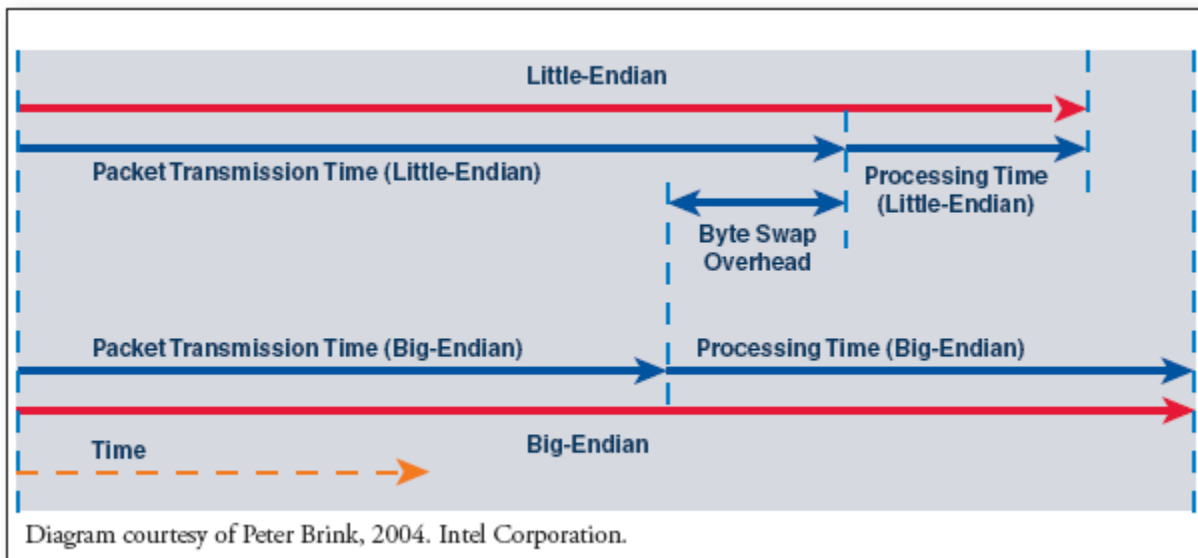
if (TestUnion.Chars == 0x61626364)
    /* It's big endian */

```

**Figure 1** Run-Time Byte Order Test (Source code example courtesy of Mark Sullivan, 2004. Intel Corporation.)

## 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 even the fastest network transmission speeds available today. The byte-swapping overhead, though it undeniably exists, can be readily recovered, especially with the latest performance capabilities of today's processors. Figure 2 depicts a time exaggerated example of the overhead required for byte swapping, as well as the reduced processing time of current little-endian processors that recover the time.



**Figure 2** Example of Byte Swap Overhead and Recovery

Figure 2 shows that some overhead is associated with swapping the bytes in the network headers. However, given a substantial increase in processor performance, the time associated with processing 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 vice 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 depending on the system design this type of data is accessible between platforms.

**Problem 1:** The endian architecture of the platforms that access the data could be different. The format in which 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 1: Accessing data across platforms**

The big endian system writes the value 0x11223344. The little-endian system reads the value as 0x44332211.

#### **Solution 1.A: Store the data in an endian-neutral format**

For example: use text files with string data format, or the External Data Representation<sup>4</sup> (XDR) protocol. XDR is a protocol governed by standards and formalizes a platform-independent way for computers to send data to each other.

#### **Solution 1.B: Specify the endian format**

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 2:** 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 2: Transferring data between big-endian and little-endian systems**

The big-endian system transmits the value 0x11223344. The little-endian system receives the value as 0x44332211.

**Solution 2: Byte swap the data**

Whenever 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

The use of certain data types, such as unions, byte arrays, bit fields, bit masks, pointer casts, can create porting problems.

### *Unions*

A union is a data type 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 3:** 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.

**Example 3: Accessing the same data with different types**

Accessing the same data with different data types is not a valid use of unions and can cause endian issues. Although the code in Figure 1 suits the purpose for the runtime byte order test to check for the endianness of the machine, it is an example of an improper use of a union. Also, if data types

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

**Solution 3: Always access the same data with the same data type**

Only use unions for their purpose of conserving space. Ensure that unions are not used to access the same data with different data types.

*Byte Arrays*

A byte array is 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 4:** If data in the byte array is accessed outside of its native data type, the data becomes byte order dependent.

**Example 4: Array initialized as a list of characters**

An array that is initialized with a list of characters will be read as different values between little-endian and big-endian platforms. A byte array is initialized as "abcd". Accessing this array as a long data type on a little-endian platform results in the value 0x64636261. On a big-endian platform it results in the value 0x61626364.

**Solution 4: Avoid accessing byte arrays outside of the byte data type.**

Accessing data outside of its natural data type breaks endian neutral code. Always access byte arrays as byte values.

*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 5:** 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 four bits long, which is a bit length also referred to as 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.ver` field, and a value of 4 for the `iphdr.ihl` field.

### Example 5: IP Header Bit Fields

The code in Figure 3 illustrates how bit fields are susceptible to endian issues. The code purposefully incorrectly sets the value for the IP header version using bit fields, and then reads the data as a byte. The value of the IP header version will be 0x47 on a big-endian machine, or 0x74 on a little-endian machine.

---

```
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 3** IP Header Bit Fields (Source code courtesy of Bob Huff.)

### Solution 5: Access the entire 8-bit value in its native char data type.

Access the value as a char data type and use a mask to access the bits of each field. Table 6 shows how the bit mask and hex values are represented for the four bits of the version field, V, and the 4 bits of the header length field, L.

**Table 6** IP Header Bit Masks

Data Name and Bits	Mask Value
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 6:** 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 6: Casting pointers**

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 6: 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

Whenever 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 or 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. Conversely, 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 7 shows the conversion action that is required when accessing data outside of its native data type and on opposite endian architectures.

**Table 7** Data Type Conversion Actions

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	Never. Although this may be efficient for copies, it is not a good programming practice.
long	double	Never. Although this may be efficient for copies, it is not a good programming practice.

## 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.

To convert endian-specific code to endian-neutral code, external interfaces that use endian-specific code will need to be re-implemented using the *Endian-Neutral Coding Guidelines* in the following section.

## Endian-Neutral Coding Guidelines

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

## 1. Data Storage and Shared Memory

Store data in a format that is not tied to endian architecture. Choices include:

- Using a format that works for all architectures, such as text files and strings, or XDR protocol.
- Specifying one endian format for the stored data and always write the data in that format, or using a header that specifies the endian format.
- Wrapping 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

Never access data outside of its native data type. 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 of data type usages that can cause issues include:

- Unions  
Never use unions to access the same data with dissimilar types. See Platform Porting considerations.
- Byte Arrays  
Never access multi-byte data as a byte array. See Platform Porting considerations.
- Pointer and Variable Typecasts  
Never use type casting to change the size of a pointer or variable. 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. Example; align and pack. Directives are not always portable between compilers. The C language defined directives, such as #include and #define, are okay. Use the #define directive to define the platform endian architecture of the compiled code compilers.

For more information about endian architecture and migration issues, please refer to the book *Break Away with Intel® Atom™ Processors: A Guide to Architecture Migration* by Lori Matassa and Max Domeika.

## About the Authors

**Lori Matassa** is a Sr. Staff Platform Software Architect in Intel's Embedded and Communications Division and holds a BS in Information Technology. She has over 25 years experience as an embedded software engineer developing software for platforms including mainframe and midrange computer system peripherals, as well as security, storage, and embedded communication devices. In recent years at Intel she has contributed to driver hardening standards for Carrier Grade Linux, and has led the software enablement of multi-core adoption and architecture migration for embedded and communication applications. Lori is a key contributor to Intel's Embedded Design Center, with numerous whitepapers, blogs, and industry contributions on a variety of topics critical to embedded migration.

**Max Domeika** is an embedded software technologist in the Developer Products Division at Intel, creating tools targeting the Intel architecture market. Over the past 14 years, Max has held several positions at Intel in compiler development, which include project lead for the C++ front end and developer on the optimizer and IA-32 code generator. Max currently provides embedded tools consulting for customers migrating to Intel architecture. In addition, he sets strategy and product plans for future embedded tools. Max earned a BS in Computer Science from the University of Puget Sound, an MS in Computer Science from Clemson University, and a MS in Management in Science & Technology from Oregon Graduate Institute. Max is the author of *Software Development for Embedded Multi-core Systems* from Elsevier. In 2008, Max was awarded an Intel Achievement Award for innovative compiler technology that aids in architecture migrations.

Copyright © 2010 Intel Corporation. All rights reserved.

This article is based on material found in book *Break Away with Intel® Atom™ Processors: A Guide to Architecture Migration* by Lori Matassa and Max Domeika. Visit the Intel Press web site to learn more about this book: [http://www.intel.com/intelpress/sum\\_ms2a.htm](http://www.intel.com/intelpress/sum_ms2a.htm)

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Publisher, Intel Press, Intel Corporation, 2111 NE 25 Avenue, JF3-330, Hillsboro, OR 97124-5961. E-mail: [intelpress@intel.com](mailto:intelpress@intel.com) .