Wholesale Byte Reversal of the Outermost Ada Record Object to Achieve Endian Independence for Communicated Data Types

Randal P. Andress
Northrop Grumman Corporation
213 Wynn Drive
Huntsville, Alabama 35805

Extended Air Defense TestBed[1] (EADTB) Project
670 Discovery Drive
Huntsville, Alabama 35806

randal.andress@ngc.com

Abstract: *For years computer engineers have dealt with the problems associated with transfer of binary data between systems of different bit and byte order – Big-Endian (most significant first) vs. Little-Endian (least significant first). In this article, existing techniques are extended to develop a new, systematic method for coding Ada record component representations so that the same declaration may be used to define an input/output object of a communications link (or "flat" file) to achieve complete endian independence. The uniqueness of this approach is that rather than dealing with byte swapping at the lowest level where the fields of the most nested record components are defined, a single, simple byte-swap is done at the outermost level of the multi-field object. This technique works for multi-layered, nested records as well as for records with fields that consume partial, contiguous bytes. It is especially well suited for implementation where endian independence has not been anticipated in the design. The only special case is that of array components which require a simple pre/post fix-up. Some potential pitfalls and compiler issues are discussed.*

Introduction

Hardware and software designers and developers alike have long struggled with bit and byte order issues. Well before the key terms were coined by Danny Cohen in his humorous and informative paper[2], the difference in the order of transmission and internal memory layout of multi-bit and multi-byte objects has been the source of frustration and annoyances, if not difficult problems. These issues come into the clearest focus when binary data transfer is required between different systems. Big-Endian (BE) systems are those which *send first* or store at the *lowest address* ("0", usually) the byte and bit of most significance (MSB first). Others, known as Little-Endian (LE), *send first* and give the *lowest address* to the element of least significance (LSB first). There are many good discussions of this issue[3] in addition to Mr. Cohen's article already cited. For those not familiar with the Big-, Little-Endian issue or for those wishing to brush up on the details I recommend *Microprocessors: A Programmer's View (Computing That Works)* (Hardcover) by Robert B. K. Dewar, Matthew Smosna, McGraw-Hill .

Network communication is the most common battleground for "endian wars". However, file data written in a serial or streaming fashion (to a UNIX "flat" file) such as that done by the UNIX function "write", may also

[1] EADTB is a robust, multi-purpose, event-driven simulation and modeling tool kit with emphasis on user control and flexibility. It is designed for battle-space modeling of land, sea, space, and airborne assets. It is a product of the United States DOD Office of Technical Integration & Interoperability (OTii), Huntsville, Alabama. For more information see www.eadtb.com.

[2] Cohen, Danny, "On Holy Wars and a Plea for Peace" [USC/ISI IEN 137, dated April 1, 1980]. Mr. Cohen associates the two ways of ordering of bits and bytes with the dispute in Jonathan Swift's *Gullivers Travels* between the Big-Endians, who broke their eggs only at the big ends, and the Little-Endians, who broke theirs only at the little ends.

[3] He, Kevin Kaichuan, "Byte and Bit Order Dissection", is one which I recently read off the internet at http://www.linuxjournal.com/article.php?sid=6788

present endian problems. For example, it may be desirable to read or write files on opposite endian systems so that they can inter-operate in some way. Endian independence can sometimes ease the task of re-hosting a process to a system of different endianness. Consider, for example, a multi-process software system that has network communication components. If the communicating components were written to be endian independent, these processes could be re-hosted in stages, one process at a time. Similarly, if the re-host subject used persistent data files, it could be used on either system as it was being ported.

Just as endian problems have been around for some time, so have methods of dealing with them. In the case of network messages, one technique is to restrict multi-byte data at the point of communication to one of several simple types such as 1-, 2- or 4-byte integers and floats. Then calls are made to one of a few simple functions to byte-swap these objects before sending (htons, htonl[4]) or receiving (ntohs, ntohl[5]). On systems whose internal order is the same as network order, these routines do not alter their input data[6]. But on systems with opposite order, they perform the required byte swap. The External Data Representation Standard (XDR) is another much more flexible and extensive mechanism which provides for communication of complex data types including arrays as well as C-like structures and unions. In Ada 95 software systems, Ada.Streams and its child package, Stream_IO[7], provide a mechanism to capture the input and output of object components at their lowest level. The T'Write' and T'Read' routines give opportunity to make whatever byte or bit conversions that may be necessary before or after output/input.

This paper presents yet another method of dealing with endian order in an Ada program. It is especially suited to those cases where the requirement for endian independent I/O was not anticipated in the original software design or when compatibility of data files or communications is needed when re-hosting software to an opposite endian system. This method also has the advantage of being suitable for Ada 83 software with a small inconvenience of having to hardcode the few constants differently depending on the "endianness" of the system.

Existing Methods Extended

In a 1994 *Ada Letters* article[8], Norman Cohen presented a helpful method of representing Ada 83 record components in a way that confines the endian dependence to a few constants. One constant is `Rightward_One_Bit`, which is given a value of +1, or –1, indicating the change in bit number as one proceeds from left to right through a storage unit. In addition to this value, there is one additional constant for the size of each component data type being represented. It is set to the number of the most significant bit for that component size. If bytes (8-bits), 16-bit halfwords, 32-bit words, and 64-bit doublewords are to be represented, the following are required:

for Big-Endian systems:

```
Byte_MSB_Number       : constant :=  0;
Halfword_MSB_Number   : constant :=  0;
Word_MSB_Number       : constant :=  0;
DoubleWord_MSB_Number : constant :=  0;
Rightward_One_Bit     : constant := +1;
```

and for Little-Endian systems:

```
Byte_MSB_Number       : constant :=  7;
Halfword_MSB_Number   : constant := 15;
Word_MSB_Number       : constant := 31;
DoubleWord_MSB_Number : constant := 63;
Rightward_One_Bit     : constant := -1;
```

[4] Host TO Network Short or Long, hton[sl].
[5] Network TO Host Short or Long, ntoh[sl].
[6] Network order is almost always Big-Endian.
[7] Ada.Streams, Ada Language Reference Manual, paragraph 13.13.1.
[8] Cohen, Norman, *Ada Letters*, Vol. XIV, No. 1, Jan/Feb 1994.

On a Big-Endian machine the usual representation clause for the bits within a 32-bit word component, C,

```
C at <byte-offset> range L .. R;
```

may be written as:

```
C at <byte-offset> range
      Word_MSB_Number + Rightward_One_Bit*(L+R) + L – R) / 2 ..
      Word_MSB_Number + Rightward_One_Bit*(L+R) – L – R) / 2;
```

For Ada 83, this technique requires different coding for the definitions for the constants depending on the system bit order. However, with the advent of Ada 95 and the definitions in package `System`:

```
type Bit_Order is (High_Order_First, Low_Order_First);
Default_Bit_Order : constant Bit_Order[9];
```

these constants my be written in an endian-independent way such as:

```
Byte_MSB_Number       : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 7;
Halfword_MSB_Number   : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 15;
Word_MSB_Number       : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 31;
DoubleWord_MSB_Number : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 63;
Rightward_One_Bit     : constant :=
      1 - (2 * Boolean'Pos(Default_Bit_Order = Low_Order_First));
```

This makes the representation for, C, above to be completely endian independent.

Building on the technique presented by Mr. Cohen, some algebraic manipulation and the definition of two additional constants, F1 and F2, yields a refinement that makes it a little more attractive and easy to visualize, so that we may write:

```
C at <byte-offset> range MSBn + L*F1 + R*F2 .. MSBn + R*F1 + L*F2;
```

or if preferred, swapping F1 and F2 rather than L and R in the upper bound:

```
C at <byte-offset> range MSBn + L*F1 + R*F2 .. MSBn + L*F2 + R*F1;
```

where

`n` is the number of bits in the component being represented and

```
MSBn : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * (n-1);

Rightward_One_Bit : constant :=
   1 – 2 * Boolean'Pos(Default_Bit_Order = Low_Order_First));

F1 : constant := (Rightward_One_Bit + 1)/2;
F2 : constant := (Rightward_One_Bit - 1)/2;
```

[9] `Default_Bit_Order` is given the appropriate value on each specific system.

```
-- for BE, F1 = +1; for LE, F1 =  0
-- for BE, MSBn = 0; for LE, MSBn = n-1
-- for BE, Rightward_One_Bit = +1; for LE, Rightward_One_Bit = -1;
-- for BE, F2 = 0; for LE, F2 = -1
```

Using this technique, representations are easy to churn out from their normal Big-Endian form. For example, consider the following 2-byte record consisting of a 3-bit color field, an 11-bit speed field and a 2-bit heading field.

```
type Color_T is  (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
 for Color_T use (Red  => 1, Orange => 2, Yellow => 3, Green => 4,
                  Blue => 5, Indigo => 6, Violet => 7);
 For Color_T'size use 3;

type Speed_T is mod 2**11;
 for Speed_T'size use 11;

type Heading_T  is (North, South, East, West);
 for Heading_T use (North => 0, South => 1, East => 2, West => 3);
 for Heading_T'size use 2;

type Sighting_Rec is
   record
      Color   : Color_T;
      Speed   : Speed_T;
      Heading : Heading_T;
   end record;
for Sighting_Rec'size use 16;
```

The representation for `Sighting_Rec` would normally be written in Big-Endian as:

```
for Sighting_Rec use
   record
      Color   at 0 range  0 ..  2;
      Speed   at 0 range  3 .. 13;
      Heading at 0 range 14 .. 15;
   end record;
```

Using our endian independent shorthand, we can write the same clause as:

```
MSB16 : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 15;

-- for BE, F1 = 1, F2 = 0, and MSB16 = 0
for Sighting_Rec use
   record
      Color   at 0 range MSB16 +  0*F1 +  2*F2 ..
                         MSB16 +  2*F1 +  0*F2;    --  0 ..  2
      Speed   at 0 range MSB16 +  3*F1 + 13*F2 ..
                         MSB16 + 13*F1 +  3*F2;    --  3 .. 13
      Heading at 0 range MSB16 + 14*F1 + 15*F2 ..
                         MSB16 + 15*F1 + 14*F2;    -- 14 .. 15
   end record;
```

which, for the Little-Endian case, where `F1 = 0, F2 = -1, and MSB16 = 15,` evaluates to:

```
for Sighting_Rec use
   record
      Color   at 0 range 13 .. 15;
      Speed   at 0 range  2 .. 12;
      Heading at 0 range  0 ..  1;
   end record;
```

Note that for the Little-Endian case, the fields of the record are in reverse order with `Color` at the end of the record instead of at the beginning.

This method of endian independent representation (using `MSBn, F1, F2, L, R`) is considerably more verbose than the standard (`L .. R`) , but there is a certain regularity about it that makes it quite manageable. The fact that it enables us to specify the same bits on a Big- or Little- Endian system using identical source will be, in many cases, worth the trouble.

A New Twist

Finally, we come to the challenge of input and output of multi-component records in an endian-independent manner – using the same code on both sides of the interface. As mentioned earlier, converting from one endianness to the other is usually accomplished by swapping the bytes of each low-level component before writes and after reads. What is presented here is an alternative whereby the entire record is converted to or from Big- or Little-Endian, by a single byte reversal of the entire record on the Little-Endian system before it is written and after it is read. But in order for this to work, the component bit ranges must be specified in a certain way.

Normally when it is necessary to represent multi-component records, the bit range designations are made from a convenient byte offset within the record. A 1-byte integer field, `Int_8`, for example, located at the end of a 7-byte (`T'size = 56`) record would be designated as:

```
Int_8 at 6 range 0..7;
```

However, the quite acceptable and exact equivalent of this is

```
Int_8 at 0 range 48..55;
```

In order for the wholesale byte reversal technique to accomplish endian conversion, every component of the entire record (and sub-records, if any) must be represented at offset '0', using bit numbers from '0' to the number of bits in the record type minus one (`T'size - 1`). As an example, let's define a record containing an `Integer` (32-bits) and `Sighting_Rec(`16-bits) as defined above.

```
type IO_Rec is
   record
      Int_32   : Integer;
      Sighting : Sighting_Rec;
   end record;
for IO_Rec'size use 48;
```

Normally the representation clause would be written with Int_32 at byte offset 0 and Sighting at byte offset 4 like this:

```
for IO_Rec use
   record
      Int_32   at 0 range 0 .. 31;
      Sighting at 4 range 0 .. 15;
```

```
      end record;
```

However using the ‘at 0’ offset method, the equivalent is:

```
   for IO_Rec use
      record
         Int_32   at 0 range  0 .. 31;
         Sighting at 0 range 32 .. 47;
      end record;
```

Now using our shorthand, endian-independent technique this becomes, for Big-Endian:

```
   MSB48 : constant :=
      Boolean'Pos(Default_Bit_Order = Low_Order_First) * 47;
   for IO_Rec use
      record
         Int_32   at 0 range MSB48 +  0*F1 + 31*F2 ..
                             MSB48 + 31*F1 +  0*F2;  --  0 .. 31
         Sighting at 0 range MSB48 + 32*F1 + 47*F2 ..
                             MSB48 + 47*F1 + 32*F2;  -- 32 .. 47
   end record;
```

which for Little-Endian (`MSB48 = 47, F1 = 0, F2 = -1`) evaluates to:

```
   for IO_Rec use
      record
         Int_32   at 0 range  8 .. 47;
         Sighting at 0 range  0 .. 16;
      end record;
   My_IO_Rec : IO_Rec;
```

Now let’s suppose that an instance of `IO_Rec` is assigned the following:

```
   My_IO_Rec.Int_32           := 16#0c0a0f0e#
   My_IO_Rec.Sighting.Color   := Orange;   -- value 2, bits  0 ..  2
   My_IO_Rec.Sighting.Speed   := 16#122#   --          bits  3 .. 13
   My_IO_Rec.Sighting.Heading := South;    -- value 1, bits 14 .. 15;
```

Let’s look at reversing the bytes of the Big-Endian version and see if we get the Little-Endian version. We have a total of 6 bytes that may be shown in their Big-Endian form as:

```
     Byte:         0        1        2        3        4        5
      Bit:  01234567 01234567 01234567 01234567 01234567 01234567
   Binary:  00000000 00001010 00001111 00001110 01000100 10001001
Hex/Field:     0   c    0   a    0   f    0   e cccssssss ssssssshh
```

To illustrate the byte inversion of the entire `My_IO_Rec` we can leave the data in place and show Little-Endian numbering of bits and bytes:

```
     Byte:         5        4        3        2        1        0
      Bit:  76543210 76543210 76543210 76543210 76543210 76543210
   Binary:  00001100 00001010 00001111 00001110 01000100 10001001
Hex/Field:     0   c    0   a    0   f    0   e cccssssss ssssssshh
```

Or we can swap bytes of data and renumber the bit positions:

```
      Byte:         0        1        2        3        4        5
       Bit:  76543210 76543210 76543210 76543210 76543210 76543210
    Binary:  10001001 01000100 00001110 00001111 00001010 00001100
Hex/Field:  ssssssshh cccssssss    0   e    0   f    0   a    0   c
```

Recalling the Little-Endian representation clauses for both the outer `IO_Rec` and the `Sighting_Rec`, we find the Little-Endian layout in place as expected for all fields:

```
Sighting.Heading (‘hh’)          at bits  0 ..  1
Sighting.Speed   (‘sssssssssss’  at bits  2 .. 12
Sighting.Color   (‘ccc’)         at bits 13 .. 15
Int_32           (0x0c0a0f0e)    at bits 17 .. 47
```

Let’s take a closer look at the `Sighting` sub-record and in particular to the `Speed` field since it is comprised of bits from contiguous bytes. Expanding Big-Endian bytes 4 and 5 and introducing “`x0 .. xn`” nomenclature, we have

```
                        4                         5  (byte number)
0  1  2  3  4  5  6  7    0  1  2  3  4  5  6  7  (bit pos’n in byte)
0  1  2  3  4  5  6  7    8  9  10 11 12 13 14 15 (bit pos’n from 0)
c2 c1 c0 s10s9 s8 s7 s6   s5 s4 s3 s2 s1 s0 h1 h0 (data field bits)
0  1  0  0  0  1  0  0    1  0  0  0  1  0  0  1  (bit values we set)
```

where `c0 .. c2, s0 .. s10, h0 .. h1` are the bit ranges for `Color, Speed,` and `Heading` respectively, and where `x0` is the LSB and `xn` is the MSB for a field of n+1 bits. Inverting it within the 5 byte message stream and viewing Little-Endian bytes 0 and 1

```
                        0                         1  (byte number)
7  6  5  4  3  2  1  0    7  6  5  4  3  2  1  0  (bit pos’n in byte)
7  6  5  4  3  2  1  0    15 14 13 12 11 10 9  8  (bit pos’n from 0)
1  0  0  0  1  0  0  1    0  1  0  0  0  1  0  0  (bit values we set)
s5 s4 s3 s2 s1 s0 h1 h0  c2 c1 c0 s10s9 s8 s7 s6  (data field bits)
```

we can see the correspondence of the `Speed` field bits `s0 .. s10` of the Big-Endian bits `3 .. 13` and the Little-Endian `bits 2 .. 12`. Similarly, we find the Color and Heading bits, `c0..c2` and `h0 .. h1` in place at `14 .. 15` and `0 .. 1`, respectively.

Implementation

Wholesale byte reversal of the I/O record type may be accomplished with a single generic procedure or function such as:

```
generic
   type Record_type is private;
procedure Order_Bytes (IO_Record : in out Record_Type);
```

or

```
generic
   type Record_type is private;
function Order_Bytes (In_Val : Record_Type) return Record_Type;
```

There are a few issues with this method that should be pointed out. Probably the most significant is the failure of this method to render array components directly usable. On the Little-Endian system after reading and byte swapping the record and before byte swapping in preparation for a write, any array components must be inverted, otherwise `My_Array [My_Array’first]` will address `My_Array [My_Array’Last]` and vice versa. This inversion is easily done for all array component types of byte size or larger with a single generic procedure with the following spec:

```
generic
   type Array_Type is private;
   type Array_Component_Type is private;
procedure Invert_Components (IO_Array : in out Array_Type);
```

For those whose components are bit-packed such as a packed array of 1-bit boolean, a custom generic or a non-generic solution is needed. Interestingly, there is an outstanding enhancement request[10] at Ada Core Technologies (ACT) to implement

```
pragma Invert_Array_Indexing (array-type, Bit_Order)
```

which would be an extremely clean and efficient way to accomplish the required reversal in these cases.

A second issue has to do with the sub-optimal component alignment which could result from inverting a record whose components meet alignment requirements in the Big-Endian version, but which are not met in the inverted version. This can happen if the record `T'size` is not a multiple of the largest alignment requirement for the components in the record. This misalignment can be prevented by either of two alternatives. The first is by specifying an increased `T'size`[11], rounding it up to the next multiple of the largest alignment needed, for example:

```
type Pad_24_Bits_Array_T is array 1..24 of Boolean;
 for Pad_24_Bits_Array_T'size use 24

type IO_Rec is
   record
      Int_32        : Integer;
      Sighting      : Sighting_Rec;
      Alignment_Pad : Pad_24_Bits_Array_T;
   end record;

for IO_Rec use
   record
      Int_32        at 0 range MSBw +  0*F1 + 31*F2 ..
                               MSBw + 31*F1 +  0*F2;  --  0 .. 31
      Sighting      at 0 range MSBw + 32*F1 + 47*F2 ..
                               MSBw + 47*F1 + 32*F2;  -- 32 .. 47

      Alignment_Pad at 0 range MSBw + 48*F1 + 63*F2 ..
                               MSBw + 63*F1 + 48*F2;  -- 48 .. 63
end record;
for IO_Rec'size use 64;
```

In cases where you are not at liberty to change the size of the communicated record, go ahead and build the poorly aligned endian independent records to do the I/O. Then create a well aligned working record and have the I/O processing (after/before doing the endian-independent swaps and inversions as required) include a copy to/from this working record if necessary (little-endian only), thus isolating the inefficiency associated with sub-optimal alignment to a single load/store sequence. For example, after reading `My_IO_Rec` into a Little-Endian system:

```
My_Aligned_Work_Rec.My_Long_Float := My_IO_Rec.My_Long_Float;
```

[10] Tracking number TN9103-010.

[11] Adding a padding field at the end of the record may also be preferred so as to keep all bits in the record addressable.

```
My_Aligned_Work_Rec.My_Tiny_Int   := My_IO_Rec.My_Tiny_Int;
```

and before any other processing when writing `My_IO_Rec` from a Little-Endian system:

```
My_IO_Rec.My_Long_Float := My_Aligned_Work_Rec.My_Long_Float;
My_IO_Rec.My_Tiny_Int   := My_Aligned_Work_Rec.My_Tiny_Int;
```

The use of derived types here could also offer a convenient solution.

A third consideration is actually a caution against inadvertently cascading this technique with any other of the more traditional approaches. For example, if Ada.Streams is used, be sure that the T‘input and T’Output routines have not already been coded to include byte swapping. Likewise any other I/O software you are already using may take endianness into account.

Summary

By (1) defining a few constants, (2) coding record representation clauses using ‘0’ byte offsets for each component, and (3) performing wholesale byte reversal of the I/O record just before output and after input (on Opposite-Endian[12] systems), complete endian independent communications or file I/O can be accomplished. Identical source code may be used in Ada 95 regardless of the endianness of the target. For Ada 83 a few constants, namely `Rightward_One_Bit` and all `MSBn`, will require adjustments either manually or via a preprocessor. This technique is well suited for retrofitting software where endian issues were not a design consideration. It also may be helpful when re-hosting software from one type system (BE or LE) to the other. There are two important special considerations when using this technique. First, array components must have their elements reversed before the inversion of the record on writing and after inversion on reading. And second, records should be padded. if possible, so that their `T'size` is a multiple of the largest component alignment requirement.

[12] Here we have assumed Big-Endian network and file byte order as well as Big-Endian baseline representation clauses. The technique, however, works equally well if the “opposite” assumptions are made.