# A Proposal for Endian-Portable Record Representation Clauses

Norman H. Cohen

## What problem are we solving?

There are at least two “endian problems”. One is the run-time problem of transferring data between a big-endian machine and a little-endian machine (or between big-endian and little-endian processes of a biendian machine). Another is the compile-time problem of specifying a data layout portably, so that a single version of an Ada source file will produce the same layout regardless of the compiler’s default bit ordering. This is a proposal to solve the second problem.

## What is the semantic difference between big-endian and little-endian execution?

Most machine instructions operate on small values that we shall call *machine scalars*. Typical machine scalars include an 8-bit byte, a 16-bit halfword, and a 32-bit word. (To simplify the presentation, this document will, without loss of generality, be phrased in terms of an architecture supporting at least these three kinds of machine scalars.) A machine scalar generally fits in a register or a pair of registers. In a RISC architecture, all machine scalars are loaded into registers before being operated upon, while in a CISC architecture, one or more machine-scalar operands may reside in storage.

The only difference between big-endian and little-endian execution is the correspondence between a sequence of two or more bytes in memory, starting at a given address and extending to higher-addressed bytes, and machine-scalar values. (A sequence of bytes in memory is mapped to a machine-scalar value upon being loaded into a register or upon being used as an operand of a CISC register-storage instruction, and a machine-scalar value is mapped to a sequence of bytes in memory upon being stored.) In little-endian execution, the lowest-addressed byte corresponds to the low-order eight bits of the machine-scalar value, while in big-endian execution, the lowest-addressed byte corresponds to the high-order eight bits of the machine-scalar value.

## What do we mean by the “same” layout, and why is it important?

A program that never views the same bits as belonging to two different types (and never performs binary I/O, which is tantamount to viewing the raw contents of a file as the representation of some type) is inherently endian-independent. That is, barring impediments to portability unrelated to bit order, the program is portable between big-endian and little-endian machines. A source program that views the same storage as belonging to more than one type can also be endian-independent, provided that an appropriate programming discipline is followed and provided that object code is generated from the source code in a manner consistent with the target execution bit order.

A program viewing the same storage as belonging to more than one type will typically depend on properties like the following:

- A given machine scalar occurs at a specified offset within a record.
- The bits of a machine scalar can be subdivided into contiguous fields of specified sizes, occurring in a specified order from most significant bits to least significant bits.

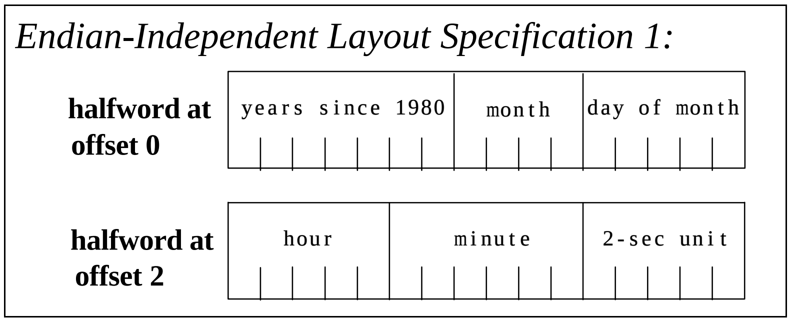
For example, the DOS 32-bit representation of a date and time can be described as follows:

- The representation consists of a halfwords at offset 0 and a halfword at offset 2. (For consistency throughout this document, we refer to a 16-bit machine scalar as a “halfword”, even though it is called a “word” in the Intel architecture.)
- In the halfword at offset 0, the high-order seven bits give the number of years since 1980, the middle four bits give the month of the year, and the low-order five bits give the day of the month.
- In the halfword at offset 2, the high-order five bits give the hour of the day, the middle six bits give the minute of the hour, and the low-order five bits give the number of two-second units within the minute.

Notice that we have described the properties of this data representation in a manner that is independent of big-endian and little-endian conventions. We shall refer to sets of properties that can be described in this way as *endian-independent layout specifications*. Suppose a compiler could be instructed, using notation independent of the bit ordering on the target machine, to produce a storage layout that obeys a given set of endian-independent layout specifications on the target machine. Then the source file of a program that depended only upon those specifications could be compiled for either a big-endian target or a little-endian target.

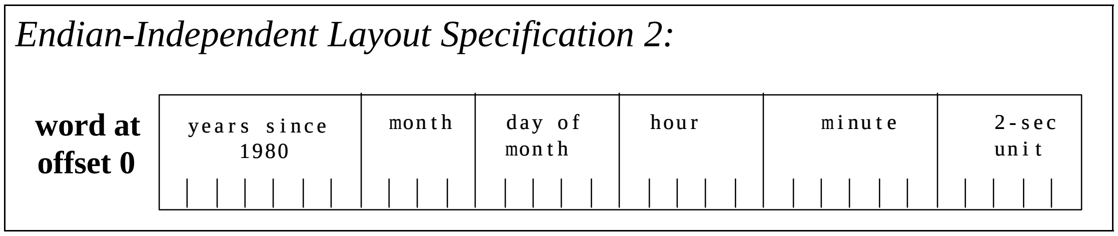
There are many ways in which programs can exploit endian-independent layout specifications. A program might depend on the date information residing at the lower-addressed halfword so that pointers to date-and-time structures could be passed to subprograms expecting only pointers to structures containing the three date components. (This is a form of homegrown polymorphism through record extension.) A program might depend on the left-to-right ordering of components within a halfword so that two dates, or two times, could be compared by a single 16-bit unsigned-integer comparison.

Endian-independent layout specifications can be represented graphically by vertically stacking machine scalars that must occur at specified byte offsets (drawing machine scalars with lower offsets at the top) and by drawing fields within machine scalars so that fields with less significant bits are to the left of fields with more significant bits. For example, the DOS 32-bit representation of a date and time can be depicted as follows:

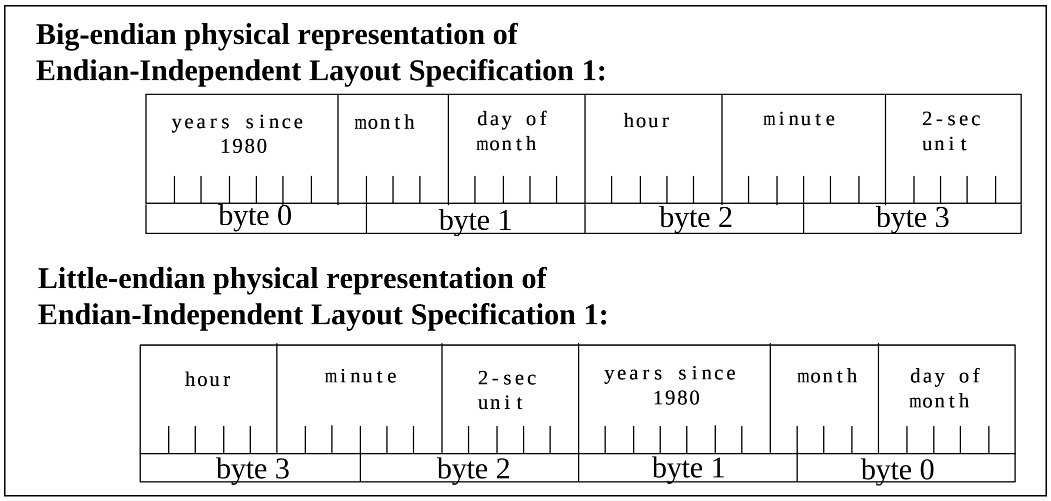


When we say that a storage representation is "the same" on both a big-endian and little-endian machine, we mean, in effect, that it is described in each case by the same such picture.

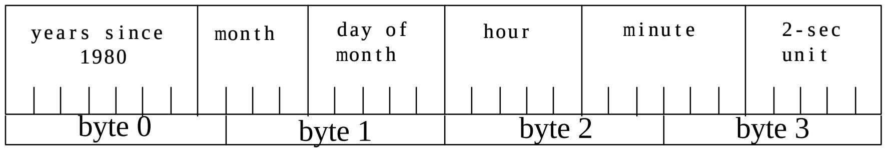
This picture expresses not the physical positions of components in memory, but the endian-independent layout specifications upon which the program depends. For example, we might also have drawn the DOS representation of a date and time as follows:



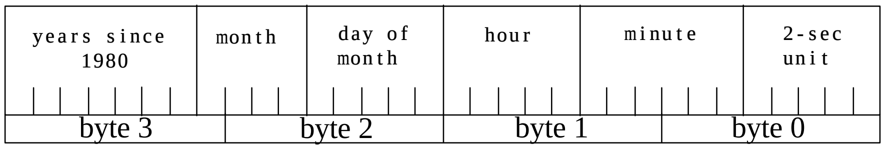
This picture corresponds to exactly the same memory mapping on a big-endian machine, but it specifies a different set of layout specifications to be preserved across bit orders, and thus a different memory mapping on a little-endian machine:


Big-endian physical representation of Endian-Independent Layout Specification 1:
years since 1980
month
day of month
hour
minute
2-sec unit
byte 0
byte 1
byte 2
byte 3
Little-endian physical representation of Endian-Independent Layout Specification 1:
hour
minute
2-sec unit
years since 1980
month
day of month
byte 3
byte 2
byte 1
byte 0

**Big-endian physical representation of Endian-Independent Layout Specification 2:**



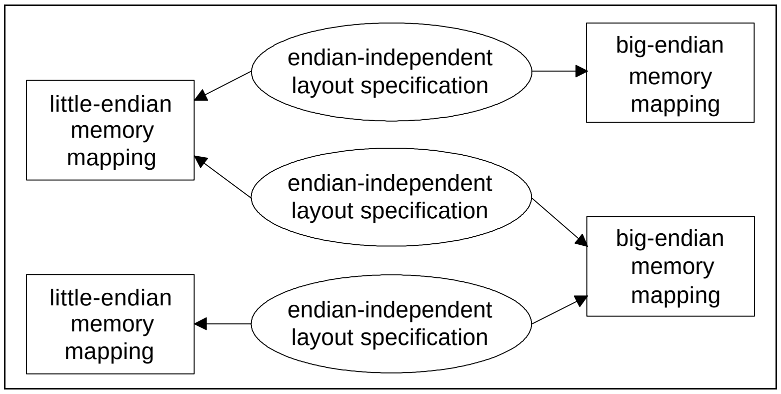
**Little-endian physical representation of Endian-Independent Layout Specification 2:**



Endian-Independent Layout Specification 2 specifies the left-to-right ordering of all six components within a single 32-bit machine scalar, and thus allows two time-and-date values to be compared by a single 32-bit unsigned-integer comparison. However, Endian-Independent Layout Specification 2 does not stipulate that the three date components reside in the halfword at offset 0, so a program that uses pointers to time-and-date structures as if they were pointers to date structures will not be portable to little-endian machines.

As this example illustrates, certain combinations of layout specifications can be maintained consistently when changing bit order, and others cannot. It is impossible to preserve both the left-to-right ordering of all six components *and* the offset of the halfword containing the date components across both bit orderings. Thus one cannot write an endian-independent program depending on all of these properties. However, layout specifications that specify only the size and location of nonoverlapping machine scalars, plus the size and left-to-right ordering of fields within those machine scalars, can be maintained consistently when changing bit order.

Each endian-independent layout specification determines one big-endian memory mapping and one little-endian memory mapping. A given memory mapping on a given machine may satisfy many different endian-independent layout specifications. Distinct endian-independent layout specifications that are satisfied by the same memory mapping in one bit ordering are satisfied by distinct memory mappings in the opposite bit ordering.


endian-independent layout specification
big-endian memory mapping
little-endian memory mapping
endian-independent layout specification
big-endian memory mapping
little-endian memory mapping
endian-independent layout specification

## Bit numbers within record-representation clauses

A storage layout for a record type can be specified in Ada by a *record-representation clause*. Such a clause specifies the location of each component in memory in terms of a specified range of bits, numbered relative to bit zero of the storage unit (on typical architectures, the byte) at a specified offset. However, whether bits are numbered left-to-right or right-to-left by default depends on the compiler. Thus, Endian-Independent Layout Specification 1 might be specified by the record-representation clause

```
for Date_And_Time_Type use
   record
      Years_Since_1980 at 0 range  0 ..  6;
      Month            at 0 range  7 .. 10;
      Day_Of_Month     at 0 range 11 .. 15;
      Hour             at 2 range  0 ..  4;
      Minute           at 2 range  5 .. 10;
      Seconds          at 2 range 11 .. 15;
   end record;
```

for a big-endian target machine, and by the record-representation clause

```
for Date_And_Time_Type use
   record
      Years_Since_1980 at 0 range  9 .. 15;
      Month            at 0 range  5 ..  8;
      Day_Of_Month     at 0 range  0 ..  4;
      Hour             at 2 range 11 .. 15;
      Minute           at 2 range  5 .. 10;
      Seconds          at 2 range  0 ..  4;
   end record;
```

for a little-endian target machine.

We would prefer to be able to write and maintain a single record-representation clause that would produce the appropriate memory mapping for the target machine whether that machine is big-endian or little-endian. At first glance, the *bit-order clause*, a new feature of Ada 95, appears to provide a solution: The bit-order clause

```
for Date_And_Time_Type'Bit_Order use High_Order_First;
```

specifies that bit numbers should be interpreted according to big-endian conventions in a record-representation clause for `Date_And_Time_Type` (i.e., with bit 0 being the high-order bit of a byte), regardless of the compiler's default bit order, while the bit-order clause

```
for Date_And_Time_Type'Bit_Order use Low_Order_First;
```

specifies those bit numbers should be interpreted according to low-endian conventions. Thus, either the first bit-order clause followed by the first record-representation clause, or the second bit-order clause followed by the second record-representation clause, should produce the appropriate storage layout for the target, regardless of whether the target is big-endian or little-endian.

Unfortunately, matters are not so simple. The Ada standard gives compilers for most machines permission to reject a bit-order clause that specifies the nondefault bit order. In the nondefault bit order, there are multiple, distinct meanings we can ascribe to bit numbers greater than or equal to the number of bits in a byte. The drafters of the Ada-95 standard implicitly ascribed a meaning that allows the specification of impractical memory mappings. To avoid requiring compilers to support these impractical mappings, they chose not to require compilers to support nondefault bit orders.

## The meaning of large bit numbers in record representation clauses

The meaning of large bit numbers in the *default* bit order is well understood: Assuming eight-bit bytes, bit 8+*b* of byte *a* is the same bit as bit *b* of byte *a*+1. Consequently, there are redundant ways to specify the same storage layout. For example, the big-endian record-representation clause could have been written equivalently (for a compiler whose default bit order is big-endian) as follows:

```
for Date_And_Time_Type use
   record
      Years_Since_1980 at 0 range 0 ..  6;
      Month            at 0 range 7 .. 10;
      Day_Of_Month     at 1 range 3 ..  7;  -- was 0 range 11 .. 15
      Hour             at 2 range 0 ..  4;
      Minute           at 2 range 5 .. 10;
      Seconds          at 3 range 3 ..  7;  -- was 2 range 11 .. 15
   end record;
```

Our proposal exploits this redundancy to express endian-independent layout specifications. Under this proposal, the two big-endian record-representation clauses specify the same memory mapping, but a different set of endian-independent layout specifications.

While the meaning of large bit numbers is obvious in the default bit order, it is not obvious in the nondefault bit order. Suppose we and compile a big-endian record-representation clause, together with a big-endian bit-order clause, for a little-endian target. The record-representation clause includes the component clause

```
Minute at 2 range  5 .. 10;
```

specifying that the `Minute` component is to occupy the following six bits:

- bit 5 (big endian) of byte 2
- bit 6 (big endian) of byte 2
- bit 7 (big endian) of byte 2
- “bit 8” (big endian) of byte 2
- “bit 9” (big endian) of byte 2
- “bit 10” (big endian) of byte 2

If we adhere to the definition that bit 8+*b* of byte *a* is the same bit as bit *b* of byte *a*+1, then this is equivalent to the following bits:

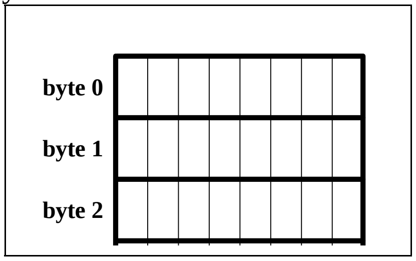
- bit 5 (big endian) of byte 2
- bit 6 (big endian) of byte 2
- bit 7 (big endian) of byte 2
- bit 0 (big endian) of byte 3
- bit 1 (big endian) of byte 3
- bit 2 (big endian) of byte 3

Under big-endian bit-numbering rules, these are the three rightmost (i.e., least significant) bits of byte 2 and the three leftmost (i.e., most significant) bits of byte 3. However, on a little-endian target, these two groups of bits will not be adjacent in a machine scalar corresponding to bytes 2 and 3. This would make it difficult to extract and update the `Minute` component. (An analogous problem arises if we try to compile a little-endian record-representation clause with a little-endian bit-order clause for a big-endian machine.)

To avoid this problem, the drafters of the Ada-95 standard chose to reject mandatory support for nondefault bit orders. We prefer to reject the definition of bit 8+*b* of byte *a* to be the same bit as bit *b* of byte *a*+1 in the nondefault bit order. We adopt a different definition that turns out to be equivalent to that definition in the default bit order, but different, and more useful, in the nondefault bit order.

## Avoiding noncontiguous bit fields

Contiguity of bits is a meaningful notion within a machine scalar, or within a single byte of memory, but not between bits in different bytes of memory. The notion of bit position within a byte is inherent in the binary representation of the numeric value contained in the byte. Our view of bit contiguity within memory is conveyed by depicting memory with bits arranged left-to-right in bytes that are stacked vertically:



Although bits in different bytes may *correspond to* adjacent bits in a machine scalar (i.e., the bits become adjacent when loaded into a register), there is no notion of two bits in different bytes of *memory* being adjacent. (In the context of a single machine with a known byte order, it is common practice to depict bytes side by side, with addresses increasing left-to-right for big-endian machines and right-to-left for little-endian machines. These depictions are convenient because bits are depicted in memory in the same left-to-right order as in the corresponding machine scalar. However, such depictions are meaningless for machines of opposite byte order.)

Consequently, the notion of a range of bits only makes sense with respect to a machine scalar. When we say

```
Minute at 2 range  5 .. 10;
```

we are referring to a range of bits in a *machine scalar* corresponding to the memory beginning at offset 2. This definition refers to a contiguous range of bits in the machine scalar regardless of the bit ordering.

In the default bit order, we number bits from zero starting in the lowest-addressed byte of the machine scalar. Thus we can readily determine the number assigned to a bit position at a given distance from the low-address end of the machine scalar. However, in the nondefault bit order, we number bits from zero starting in the highest-addressed byte of the machine scalar, so the number assigned to a bit position at a given distance from the low-address end of a machine scalar depends on the length of the machine scalar.

Therefore, we adopt the following convention: In a record-representation clause for the nondefault bit order, there is a one-to-one correspondence between byte offsets (i.e., the numbers appearing between the words "**`at`**" and "**`range`**") and machine scalars. That is, all components whose positions are specified with the same byte offset are assumed to be part of the same machine scalar (so that in typical implementations they will be loaded into a register together); and any two components required to reside within the same machine scalar have their positions specified in terms of the same byte offset. The length of a machine scalar is inferred from the highest bit number specified along with its byte position in some component clause, rounded up to the next multiple of `System.Storage_Unit.`

(This approach requires the explicit declaration of "filler" fields when the byte of a machine scalar containing the high-numbered bits is to be left unused. If no use is made of the record component `Filler`, the declarations

```
for R'Bit_Order use X;
for R use
   record
      C at 0 range 0 .. 23;
   end record;
for R'Size use 32;
```

and the declarations

```
for R'Bit_Order use X;
for R use
   record
      C      at 0 range  0 .. 23;
      Filler at 0 range 24 .. 31;
   end record;
for R'Size use 32;
```

are equivalent on compilers whose default bit order is `X`. However, on compilers with the opposite default bit order, the first set of declarations places C in the bytes at offsets 0, 1, and 2, while the second set of declarations places C in the bytes at offsets 1, 2, and 3.)

This interpretation of bit numbers makes it feasible to require compiler support for the nondefault bit order. We can then write portable record-representation clauses. These representation clauses correspond directly to endian-independent layout specifications: Endian-Independent Layout Specification 1 can be written portably either as

```
   for Date_And_Time'Bit_Order use High_Order_First;
   for Date_And_Time_Type use
      record
         Years_Since_1980 at 0 range  0 ..  6;
         Month            at 0 range  7 .. 10;
         Day_Of_Month     at 0 range 11 .. 15;
         Hour             at 2 range  0 ..  4;
         Minute           at 2 range  5 .. 10;
         Seconds          at 2 range 11 .. 15;
      end record;
```

or as

```
   for Date_And_Time'Bit_Order use Low_Order_First;
   for Date_And_Time_Type use
      record
         Years_Since_1980 at 0 range  9 .. 15;
         Month            at 0 range  5 ..  8;
         Day_Of_Month     at 0 range  0 ..  4;
         Hour             at 2 range 11 .. 15;
         Minute           at 2 range  5 .. 10;
         Seconds          at 2 range  0 ..  4;
      end record;
```

Endian-Independent Layout Specification 2 can be written portably either as

```
   for Date_And_Time'Bit_Order use High_Order_First;
   for Date_And_Time_Type use
      record
         Years_Since_1980 at 0 range  0 ..  6;
         Month            at 0 range  7 .. 10;
         Day_Of_Month     at 0 range 11 .. 15;
         Hour             at 0 range 16 .. 20;
         Minute           at 0 range 21 .. 26;
         Seconds          at 0 range 27 .. 31;
      end record;
```

or as

```
   for Date_And_Time'Bit_Order use Low_Order_First;
   for Date_And_Time_Type use
      record
         Years_Since_1980 at 0 range 25 .. 31;
         Month            at 0 range 21 .. 24;
         Day_Of_Month     at 0 range 16 .. 20;
         Hour             at 0 range 11 .. 15;
         Minute           at 0 range  5 .. 10;
         Seconds          at 0 range  0 ..  4;
      end record;
```

In all four record-representation clauses, each distinct byte-offset value corresponds to a distinct machine scalar (i.e., to a line of an endian-independent-layout-specification picture), and the ranges associated with a given byte offset correspond to the position of a field within that machine scalar.