Document Number: WG14 N464/X3J11 95-065

C9X Revision Proposal
=====================

Title: Ordering and Alignment Extensions (OAX)
Author: Frank Farance, William Rugolsky, Jr.
Author Affiliation: Farance Inc.
Postal Address: 555 Main Street, New York, NY, 10044-0150, USA
E-mail Address: frank@farance.com, rugolsky@farance.com
Telephone Number: +1 212 486 4700
Fax Number: +1 212 759 1605
Sponsor: X3J11
Date: 1995-08-25
Proposal Category:
- __ Editorial change/non-normative contribution
- __ Correction
- X_ New feature
- __ Addition to obsolescent feature list
- __ Addition to Future Directions
- __ Other (please specify) ______________________________

Area of Standard Affected:
- __ Environment
- X_ Language
- __ Preprocessor
- __ Library
  - __ Macro/typedef/tag name
  - __ Function
  - __ Header

Prior Art: IEEE 1596.5 standard, BSD networking functions
Target Audience: C programmers, network and database applications
Related Documents (if any): IEEE 1596.5, SBEIR proposal, REP proposal
Proposal Attached: X_ Yes __ No, but what's your interest?
Abstract:

This proposal enhances the features of the SBEIR (Specification-Based Extended Integer Range) proposal by adding storage qualifiers for bit/byte ordering and bit/byte alignment. Ultimately, this proposal should be incorporated into the SBEIR proposal. The proposal is presented separately because the feature is conceptually, semantically, and syntactically separate. The Data Representation Proposal (REP) might be used in conjunction with this proposal. The problem this proposal solves is handling the different ordering and alignment methods for different implementations.

A common misconception is that these features can be provided by the programmer: the programmer manually packs and unpacks data is the desired ordering and alignment. While the programmer can do this manually, the code becomes error-prone and non-obvious. For example, the BSD networking library functions "htonl" (host to network long) and "htons" (host to network short) are provided to convert from the native ordering to the network ordering. Many programmers get this wrong because of conceptual problems with objects they are manipulating: rather than working with values, they are now working with values (native form) and data (twisted values stored in the same kind of container). This is one area where the compiler can help: mechanically generating code that is otherwise error prone to generate. In fact, this is probably one of the primary reasons we use higher level languages rather than assembler.

BIT/BYTE ORDERING

Some implementations use big endian ordering (highest byte first), others use little ending ordering (lowest byte first), while others use mixed ordering. The bit/byte ordering is only a concern when a program must conform to externally defined layouts of data structures, e.g., a data structure intended to be passed over the network or the command register for a hardware device. It is important to note that I/O may be performed in traditional methods (e.g., "fread", "fwrite"), via memory-mapped transfer, or shared-memory among communication systems. In order word, ordering isn't just associated "fread" or "fwrite".

The following storage qualifiers are added to specify ordering:

```
bigend
littleend
```

for specifying big endian and little endian. Although there are other types of orderings, they aren't widely used for data interchange among *heterogeneous* systems. For data interchange, bit orderings are the same as byte orderings, e.g., big endian means bytes are numbered from starting from highest order and bit fields are packed from highest order. These qualifiers only affect how the value is stored, not the value itself. Thus, programmers think only about values (reduces programming errors).

The following examples show the use of these extensions. For the purposes of these examples, assume that "short" is 16 bits, "int" is 32 bits, and "long" is 32 bits. The appropriate SBEIR types would be used otherwise for specifying the precision.

The first example shows the use of a IP address and port number.

```
struct external_rep
{
        bigend long ip_address;
        bigend short port;
} er;

/* Stores IP address "12.34.56.78". */
er.ip_address = (((12L<<8)+34L)<<8+56L)<<8+78L;
```

```
er.port = 13;
write(fd,&er,sizeof er);
```

The second example shows the use of bit ordering to layout the contents of an IP packet.

```
struct ip_packet
{
        bigend int version:4;
        bigend int IHL:4
        bigend int type_of_service:8;
        bigend int total_length:16;
        bigend int identification:16;
        bigend int flags:4;
        bigend int fragment_offset:4;
        bigend int time_to_live:8;
        bigend int protocol:8;
        bigend int header_checksum:16;
        bigend int source_ip_address;
        bigend int destination_ip_address;
        bigend int options:24;
        bigend int padding:8;
};
```

Within a structure all bit orderings must be the same, otherwise the result is unspecified.

BIT/BYTE ALIGNMENT

Alignment concerns the storage alignment of objects and bit fields. Alignment control is used for two purposes: data interchange and performance. For data interchange, a programmer-specified alignment provides a well-known layout regardless of how the implementation normally aligns members of, say, a structure. Other uses for alignment are for performance reasons. An alignment of 1 gives the smallest packing of data. An alignment of 8 (on a 64-bit machine with 8-bit bytes) might give faster access because of word alignment.

The alignment is specified with the storage qualifier:

```
align:N
```

where N is a compile-time constant. If there is no alignment specification, the object is aligned natively, i.e., current alignment rules. An alignment of N (for N>0) means that the object should be aligned in a memory address that is a multiple of N. Although powers of 2 are commonly used for alignment, on implementations that use 3 bytes per word (e.g., a 24-bit implementation), it might be useful to use an alignment of 3 or 6.

For objects within structures, the alignment (i.e., a multiple of N) refers to the byte offset within the structure, not the memory address of that element.

An alignment of 0 is used in structure bit fields: each bit field must be adjacent to the next bit field with no ''holes'' -- even if the bit field is split across ''word'' boundaries.

SUMMARY

This proposal incorporates the following changes:

1. Adding the storage qualifiers "bigend", "littleend", "align:N".

2. Requires the implementation to generate reordering code when reading or writing a value of that has a non-native ordering.

3. Requires the implementation to organize bit fields in a particular order if an ordering is specified.

4. Requires the implementation to align objects if an alignment is specified.

5. Requires the implementation to align structure members if an alignment is specified.

6. For bit fields with an alignment of 0, requires the implementation to pack bit fields even though they may cross word boundaries.