-
- EventHelix.com
- eventstudio *model object and message flows*
- visualether *Wireshark pcap to call flow*
- system design *LTE IMS GSM TCP/IP Embedded OOAD*
- company *contact us support*
- facebook *like us and stay connected*

Share

Byte Alignment and Ordering

Realtime systems consist of multiple processors communicating with each other via messages. For message communication to work correctly, the message formats should be defined unambiguously. In many systems this is achieved simply by defining C/C++ structures to implement the message format. Using C/C++ structures is a simple approach, but it has its own pitfalls. The problem is that different processors/compilers might define the same structure differently, thus causing incompatibility in the interface definition.

There are two reasons for these incompatibilities:

- Byte Alignment Restrictions
- Byte Ordering

Byte Alignment Restrictions

Most 16-bit and 32-bit processors do not allow words and long words to be stored at any offset. For example, the Motorola 68000 does not allow a 16 bit word to be stored at an odd address. Attempting to write a 16 bit number at an odd address results in an exception.

Why Restrict Byte Alignment?

32 bit microprocessors typically organize memory as shown below. Memory is accessed by performing 32 bit bus cycles. 32 bit bus cycles can however be

performed at addresses that are divisible by 4. (32 bit microprocessors do not use the address lines A1 and A0 for addressing memory).

The reasons for not permitting misaligned long word reads and writes are not difficult to see. For example, an aligned long word X would be written as X0, X1, X2 and X3. Thus the microprocessor can read the complete long word in a single bus cycle. If the same microprocessor now attempts to access a long word at address 0x000D, it will have to read bytes Y0, Y1, Y2 and Y3. Notice that this read cannot be performed in a single 32 bit bus cycle. The microprocessor will have to issue two different reads at address 0x100C and 0x1010 to read the complete long word. Thus it takes twice the time to read a misaligned long word.

	Byte 0	Byte 1	Byte 2	Byte 3
0x1000				
0x1004	X0	X1	X2	X3
0x1008				
0x100C		Y0	Y1	Y2
0x1010	Y3			

Compiler Byte Padding

Compilers have to follow the byte alignment restrictions defined by the target microprocessors. This means that compilers have to add pad bytes into user defined structures so that the structure does not violate any restrictions imposed by the target microprocessor.

The compiler padding is illustrated in the following example. Here a char is assumed to be one byte, a short is two bytes and a long is four bytes.

User Defined Structure

```
struct Message
{
  short opcode;
  char subfield;
  long message_length;
  char version;
  short destination_processor;
};
```

Actual Structure Definition Used By the Compiler

```
struct Message
{
  short opcode;
  char subfield;
  char pad1;            // Pad to start the long word at a 4 byte boundary
  long message_length;
  char version;
  char pad2;            // Pad to start a short at a 2 byte boundary
  short destination_processor;
  char pad3[4];        // Pad to align the complete structure to a 16 byte boundary
};
```

In the above example, the compiler has added pad bytes to enforce byte alignment rules of the target processor. If the above message structure was used in a different compiler/microprocessor combination, the pads inserted by that compiler might be different. Thus two applications using the same structure definition header file might be incompatible with each other.

Thus it is a good practice to insert pad bytes explicitly in all C-structures that are shared in a interface between machines differing in either the compiler and/or microprocessor.

General Byte Alignment Rules

The following byte padding rules will generally work with most 32 bit processor. You should consult your compiler and microprocessor manuals to see if you can relax any of these rules.

- Single byte numbers can be aligned at any address
- Two byte numbers should be aligned to a two byte boundary
- Four byte numbers should be aligned to a four byte boundary
- Structures between 1 and 4 bytes of data should be padded so that the total structure is 4 bytes.
- Structures between 5 and 8 bytes of data should be padded so that the total structure is 8 bytes.
- Structures between 9 and 16 bytes of data should be padded so that the total structure is 16 bytes.
- Structures greater than 16 bytes should be padded to 16 byte boundary.

Structure Alignment for Efficiency

Sometimes array indexing efficiency can also determine the pad bytes in the structure. Note that compilers index into arrays by calculating the address of the indexed entry by the multiplying the index with the size of the structure. This number is then added to the array base address to obtain the final address. Since this operation involves a multiply, indexing into arrays can be expensive. The array indexing can be considerably speeded up by just making sure that the structure size is a power of 2. The compiler can then replace the multiply with a simple shift operation.

Byte Ordering

Microprocessors support big-endian and little-endian byte ordering.
Big-endian is an order in which the "big end" (most significant byte) is stored first (at the lowest address). Little-endian is an order in which the "little end" (least significant byte) is stored first.

The table below shows the representation of the hexadecimal number 0x0AC0FFEE on a big-endian and little-endian machine. The contents of memory locations 0x1000 to 0x1003 are shown.

	0x1000	0x1001	0x1002	0x1003
Big-endian	0x0A	0xC0	0xFF	0xEE
Little-endian	0xEE	0xFF	0xC0	0x0A

Why Different Byte Ordering?

This is a difficult question. There is no logical reason why different microprocessor vendors decided to use different ordering schemes. Most of the reasons are historical. For example, Intel processors have traditionally been little-endian. Motorola processors have always been big-endian.

The situation is actually quite similar to that of Lilliputians in Gulliver's Travels. Lilliputians were divided into two groups based on the end from which the egg should be broken. The big-endians preferred to break their eggs from the larger end. The little-endians broke their eggs from the smaller end.

Conversion Routines

Routines to convert between big-endian and little-endian formats are actually quite straight forward. The routines shown below will convert from both ways, i.e. big-endian to little-endian and back.

Big-endian to Little-endian conversion and back

```
short convert_short(short in)
{
  short out;
  char *p_in = (char *) &in;
  char *p_out = (char *) &out;
  p_out[0] = p_in[1];
  p_out[1] = p_in[0];
  return out;
}

long convert_long(long in)
{
  long out;
  char *p_in = (char *) &in;
  char *p_out = (char *) &out;
  p_out[0] = p_in[3];
  p_out[1] = p_in[2];
  p_out[2] = p_in[1];
  p_out[3] = p_in[0];
  return out;
}
```

EventStudio

- call flow gallery
- sequence diagrams
- use cases & more
- testimonials
- download free trial

VisualEther

- Wireshark gallery
- visualize Wireshark
- auto diagnose
- select fields
- download free trial

Telecom+networking

- LTE tutorials and call flows
- Long Term Evolution blog
- telecom call flows
- TCP/IP protocol flows
- TCP/IP Networking blog

Software Design

- object oriented design
- design patterns
- embedded design
- fault handling
- Software Design blog

Follow

- facebook
- twitter
- linkedin
- medium
- google+

Share

Company

- contact us
- blog
