# Chapter 1 Byte alignment concept and big/little endian

## 1.1 Why Byte Alignment Occurred

In modern computers architecture, the memory space is divided by bytes. In theory, it seems that access to any type of variable can start at any address, but in reality, when accessing a specific type of variable, it is often accessed at a specific memory address. This requires various types of data to be arranged in space according to certain rules (see the byte alignment diagram in Chapter 2.2 for details), rather than sequentially discharging one after another, This is alignment. Typical application scenarios are as follows:：

1. The purpose of byte alignment is that the CPU can accelerate the reading and writing of memory data, because if a variable occupies multiple bytes, assuming a compact layout in memory, for 32-bit hardware systems, the basic registers are also 32-bit, so when reading from the address bus, it is also read in 32-bit units. Failure to use this alignment method will result in the variable being read, At least two readings are required to fully concatenate the corresponding values. So byte alignment is a typical subsitute of space for time。

2. Sometimes there is the opposite, using a compact format to store different data types. The main reason is that the internet bandwidth is insufficient in the early morning, and the device memory is expensive. In order to carry as much effective information as possible, a method of exchanging time for space has been adopted.

When we don't specify it specifically, the compiler will be very clever in automatically aligning us (naturally), so it seems that we can also ignore this issue. However, in reality, in network programming, in order to improve the efficiency of encapsulation or de encapsulation, we often access the latter variable or structure through offset, which can lead to incorrect access results without considering byte alignment issues. Sometimes the data structures in our own code are also accessed through offsets, and we still need to pay attention to these details. Including the size when applying for memory, attention should be paid to the actual memory size caused by natural byte alignment, which may trigger overflow operations in memcpy. Of course, if execution efficiency is not considered, in order to facilitate code operations during unpacking and encapsulation, we can also use specified alignment values for forced alignment. Please refer to the "Alignment Principles" chapter for details.

## 1.2 Some issues with non naturally aligned data access

For non naturally aligned data access, processors with different architectures may have different coping strategies：

1. Some will be transparently processed (such as x86 architecture), which software may not feel, but it will affect performance dependency. For example, non aligned data access exists across bytes. The compiler processes non aligned data access operations by copying instructions or shifting children multiple times, which requires more Instruction cycle to complete read and write operations.

2. Some may not report hardware exceptions, but continuing to run may cause problems.

3. Although some may report exceptions, the information provided may not be sufficient to indicate that it was caused by non aligned access.

For 2 and 3, it is often necessary to add program debugging statements or disassemble code for analysis.

The examples in this document are limited to 32-bit Bitwise operation and 32-bit compiler, and the results are shown.

## 1.3 Overview of big and little endian

### 1.3.1 **Basic concepts for the big and little endian**

Le is called little endian and be is called big endian. These are two Endianness, called little endian mode and big endian mode respectively.

Le represents the low bit of data stored in the low bit of memory address, and the high bit of data stored in the high bit of memory address.

Be represents the high bit of data stored in the low bit of memory address, and the low bit of data stored in the high bit of memory address.

Different Endianness is used for different CPUs. For example, PowerPC series CPUs use the big endian mode, while ARM and x86 use the little endian mode

Therefore, for different CPUs, be32_to_cpu, cpu_to_be32, cpu_to_le16,cpu_to_le32, the execution results of these several functions are also different.

However, wherever xx_to_cpu The CPU indicates that the results are intended for use by the CPU. Conversely, the cpu_to_xx means the conversion from the endian of the CPU to the target Endian.

If the CPU itself is in little endian mode, then the CPU_To_le32 will do nothing.

Why is there a distinction between big and little endian modes? This is because in computer systems, we are measured in bytes, and each address unit corresponds to a byte, with each byte being 8 bits. However, in C language, in addition to 8-bit chars, there are also 16-bit short types and 32-bit long types (depending on the specific compiler). In addition, for processors with bits greater than 8, such as 16 or 32-bit processors, there is inevitably a problem of arranging multiple bytes due to register widths greater than one byte. Therefore, this has led to the big endian storage mode and the little endian storage mode. For example, if a 16bit short x has an address of 0x0010 in memory and a value of 0x1122, then 0x11 is the high byte and 0x22 is the low byte. For the big endian mode, 0x11 is placed in the low address, i.e. 0x0010, and 0x22 is placed in the high address, i.e. 0x0011. The actual storage method of x is 0x2211. little endian mode, on the contrary, is still 0x1122. Our commonly used X86 structure is in the little endian mode, while KEIL C51 is in the big endian mode. Many ARM and DSP are in little endian mode. Some ARM processors can also be selected by hardware to choose between big endian mode or little endian mode.

For processors with bits greater than 8, if the width of the register is greater than 1 byte, there will be two methods of storing multiple bytes in the register: large and small storage. There is no distinction between large and small storage, both of which are storage methods

The following are the definitions for the big and little endian in linux/byteorder/generic. h, as well as several commonly used Endianness conversion macros, such as ntohl and htonl. In the network programming stage, because all messages in the network are big endian, our equipment, which may be small endian or big endian, needs to do conversion for the corresponding content in the received message.

```
#if __BYTE_ORDER == __LITTLE_ENDIAN
# define cpu_to_le16(x)      (x)
# define cpu_to_le32(x)      (x)
# define cpu_to_le64(x)      (x)
# define le16_to_cpu(x)      (x)
# define le32_to_cpu(x)      (x)
# define le64_to_cpu(x)      (x)
# define cpu_to_be16(x)      uswap_16(x)
# define cpu_to_be32(x)     uswap_32(x)
# define cpu_to_be64(x)     uswap_64(x)
# define be16_to_cpu(x)     uswap_16(x)
# define be32_to_cpu(x)     uswap_32(x)
# define be64_to_cpu(x)     uswap_64(x)
# define le16_to_cpus(x)
#else
# define cpu_to_le16(x)     uswap_16(x)
# define cpu_to_le32(x)     uswap_32(x)
# define cpu_to_le64(x)     uswap_64(x)
# define le16_to_cpu(x)     uswap_16(x)
# define le32_to_cpu(x)     uswap_32(x)
# define le64_to_cpu(x)     uswap_64(x)
# define cpu_to_be16(x)     (x)
# define cpu_to_be32(x)     (x)
# define cpu_to_be64(x)     (x)
# define be16_to_cpu(x)     (x)
# define be32_to_cpu(x)     (x)
# define be64_to_cpu(x)     (x)

#undef ntohl
#undef ntohs
#undef htonl
#undef htons

#define ___htonl(x) __cpu_to_be32(x)
#define ___htons(x) __cpu_to_be16(x)
#define ___ntohl(x) __be32_to_cpu(x)
#define ___ntohs(x) __be16_to_cpu(x)

#define htonl(x) ___htonl(x)
#define ntohl(x) ___ntohl(x)
#define htons(x) ___htons(x)
#define ntohs(x) ___ntohs(x)
```

### 1.3.2 How to determine the support of the current operating platform for both large and small endian

When writing programs in C language, it is necessary to know whether it is in big endian or little endian.

Example of verifying that the current environment works on big or little endian:

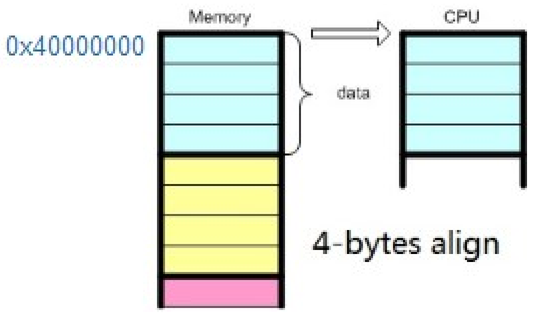
```
//Determine the size of the end
//Assuming high address on the right and low address on the left
int main()
{
    int a = 1; // 0x0000 0001
    //If it is big endian, the low-end byte value put at the high address and the high-end byte at the low address
    //00 00 00 01
    //If it is little endian, Place the high byte value at the high address and the low byte at the low address
    //01 00 00 00
    char* p = (char*)&a; //Character pointer read only 1 byte, read 8 bits
    //If it is a large end storage, the value read by p is 0
    //If it is a small end storage, the value read by p is 1
    if (*p == 1)
        printf("little endian\n");
    else
        printf("big endian\n");
    return 0;
}
```

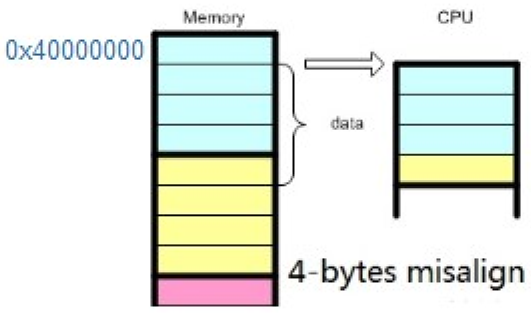
# Chapter 2 Alignment principle

## 1.4 Natural alignment and specified alignment

### 1.4.1 Basic concepts

**Natural Alignment refers to reading N bytes from an address that can be evenly divided by N (addr% N==0). For example, in a system with a 4-byte boundary, reading the 4 bytes starting from address 0x40000000 is aligned, while reading the 4 bytes starting from address 0x40000001 is not aligned.**





When we define a structure, there are many non aligned situations, but why didn't there be any non aligned access exceptions? This is because most compilers can recognize this non aligned situation and automatically add padding. For example, the following structure:

```
struct foo {
        uint8     field1;
        uint16    field2;
        uint8     field3;
};
```

The access to 'field2' in the structure is non aligned, and the compiler will adjust the above structure to the following form during the compilation phase

```
struct foo {
        uint8     field1;
              #padding
        uint16    field2;
        uint8     field3;
              #padding
}
```

Assuming the address of this structure is 0x40000000, then the address of field2 will not be 0x40000001, but 0x4000002, which is at most a waste of storage space. If you use 'sizeof (structure foo)' to look at it, the resulting size will be 6 bytes instead of 4 bytes.

If we manually adjust the order of elements in the structure, like this:

```
struct foo {
    uint16 field2;
    uint8 field1;
    uint8 field3;
};
```

or

```
struct foo {
    uint8 field1;
    uint8 field3;
    uint16 field2;
};
```

So at this point, the access to all elements in the structure is aligned, and the compiler does not need to add padding. Taking a look at 'sizeof ()' again, the resulting size will be 4 bytes.

## 1.4.2 Alignment Rules

1. The alignment value of the data type itself: This value is the length of the basic type on the specified platform. For example, for 32 operating systems and compilers, for char data, its self alignment value is 1, for short data is 2, and for int, float, and double data, its self alignment value is 4, in bytes. The memory address corresponding to the variable to be accessed in the data structure must be divisible by the size of the variable itself.

2. The self alignment value of a structure or class in C++: the value with the highest self alignment value among its members. For example, there are other components of 1, 2, 4, and 8 bytes in structure A, which is the maximum self alignment value of 8 bytes in structure A. Assuming that the size of the entire structure is 24 bytes, the address mapping relationship of this structure in memory is: the starting address of this structure stored in memory must be divisible by 8.

offset = orignal start address-value start address

(1) The first member is at an address offset of 0 from the structural variable.

(2) Other Member variable should be aligned to the address of an integer multiple of a number (alignment number).

Alignment number=the smaller value between the compiler's default alignment number and the size of the member.

(The default value in VS2019 is 8)

(3) The total size of the structure is an integer multiple of the maximum alignment number (each Member variable has an alignment number).

(4)If the nested structure is aligned to an integer multiple of its maximum alignment number, the overall size of the structure is an integer multiple of all maximum alignments (including the alignment number of nested structures).

3. Specify alignment values：

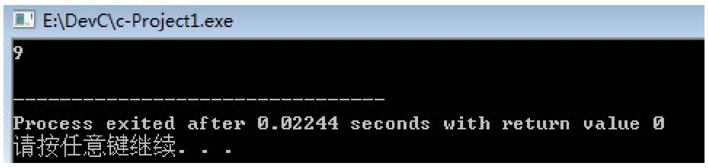
1) #Pragma pack (n), specifying the alignment value n.

End alignment, use #pragma pack(0) or #pragma pack( )

support nesting alignment or not: Supports nested structures, as packs have start and end flags, which are equivalent to being effective for all data structures within the entire interval. For example, if structure A is nested with structure B, and B is defined in structure A, then # pragma pack (n) outside structure A is effective for both A and B.

For example, specifying 1-byte alignment

```
#pragma pack(1)
typedef struct _SYS_MANAGER_1_T
{
    struct _SYS_CONFIG_0_T
    {
    uint8_t     flag;
    uint32_t    data;
    }cfg;
    uint32_t            data;
}SYS_MANAGER_1_T;
#pragma pack()

int main()
{
    printf("%d\r\n",sizeof(SYS_MANAGER_1_T));
    return 0;
}
```



```
E:\DevC\c-Project1.exe
9

--------------------------------
Process exited after 0.02244 seconds with return value 0
请按任意键继续. . .
```

2) #pragma align n

End alignment: use # pragma align 0

support nesting alignment or not: Supports nested structures, reasons can be found in # pragma pack

3) __attribute__((aligned(n))) or __attribute__((packed))。This approach is most commonly used in Linux style code.

if n < min{the MAX aligment value of all members, the Specify alignment value}

ignore the __attribute__

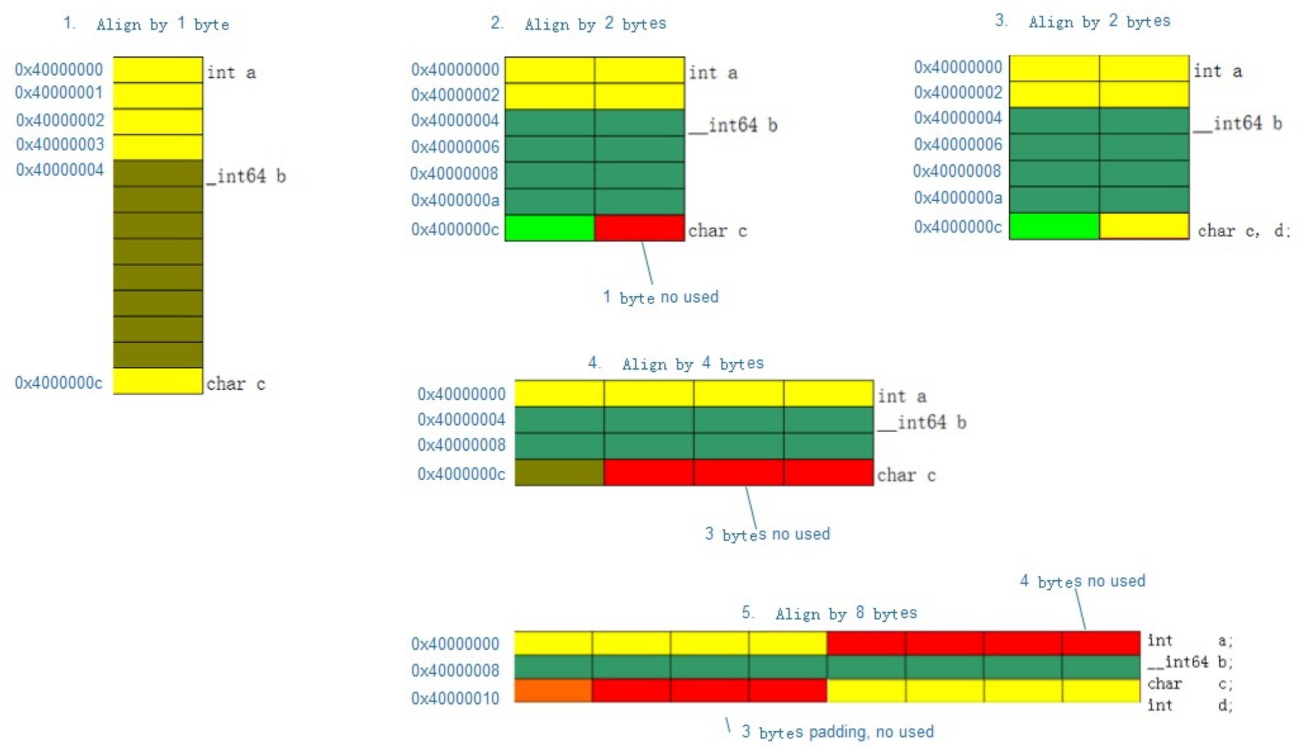
else if n > min {the MAX aligment value of all members, the Specify alignment value}

the structure's members are total aligned with the value "n".

support nesting alignment or not: Supports nested structures, as attributes only take effect on the specified structure itself. If the structure B is nested inside A, the attribute outside A is also not effective for B, so it is necessary to use the attribute to constrain the structure B. Please refer to 2.2.3 of this chapter for details.

4. The effective alignment value of data members, structures, and classes: the smaller value that between the self alignment value and the specified alignment value.

### 1.4.3 Alignment Rule Diagram



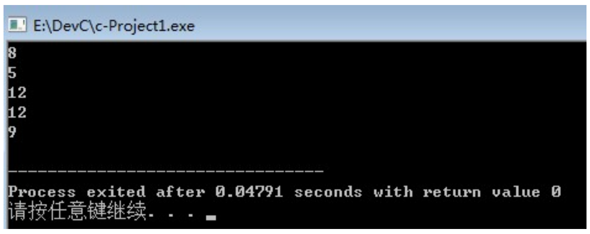
If aligned according to n bytes, the address 0xXXXXXXXX of this variable in memory must be divisible by n(what's more, no remainder).

## 1.5 Nesting rules for specifying alignment values

### 1.5.1 Examples of failed nesting of structures

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

typedef struct _SYS_CONFIG_0_T
{
    uint8_t     flag;
    uint32_t    data;
}SYS_CONFIG_0_T;

#pragma pack(1)
typedef struct _SYS_CONFIG_1_T
{
    uint8_t     flag;
    uint32_t    data;
}SYS_CONFIG_1_T;
#pragma pack()

typedef struct _SYS_MANAGER_0_T
{
    SYS_CONFIG_0_T    cfg;
    uint32_t          data;
}SYS_MANAGER_0_T;

#pragma pack(1)
typedef struct _SYS_MANAGER_1_T
{
    SYS_CONFIG_0_T    cfg;
    uint32_t          data;
}SYS_MANAGER_1_T;
#pragma pack()

#pragma pack(1)
typedef struct _SYS_MANAGER_2_T
{
    SYS_CONFIG_1_T    cfg;
    uint32_t          data;
}SYS_MANAGER_2_T;
#pragma pack()
/* run this program using the console pauser or add your own getch, system("pause") or input loop */

int main(int argc, char *argv[]) {

    printf("%d\r\n",sizeof(SYS_CONFIG_0_T));
    printf("%d\r\n",sizeof(SYS_CONFIG_1_T));
    printf("%d\r\n",sizeof(SYS_MANAGER_0_T));
    printf("%d\r\n",sizeof(SYS_MANAGER_1_T));
    printf("%d\r\n",sizeof(SYS_MANAGER_2_T));
    return 0;
}
```

Program execution results:



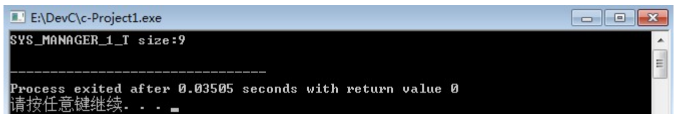
```
8
5
12
12
9

--------------------------------
Process exited after 0.04791 seconds with return value 0
请按任意键继续. . .
```

Analysis: It can be seen that the definition of struct_ SYS_ MANAGER_ 1_ When T, # pragma pack (1) was specified, and it was originally believed that this restriction would also affect the structure SYS contained in the structure_ CONFIG_ 0_ T cfg; Make the same restrictions, and during actual operation, it will be found that SYS_ CONFIG_ 0_ T did not receive the impact of pack (1) here because the structural element SYS_ CONFIG_ 0_ When defining T, pack (1) was not specified, and the compiler will focus on SYS_ CONFIG_ 0_ The definition of T has already determined the natural alignment method. If other structures refer to SYS again_ CONFIG_ 0_ T. The reference section must be processed according to the alignment of the referenced person.

## 1.5.2 Examples of successful nesting of structures

```
#include <stdio.h>
#include <stdint.h>
#pragma pack(1)
typedef struct _SYS_MANAGER_1_T
{
    struct _SYS_CONFIG_0_T
    {
        uint8_t     flag;
        uint32_t    data;
    }cfg;
    uint32_t            data;
}SYS_MANAGER_1_T;
#pragma pack()

int main()
{
    printf("SYS_MANAGER_1_T size:%d\r\n",sizeof(SYS_MANAGER_1_T));
    return 0;
}
```

Program execution results:



```
SYS_MANAGER_1_T size:9

--------------------------------
Process exited after 0.03505 seconds with return value 0
请按任意键继续. . .
```

Analysis: Here we are defining the structure_ SYS_ MANAGER_ 1_ When T, the structure is embedded_ SYS_ CONFIG_ 0_ The complete definition of T, rather than a reference, is used in the structure_ SYS_ MANAGER_

1_ The # pragma pack (1) outside of T remains in effect until # pragma pack (), which constrains it_ SYS_ CONFIG_ 0_ The alignment of components in T.

### 1.5.3 **“attribute” is not supported nesting of structures**

For example, in the following structure, there are consortia and structures within the consortia, which are used in the outer layer__ Attribute__ (aligned (2)) actually does not achieve the effect because it is known that the alignment value of the memory structure is 4 bytes, so in reality, the entire structure is aligned according to 4 bytes, resulting in accessing the assignment of type 0 based on the data offset_ 4_ Byte will have issues because ASN_ 2_ After byte access, the corresponding address pointer only moved two bytes. If the method of directly accessing the component names of the data structure is adopted, there will be no problem.

```
typedef struct vrf_rd_
{
    uint16          rd_type;    /*RD 类型 */
    union{
        struct{
            uint16  asn_2_byte;
            uint32  assign_4_byte;
        }type0;
        struct{
            uint32  ip_address;
            uint16  assign_2_byte;
        }type1;
        struct{
            uint32  asn_4_byte;
            uint16  assign_2_byte;
        }type2;
    }rd_str;
} __attribute__((aligned(2))) vrf_rd_t;
```

Nesting is not supported because attribute only takes effect on the specified structure itself, unlike pack and align, which have a range of start and end. So__ Attribute__ (aligned (2)) vrf_ Rd_ T can only require alignment requirements for the basic data types in this structure; But because of struct vrf_ Rd_ The structure also contains other structures, and the minimum alignment value in other structures is 4 bytes, so aligned (2) is invalid here.

Solution: If the structure B is nested within A, if you want to use the__ Attribute__ (aligned (X)), then the definition of structure B contained in A also needs to be synchronously constrained using attributes.

The definition of this example can be modified as follows:

```
typedef struct vrf_rd_
{
    uint16          rd_type;
    union{
            struct{
                uint16    asn_2_byte;
                uint32    assign_4_byte;
            } __attribute__((packed)) type0;
            struct{
                uint32    ip_address;
                uint16    assign_2_byte;
            } __attribute__((packed)) type1;
            struct{
                uint32    asn_4_byte;
                uint16    assign_2_byte;
            }__attribute__((packed)) type2;
        }rd_str;
    } __attribute__((packed)) vrf_rd_t;
```

## 1.6 Problems that cannot be solved by automatic alignment or reorder

In chapter 2.1.1, the alignment issue was solved by reordering the components of the structure foo, or by adding padding through the compiler's automatic alignment. However, in many cases, both methods cannot be used. Please refer to the analysis below for details.

### 1.6.1 **Examples of structures that cannot be reordered**

Taking accessing the I2C bus as an example, accessing I2C devices sequentially provides the address of the device, the address of the device's internal register, and the value of the register. If encapsulated with the structure "foo", they will correspond to "field1" of "u8", "field2" of "u16", and "field3" of "u8", respectively. If the order of "field1" and "field2" is adjusted, storage space is saved, but the semantics expressed by the structure are not so clear, and the readability of the program becomes worse.

In response to this situation, if the compiler could perform a reorder operation on the elements within the structure during compilation, automatically swapping the order of "field1" and "field2", wouldn't it not only save space but also not affect the readability of the code?

The compiler can indeed easily achieve this. But C language does not allow compilers to do so. The reason for making this seemingly 'self deprecating martial arts' limitation is also reasonable, because there are some structures that express objects that have a' particular 'distribution order in memory. The most typical examples are some structures related to network protocols, because the protocol is agreed upon by both communication parties, and what each byte represents is predetermined. If one party makes a reorder when sending, the other party cannot parse it correctly after receiving it.

### 1.6.2 **a scenario where the compiler cannot perform padding**

In the face of network protocols, not only reorder is not enough, but also the compiler's padding is not enough. Sending data packets with padding added to memory directly can also make the other party "messy". But how does the compiler know that your structure is used for network communication. There's no way, only when defining the structure, add "__attribute__ ((packed))" explicitly.

The meaning of this' attribute 'passed to the compiler is not to add padding, so that the elements in the structure are tightly packed together. In the same example above, after adding this' attribute 'and then using' sizeof () ', the resulting size will be 4 bytes.

```
struct foo {
        uint8     field1;
        uint16 field2;
        uint8     field3;
}__attribute__((packed));
```

However, without padding, it seems non aligned access, won't it trigger hardware exceptions? As a compiler that understands hardware very well, it will confidently tell you: No. Because its family has members targeting different processor architectures, such as the GCC version for ARM, it is well aware of ARM's limitations in non aligned access. Therefore, it will add additional instructions to avoid the occurrence of non aligned access.

## 1.7 Common Exception Triggering Sources for Byte Aligned Access

1、Read data from Flash or received messages and store it directly in a structure.

2、Use the functions of memcpy to directly throw the entire data into the structure.

3、Use a pointer to point after the structure to make a value call.

These issues may all be caused by byte alignment, resulting in issues with obtaining data in the end. Specific cases will be presented in this chapter and subsequent chapters.

## 1.8 Common methods for handling non aligned exceptions

1. By reordering the data structure components, alignment is achieved
2. Access through natural alignment, but it is important to note that the compiler automatically adds padding alignment, resulting in gaps in memory. Special attention should be paid when using pointer offset.
3. Read and write control is performed by directly accessing the name of the Member variable in the structure.
4. By actively adding padding placeholders before and after components that cannot be aligned in the structure, the next component satisfies alignment
5. For non aligned quantities, use segmented copy or shift offset to perform read and write operations.
6. Specify a data structure definition within a certain range through the pre compiled macro # pragma pack (N) or align N, and align it according to N. However, please refer to 2.1.2 for this specific rule. By 'attribute' a data structure, the meaning it passes to the compiler is to not add padding and keep the elements in the structure tightly together.
7. By getting_ Unaligned() or put_ The unaligned() function forcibly reads and writes an expansion of a non aligned address space.

This section supplements cases that are rarely encountered in daily life. The other methods require self reading and self matching of the entire text.

### 1.8.1 **get_unaligned() or put_unaligned()**

example:

```
void func(u8 *data, uint32 value)
{
        ...
        *((uint32 *) data) = cpu_to_le32(value);
        ...
}
```

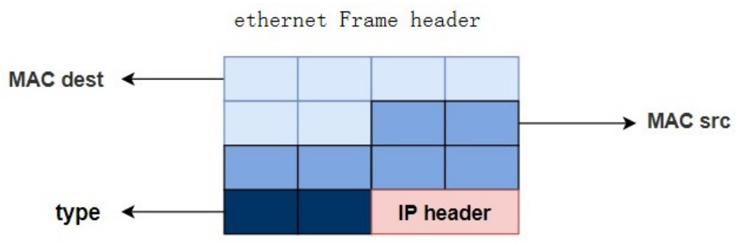
Here is a pointer cast. Assuming that the "data" pointer points to the address 0x40001001, then cast it to a pointer to the "uint32" data type, and the next access will be in 4-byte units, starting with "0x40001001" for read and write operations, which is non aligned.

This scenario has exceeded the "intelligence" range of the compiler, and in order to avoid non aligned access at this time, it is necessary to add "get" to the display_ Unaligned() or put_ Unaligned(), like this:

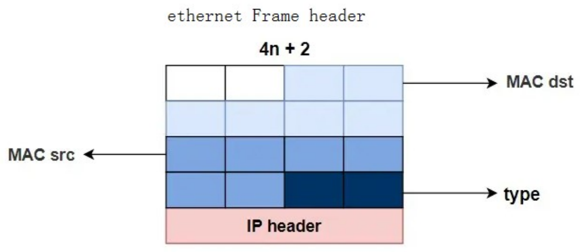
```
value = cpu_to_le32(value);
put_unaligned(value, (u32 *) data);
```

## 1.8.2 Non alignment strategy for direct memory access

The non alignment strategy of direct memory access is used in the transmission of TCP/IP packets, as the MAC header of the message frame is 14 bytes. If no processing is performed during reception, the IP header is not naturally aligned when the software parses it.



One solution is to shift the received target address to the right by 2 bytes, designed as a "4n+2" format：



In Linux OS, its corresponding implementation is to use skb_ Reserve():

SKB_ Reserve (skb, NET_IP_ALIGN);

The default value of 'NET_IP_ALIGN' is 2 (defined in include/Linux/sbuff. h).

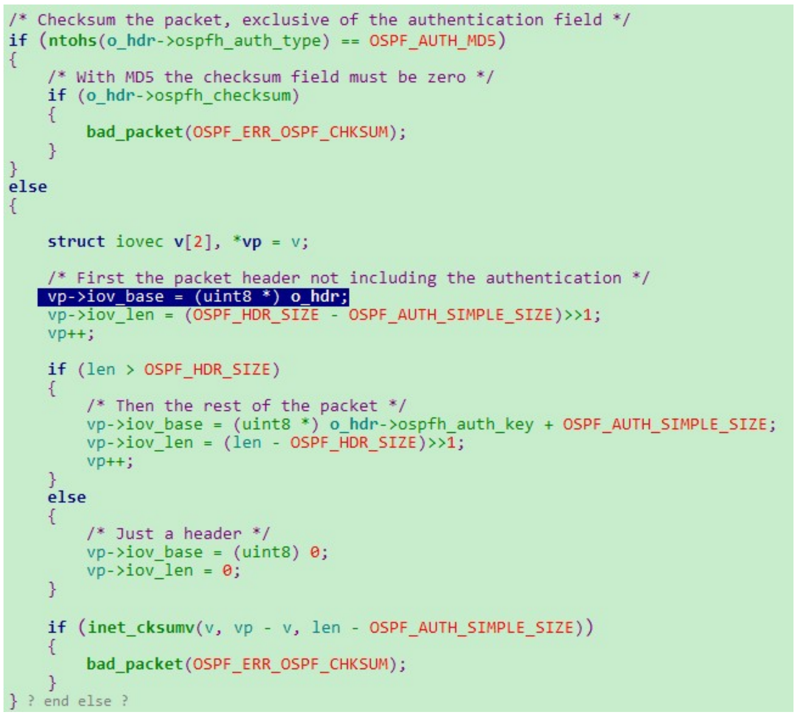
However, due to the fact that network packets are usually received in the form of DMA, this can cause misalignment of DMA addresses. In some architectures, the damage of DMA misalignment may even exceed the performance gain brought about by IP header alignment. The lesser of the two harms, PowerPC and x86 currently set the value of "NET_IP_ALIGN" to 0, which means there is no DMA target address offset.

# Chapter 3 Appendix: Specific Applications and Examples of Common Errors

As mentioned in the first chapter, many times when we parse or encapsulate messages, we define a receiving and sending queue, or simply define an array similar to the following. Before sending, we construct the message (according to the RFC or IEEE document format of each protocol module), and define the corresponding struct structure to fill in the specified array. In this case, we often need to fill in the message by pointer offset. This is also one of the specific applications of using pointers to access defined arrays. In order to save network bandwidth, the specified message format for early rise rfc is quite compact, all aligned according to the method of pack (1); Later on, as network bandwidth gradually became cheaper, many new rfcs also took into account efficiency issues. Instead of strictly following pack (1) to locate message formats, they considered the functions of different message fields based on even byte offsets. If the number of even bytes was not enough, they used char pad or char reserve to occupy the space (as described here, it is not comprehensive).

char i_buf[4096];

char o_buf[4096];

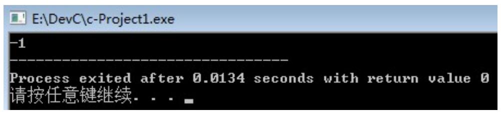


```
/* Checksum the packet, exclusive of the authentication field */
if (ntohs(o_hdr->ospfh_auth_type) == OSPF_AUTH_MD5)
{
    /* With MD5 the checksum field must be zero */
    if (o_hdr->ospfh_checksum)
    {
        bad_packet(OSPF_ERR_OSPF_CHKSUM);
    }
}
else
{

    struct iovec v[2], *vp = v;

    /* First the packet header not including the authentication */
    vp->iov_base = (uint8 *) o_hdr;
    vp->iov_len = (OSPF_HDR_SIZE - OSPF_AUTH_SIMPLE_SIZE)>>1;
    vp++;

    if (len > OSPF_HDR_SIZE)
    {
        /* Then the rest of the packet */
        vp->iov_base = (uint8 *) o_hdr->ospfh_auth_key + OSPF_AUTH_SIMPLE_SIZE;
        vp->iov_len = (len - OSPF_HDR_SIZE)>>1;
        vp++;
    }
    else
    {
        /* Just a header */
        vp->iov_base = (uint8) 0;
        vp->iov_len = 0;
    }

    if (inet_cksumv(v, vp - v, len - OSPF_AUTH_SIMPLE_SIZE))
    {
        bad_packet(OSPF_ERR_OSPF_CHKSUM);
    }
} ? end else ?
```

## 1.9 Access errors for ignoring byte alignment (Case 1)

```
#include <stdio.h>


struct x
{
   char a;
   short b;
};
int main()
{
   struct x z;
   char *c
   memset(&z, 0xFF, sizeof(z));
   c = &(z.a);
   z.a = 1;
   z.b = 2;
   c++;
   printf("%d",*c);
   return 0;
}
```

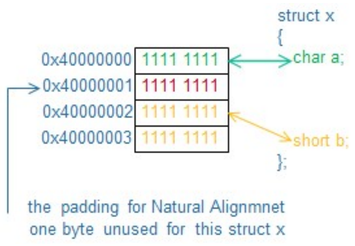
result：



```
E:\DevC\c-Project1.exe
-1
--------------------------------
Process exited after 0.0134 seconds with return value 0
请按任意键继续. . .
```

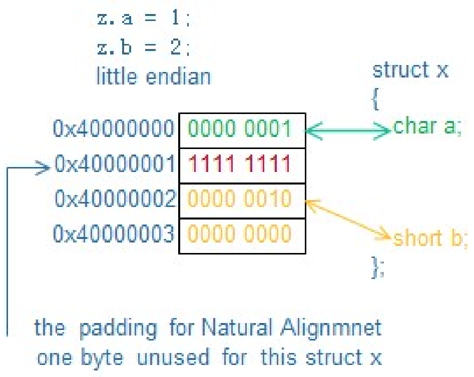
**Analysis:**

1. Initialize memset (&z, 0xFF, sizeof (z)); Afterwards, the memory information layout of the function stack space is as follows



**Note: memset initializes a specified range of memory in its own way, byte by byte at a time.**

2. After assigning values to the structural components, the memory information layout of the function stack space is as follows



3. after c=&(z.a); c points to 0x40000000; Our original intention was for c++to point to the address where z. b is located, in order to access the value of z. b. But because the naturally aligned padding here, c, as a char pointer, actually points to 0x4000001, so * c takes the value corresponding to the address space

4. The output format is% d, which means signed output, so 0xFF, the output is -1. Instead of what we expected 2。

5. In addition, if the memset initialization operation is not performed, the memory corresponding to it in the function stack is likely to be a random value, because the value in the address 0x4000001 corresponding to the

naturally aligned pad added may be a random value. In situations where memory usage is not tight, the value 0 is more possible.

Corrections:

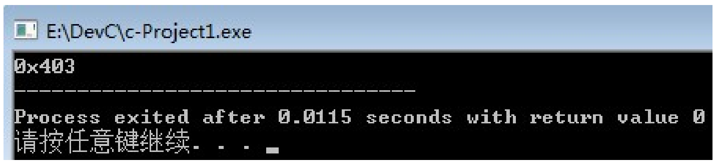
1. uses # pragma pack (1) to impose restrictions on struct x, and b will closely follow a, thus solving the above problem.

2. or using struct components members' name for direct access, read and write operations. The disadvantage is that it is not possible to access all components in the structure through a temporary variable, using pointer offset and data type coercion. And this requirement is often a necessary operation method in network programming, such as encapsulating messages and unpacking. Otherwise, too many types of temporary variables will need to be defined in a unpacking function, resulting in poor code readability and inability to traverse this message through a for loop, resulting in excessive redundancy in the function code.

## 1.10 access errors for ignoring byte alignment (Case 2)

```
#include <stdio.h>
struct x
{
   char a;
   short b;
};
int main()
{
   char c[] = {1,2,3,4,5,6};
   struct x z;
   memcpy(&z, c, sizeof(struct x));
   printf("%x",z.b);
   return 0;
 }
```

result：



```
E:\DevC\c-Project1.exe
0x403
--------------------------------
Process exited after 0.0115 seconds with return value 0
请按任意键继续. . . ▬
```

The expected result of a normal call may be 0x302, but due to char, byte alignment is performed. Therefore, in memory layout, due to the natural alignment of b, b does not directly follow a closely, but rather needs to empty a byte after the address space of a. An invisible abyss is created between a and b, and 2 of them is used to fill in the byte space caused by byte alignment.

Correction methods:

1. Using # pragma pack (1) to impose restrictions on struct x, b will closely follow a, and the above problem has been solved.

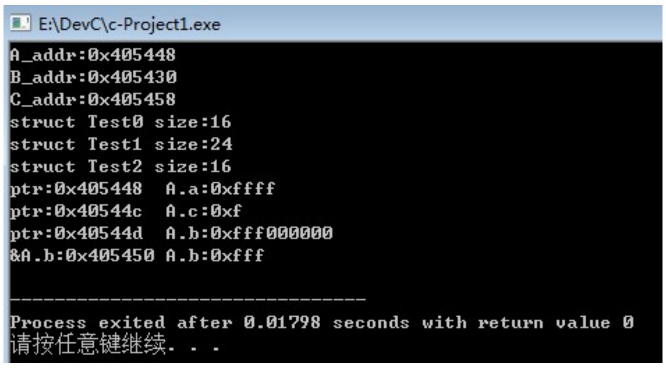
2. Directly perform access, read, and write operations using structural components.

## 1.11 Case Learn of Pointer Access (Complex problem)

```
#include <stdio.h>
#include <stdint.h>

struct Test0
{
    int a;
    char c;
    __int64 b;
}A;

struct Test1
{
    int a;
    __int64 b;
    char c;
}B;

#pragma pack(4)
struct Test2
{
    int a;
    __int64 b;
    char c;
}C;
#pragma pack()

int main()
{
    void *ptr = NULL;
    printf("A_addr:0x%x\r\n", &A);
    printf("B_addr:0x%x\r\n", &B);
    printf("C_addr:0x%x\r\n", &C);

    printf("struct Test0 size:%d\r\n",sizeof(A));
    printf("struct Test1 size:%d\r\n",sizeof(B));
    printf("struct Test2 size:%d\r\n",sizeof(C));

    A.a = 0xFFFF;
```

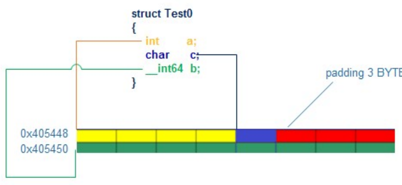
```
        A.c = 0X0F;
        A.b    = 0xFFF;
        ptr = (void *)&A;
        printf("ptr:0x%x    A.a:0x%x\r\n", ptr, *(int *)ptr);

        ptr = ptr + sizeof(int);
        printf("ptr:0x%x    A.c :0x%x\r\n", ptr, *(char *)ptr);

        ptr = ptr + sizeof(char);
        printf("ptr:0x%x    A.b:0x%llx\r\n", ptr, *(__int64 *)ptr);
        printf("&A.b:0x%x A.b:0x%llx\r\n", &A.b, A.b);

        return 0;
}
```

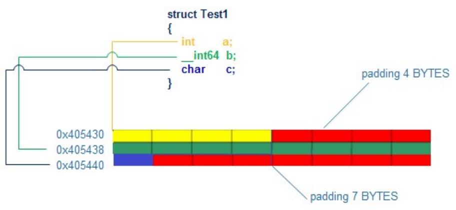
运行结果

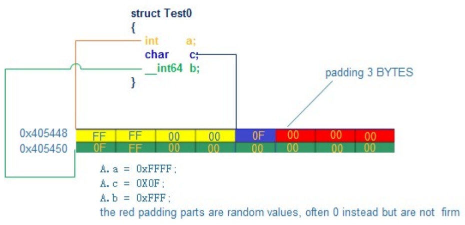


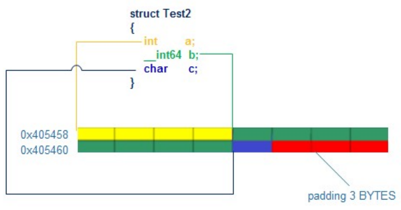
```
E:\DevC\c-Project1.exe
A_addr:0x405448
B_addr:0x405430
C_addr:0x405458
struct Test0 size:16
struct Test1 size:24
struct Test2 size:16
ptr:0x405448   A.a:0xffff
ptr:0x40544c   A.c:0xf
ptr:0x40544d   A.b:0xfff000000
&A.b:0x405450 A.b:0xfff


--------------------------------
Process exited after 0.01798 seconds with return value 0
请按任意键继续. . .
```

This example also involves the knowledge points of Endianness, namely big endian and little endian. The local architecture is X86, so it is little endian, and memory layout analysis is carried out accordingly.









Because printf ("ptr: 0x% x A. b: 0x% llx r n", ptr, * (__int64 *) ptr); In fact, what is printed is the 8 bytes of the starting address calculated from the red filled part after the assignment of Test0. According to the little endian, the high address stores the high bit of the value, so the value printed at this time should be a filled field containing the red part. So the output is 0xFFF00000

And printf ("&A.b: 0x% x A.b: 0x% llx r n",&A.b, A.b); The printed address is 8 bytes starting from 0x405450, so the value is 0xFFF

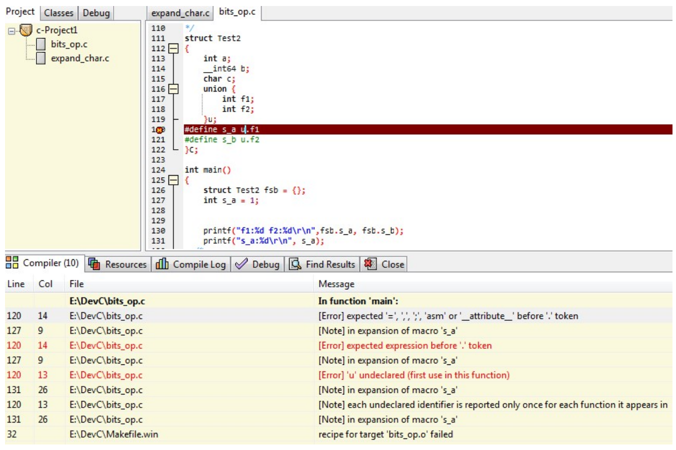
## 1.12 Does the macro definition in the structure have a scope

Originally, this concept did not fall within the scope of this article's discussion, as when discussing byte alignment, the structure was mentioned multiple times, so this content is attached as a reminder.

In C language programming, it is common to define macros in structures. Due to the fact that macros are replaced during preprocessing, the scope of macros defined in structures is still from definition to the end of the file (if there is no # undef). Therefore, defining macros in structures is no different from defining variables outside of structures.

For example, in the following example, there is a macro definition # define s in the structure_ A. u.f1; Causing us to define ints_ When a, due to macro expansion, a situation similar to int u.f1 may occur, and u is undefined, which can lead to compilation errors.

```
struct Test2
{
    int a;
    __int64 b;
    char c;
    union {
        int f1;
        int f2;
    }u;
#define s_a u.f1
#define s_b u.f2
}C;

int main()
{
    struct Test2 fsb = {};
    int s_a = 1;

    printf("f1:%d f2:%d\r\n",fsb.s_a, fsb.s_b);
    printf("s_a:%d\r\n", s_a);
    return 0;
}
```



```
110  */
111  struct Test2
112  {
113      int a;
114      __int64 b;
115      char c;
116      union {
117          int f1;
118          int f2;
119      }u;
120  #define s_a u.f1
121  #define s_b u.f2
122  }C;
123
124  int main()
125  {
126      struct Test2 fsb = {};
127      int s_a = 1;
128
129
130      printf("f1:%d f2:%d\r\n",fsb.s_a, fsb.s_b);
131      printf("s_a:%d\r\n", s_a);
```

| Line | Col | File | Message |
|---|---|---|---|
| | | **E:\DevC\bits_op.c** | **In function 'main':** |
| 120 | 14 | E:\DevC\bits_op.c | [Error] expected '=', ',', ';', 'asm' or '__attribute__' before '.' token |
| 127 | 9 | E:\DevC\bits_op.c | [Note] in expansion of macro 's_a' |
| 120 | 14 | E:\DevC\bits_op.c | [Error] expected expression before '.' token |
| 127 | 9 | E:\DevC\bits_op.c | [Note] in expansion of macro 's_a' |
| 120 | 13 | E:\DevC\bits_op.c | [Error] 'u' undeclared (first use in this function) |
| 131 | 26 | E:\DevC\bits_op.c | [Note] in expansion of macro 's_a' |
| 120 | 13 | E:\DevC\bits_op.c | [Note] each undeclared identifier is reported only once for each function it appears in |
| 131 | 26 | E:\DevC\bits_op.c | [Note] in expansion of macro 's_a' |
| 32 | | E:\DevC\Makefile.win | recipe for target 'bits_op.o' failed |

The advantage of defining a macro in a structure is that it is easy to read: it makes it easier for programmers to understand the logical meaning of the macro;

If compiling with gcc under Linux, you can view the pre compiled file content through: gcc - E *. c;