С. Размещение структур данных в памяти. Доступ к структурам данных

Хохлов С.А.

МГТУ им. Н.Э. Баумана

30 сентября 2025 г.

Выравнивание структур в памяти

#pragma pack / __attribute__((packed))

Назначение: управляют выравниванием (alignment) полей структуры, убирают автоматически вставляемые «пустые» байты.

- #pragma pack(push,1) ... #pragma pack(pop) — директива препроцессора MSVC/GCC.
- __attribute__((packed)) — атрибут GCC/Clang.

Важно

уменьшает размер структуры до «суммы полей», но может привести к медленному доступу на некоторых архитектурах.

Структура с выравниванием

```
1  #include <stddef.h>
2  #include <stdalign.h>
3  #include <stdio.h>
4
5  struct S {
6      char  c;    // 1 байт
7      int   i;    // 4 байта
8      short s;    // 2 байта
9  };
10
11 int main() {
12     printf("sizeof(S)   = %zu\n", sizeof(struct S));
13     printf("alignof(S)  = %zu\n", alignof(struct S));
14     printf("offsetof(i) = %zu\n", offsetof(struct S,i));
15     printf("offsetof(s) = %zu\n", offsetof(struct S,s));
16 }
```

На Cortex-M (32-бит, выравнивание 4) I

Без упаковки

sizeof(S) = 12 // из-за паддинга

alignof(S) = 4

offsetof(i) = 4

offsetof(s) = 8

На Cortex-M (32-бит, выравнивание 4) II

```
1 struct __attribute__((packed)) S2
2 {
3     char c;
4     int i;
5     short s;
6 };
```

С упаковкой

sizeof(S2) = 7

alignof(S2) = 1

offsetof(i) = 1

offsetof(s) = 5

На Cortex-M (32-бит, выравнивание 4) III

Важно

Но с упаковкой доступ к i может быть медленнее или требовать специальных инструкций.

Пример pack/unpack

Битовые поля vs ручные маски I

Битовые поля:

```
1 struct Flags {  
2     uint16_t mode : 3;  
3     uint16_t en   : 1;  
4     uint16_t res  :12;  
5 };
```

- читается легко
- но порядок бит зависит от компилятора/endianness, не всегда переносимо.

Битовые поля vs ручные маски II

Маски:

```
1 uint16_t reg;  
2 // безопаснее для протоколов  
3 reg = (reg & ~0x0007) | (mode & 0x7);
```

- более переносимо, даёт полный контроль.

Упаковка/распаковка с битовыми полями

```
1 #include <stdint.h>
2 #include <string.h>
3
4 // Упакованная структура: 12 бит счетчика, 3 бита флагов, 1 бит паритета.
5 struct __attribute__((packed)) Packet {
6     uint16_t counter : 12;
7     uint16_t flags   : 3;
8     uint16_t parity  : 1;
9 };
10
11 // -- Упаковка --
12 void pack(uint8_t *buf,
13          uint16_t counter, uint8_t flags, uint8_t parity)
14 {
15     struct Packet p = { .counter = counter & 0x0FFF, //здесь маски необязательны
16                       .flags   = flags & 0x07,
17                       .parity  = parity & 0x01 };
18     memcpy(buf, &p, sizeof p); // побайтная копия
19 }
20
21 // -- Распаковка --
22 void unpack(const uint8_t *buf,
23            uint16_t *counter, uint8_t *flags, uint8_t *parity)
24 {
25     struct Packet p;
26     memcpy(&p, buf, sizeof p);
27
28     *counter = p.counter;
29     *flags   = p.flags;
30     *parity  = p.parity;
31 }
```

Упаковка/распаковка с масками

```
1 // поля: 12-бит счетчик, 3-бит флаги, 1-бит паритет
2 void pack(uint8_t *buf, uint16_t counter, uint8_t flags, uint8_t parity)
3     uint16_t word = ((counter & 0x0FFF)      ) |
4                     ((flags  & 0x07) << 12) |
5                     ((parity & 0x01) << 15);
6     buf[0] = word & 0xFF;
7     buf[1] = word >> 8;
8 }
9
10 void unpack(const uint8_t *buf,
11             uint16_t *counter, uint8_t *flags, uint8_t *parity) {
12     uint16_t word = buf[0] | (buf[1] << 8);
13     *counter = word & 0x0FFF;
14     *flags  = (word >> 12) & 0x07;
15     *parity = (word >> 15) & 0x01;
16 }
```

Комментарии

- `__attribute__((packed))` для структуры убирает выравнивание, иначе компилятор может добавить паддинг.
- битовые поля описывают, сколько бит отводится каждому полю — код становится короче и нагляднее.
- порядок битов внутри байтов зависит от компилятора и `endianness`, поэтому битовые поля обычно используют только для работы внутри одной платформы.
- для обмена между разными архитектурами безопаснее вариант с ручными масками и сдвигами.

Преобразования типов

Преобразование типов и aliasing/выравнивание I

Читать объект через несовместимый тип → Undefined Behavior.

Нельзя безопасно делать:

```
1 uint8_t buf[4];  
2 float *f = (float*)buf; // UB  
3 *f = 1.0f;
```

Возможные проблемы с выравниванием

Например, некоторые архитектуры требуют, чтобы `uint32_t` лежал по адресу, кратному 4.

Нарушение → аппаратное исключение или медленный доступ.

```
1 uint8_t data[6];  
2 uint32_t *p = (uint32_t*)&data[1];  
3 // может быть плохо (адрес не кратен 4)
```

- решения: `memcpy`, `__attribute__((aligned(4)))`, выровненные буферы.

Безопасное преобразование типов через memcpy

```
1 float f;  
2 uint8_t buf[4];  
3 memcpy(buf, &f, sizeof f); // безопасно
```

- unsigned char* — единственный тип, который по стандарту может «смотреть» на любые данные.

Безопасное приведение типов через union (объединение)

Позволяет хранить несколько представлений одних и тех же байтов.

Пример с union

```
1 union U {  
2     uint32_t u32;  
3     uint8_t b[4];  
4 };  
5  
6 union U val;  
7 val.u32 = 0x12345678;  
8 printf("%02X %02X %02X %02X\n",  
9     val.b[0], val.b[1],  
10    val.b[2], val.b[3]);
```

Используется для:

- дешёвой «реинтерпретации» без memscan,
- определение endianness, для отладки,
- работы с протоколами.

Размещение в абсолютном адресе + `volatile`

Прямой доступ к периферийному регистру:

```
1 #define GPIOA_ODR (*(volatile uint32_t*)0x40020014)
2
3 void set_pin5(void) {
4     GPIOA_ODR |= (1u << 5); // реально пишет в регистр
5 }
```

- `volatile` гарантирует, что компилятор не закэширует значение и каждый доступ попадёт в память/регистр.

Порядок байтов

Пример демонстрации endianness с union

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  union U32Bytes {
5      uint32_t u32;
6      uint8_t  b[4];
7  };
8
9  int main(void) {
10     union U32Bytes u;
11     u.u32 = 0x12345678;
12
13     printf("u32 = 0x%08X\n", u.u32);
14     printf("bytes: %02X %02X %02X %02X\n",
15           u.b[0], u.b[1], u.b[2], u.b[3]);
16     return 0;
17 }
```

Пример демонстрации endianness с union I

На little-endian (ARM Cortex-M, AVR, x86):

u32 = 0x12345678

bytes: 78 56 34 12

Младший байт хранится по младшему адресу.

На big-endian (IBM 360/370/390, SPARC, Motorola 68000, сетевые протоколы, многие форматы файлов):

u32 = 0x12345678

bytes: 12 34 56 78

Переносимый «переход» между представлениями I

- 1 memcpy — самый безопасный способ прочитать отдельные байты, не меняя их порядок:

```
1 uint32_t value = 0x12345678;
2 uint8_t buf[4];
3 memcpy(buf, &value, sizeof value);
4 // buf[0..3] — байты в машинном порядке
```

- Нулевой байт массива (buf[0]) — это просто первый байт из памяти по наименьшему адресу.
- На little-endian там окажется младший байт (0x78 в примере).
- На big-endian — старший байт (0x12).

Переносимый «переход» между представлениями II

2 Явное формирование big-endian / little-endian:

```
void to_big_endian(uint32_t v, uint8_t out[4]) {  
    out[0] = (v >> 24) & 0xFF;  
    out[1] = (v >> 16) & 0xFF;  
    out[2] = (v >> 8) & 0xFF;  
    out[3] = v & 0xFF;  
}
```

Переносимый «переход» между представлениями III

- Стандартные функции (POSIX / GCC): `htobe32`, `be32toh`, `__builtin_bswap32` — если доступны.

Для кросс-платформенного кода в стиле «сетевой протокол <-> локальная машина» традиционно рекомендуют использовать семейство функций `hton*` / `ntoh*`:

- `htons` / `ntohs` – 16-битное значение
- `htonl` / `ntohl` – 32-битное значение
- (на некоторых системах есть `htonll` / `ntohll` – 64-битное)

Переносимый «переход» между представлениями IV

Преобразование порядка байт

- host to network / network to host.
- «сетевой порядок» (network byte order) – по стандарту big-endian.
- на big-endian машине функции — просто возврат аргумента.
- на little-endian — выполняют перестановку байтов.

Переносимый «переход» между представлениями V

На встраиваемых системах/микроконтроллерах:

- либо используют `__builtin_bswap16/32/64`
- либо пишут явные сдвиги самостоятельно.

`__builtin_bswap16/32/64`

- Компилятор распознаёт вызов как встроенную операцию (не обычную функцию).
- Если целевая архитектура поддерживает специальную инструкцию для byte-swap, он подставит её (ARM Cortex-M).
- Если такой инструкции нет — сгенерирует оптимальную последовательность сдвигов/масок (AVR).

Кратко

- `pack/packed` — уменьшает размер структуры, но снижает скорость/может ломать выравнивание.
- `union` — разные виды доступа к одним байтам, полезно для протоколов.
- `volatile` — «каждый доступ настоящий», нужен для регистров и переменных, изменяемых извне.
- `bitfield` vs маски — `bitfield` удобно читать, маски переносимее.
- `aliasing` — не интерпретируйте память произвольным типом; используйте `memcpu` или `unsigned char*`.
- `alignment` — следите за адресами для многобайтных типов.
- `big/little endian` — `big endian` для передачи и хранения данных, преобразование если архитектура `little endian`