How much memory does a uint8_t occupy? [duplicate]

Ask Question

Asked 2 years, 1 month ago Modified 2 years, 1 month ago Viewed 2k times

-1

This question already has answers here:

Why isn't sizeof for a struct equal to the sum of sizeof of each member? (13 answers)

Closed 2 years ago.

I'm slightly confused as to how much a uint8_t occupies when using the MSVC compiler. Also, I'm somewhat familiar with concept of struct padding such memory is aligned for efficient read/writes. However, my tests show some weird results. I define three structs:

The first features a uint8 and int32. I would expect this to occupy 8 bytes, since the int32 must be word aligned forcing 3 padding bytes to be added. I am correct in my assumption.

The second features a single uint8. I would have expected this to occupy 4 bytes.. but instead it only occupies 1. This kind of confuses me.

The third (and most confusing one) features an int32 followed by a uint8. Using the logic from struct 2 where a lone uint8 occupies a single byte, I would have assumed this struct to occupy 5 bytes. But it occupies 8 bytes. This kind of makes sense, but it doesn't make sense then that struct 2 would only occupy 1 byte.

How much space does uint8 actually occupy?

```
typedef struct StructOne
{
    uint8_t member1;
    int32_t member2;

} StructOne;

typedef struct StructTwo
{
    uint8_t member1;

} StructTwo;

typedef struct StructThree
{
    int32_t member2;
    uint8_t member1;

} StructThree;


int main(int argc, char* args[])
{
    size_t size_struct_one = sizeof(StructOne);
    size_t size_struct_two = sizeof(StructTwo);
    size_t size_struct_three = sizeof(StructThree);

    printf("Size of StructOne = %u\n", sizeof(StructOne));
    printf("Size of StructTwo = %u\n", sizeof(StructTwo));
    printf("Size of StructThree = %u\n", sizeof(StructThree));

    return 0;

}
```

```
Microsoft Visual Studio Debug Console
Size of StructOne = 8
Size of StructTwo = 1
Size of StructThree = 8
```

c struct padding

Share Improve this question Follow edited Jul 10, 2021 at 4:22

asked Jul 10, 2021 at 4:03

Izzo
4,441 ● 13 ● 45 ● 81

3 `uint8_t` is 8 bits, or the same size as `char` which is what sizeof reports. The reason your structs are larger is due to padding to maintain alignment. – Retired Ninja Jul 10, 2021 at 4:24

@RetiredNinja Why isn't a single char padded then? Why are chars only padded when part of a struct? – Izzo Jul 10, 2021 at 4:29

1 The struct is not padded because of the `uint8_t`, it is because of the `uint32_t`. Put as many `char` or `uint8_t` in a struct as you want and there won't be padding. – Retired Ninja Jul 10, 2021 at 4:38

Add a comment

2 Answers

Sorted by: Highest score (default)

2

`uint8_t` is one byte (on implementations that provide it at all, which is most of them).

But there's a rule of `struct` layout that you're missing: the size of a `struct` must be a multiple of its required alignment. Since as you say `int32_t` requires 4-byte alignment, hence so does `struct StructThree`, and so even though its members would fit in 5 bytes, it is padded out to 8.

To see why, imagine you have an array `struct StructThree arr[10];`. It's guaranteed that the elements of this array are placed contiguously, so if `sizeof(struct StructThree)` were only 5, then `arr[1]` would have to start exactly 5 bytes after `arr[0]`, which would break its alignment. (For purposes of ordinary `struct StructThree` pointer arithmetic, it wouldn't matter if they were contiguous or not, but it does matter if you start handling them byte by byte, as with `memcpy` etc.)

Share Improve this answer Follow edited Jul 10, 2021 at 15:06

answered Jul 10, 2021 at 4:44

Nate Eldredge
48.6k ● 6 ● 54 ● 82

1 Nitpick: "at least on typical platforms including yours" hmm... are there any systems where `uint8_t` isn't one byte? I'm not 100% sure but I think the C standard would require it to be. – Support Ukraine Jul 10, 2021 at 6:01

2 @4386427 Indeed, if uint8_t is defined then `sizeof(uint8_t)==1`. This is because if `uint8_t` is defined at all, then each one is addressable. In other words, if uint8_t is defined then `CHAR_BIT <= 8`. But also, `CHAR_BIT>=8` must be true per the standard and so CHAR_BIT must be exactly 8; and so at most byte is required, showing `sizeof(uint8_t) = 1`. – GManNickG Jul 10, 2021 at 6:14

1 That's also what I was thinking. Perhaps: "at least on typical platforms including yours" -> "on all platforms where it exists" – Support Ukraine Jul 10, 2021 at 6:25

Okay, yes, and the one piece I forgot is that `uintN_t` isn't allowed to have any padding bits. I'll edit the answer. – Nate Eldredge Jul 10, 2021 at 15:05

Add a comment

0

Standard Stuff

To add to Nate Eldredge's excellent answer, it's worth recognising that a `struct` is not an entity that is understood by the underlying CPU. It is something that the C language standard defines for the convenience of programmers (and a struct is a very handy way of saying "all these data items belong together"). Compilers have to generate op codes for CPUs to handle structs as defined by the language standard.

Where the need for alignment comes in is that CPUs, generally, can be fussy about how variables are stored in memory. For speed reasons it's not uncommon for a 32bit CPU to require the address of a 32bit integer to be 4 byte aligned, if the integer is to be referenced by the "add" op-code.

However, it's normally possible for bytes to be shifted around inside a computer. So there's nothing at all preventing the compiler generating code that stores the integer at an address not aligned to 4 bytes, and juggles it around to get it into a register before performing an operation with it.

The reason compilers don't do this as a rule is because it's slow. The language standard writers understood that, and so they've defined the behaviour of a struct so as to allow compilers to generate fast code. Generally, most compilers have `#pragma` statements that allow you to tell the compiler to pack things in, rather than spread them out for best possible speed, if that is what is really wanted.

Word Addressing

Ok, so far so good, we're in the bounds of standard knowledge. The thing is, there are computer types where there is no option to pack a struct, where memory was not byte addressed.

Almost all computers today address memory byte by byte, and whilst their microelectronics will load up 4, 8, bytes at a time from memory, their instruction set permits addressing of individual bytes. Go back a few decades, and this was not the case; old Crays, Prime mainframes did not have byte addressing, but word addressing. So the machine gave no option to do "unaligned" stores; there was no concept of a byte having an address in the first place. The minimum allocation on such a machine was 1 word.

So on such a machine, your `Struct2` would be 1 word in size, not one byte, and `sizeof(Struct2)` would return the word length (2, or 4, probably).

C is old enough for this to have been relevant, and hence why it's a matter of discussion in the standards.

Other Languages

Other languages are sufficiently abstract that they don't even let programmers know how data is stored in memory. For example, a `class` in Java or C# will be storing things in memory, but there's nothing (AFAIK) in the language that tells you how this is done, what order the members are in memory, how big they are, or anything. This makes interop between higher level languages and lower level languages such as C/C++ a bit tricky; hence all the marshalling stuff one has to do in C#.

Share Improve this answer Follow

answered Jul 10, 2021 at 5:40

bazza
7,555 ● 15 ● 22

Add a comment

Not the answer you're looking for? Browse other questions tagged c struct padding or ask your own question.