List of platforms supported by the C standard

Asked 13 years, 1 month ago Modified 3 years, 5 months ago

Viewed 1k times

26

Does anyone know of any platforms supported by the C standard, for which there is still active development work, but which are:

- not 2's complement or
- the integer width is not 32 bits or 64 bits or
- some integer types have padding bits or
- if you worked on a 2's complement machine, the bit pattern with sign bit 1 and all value bits zero is not a valid negative number or
- integer conversion from signed to unsigned (and vice versa) is not via verbatim copying of bit patterns or
- right shift of integer is not arithmetic shift or
- the number of value bits in an unsigned type is not the number of value bits in the corresponding signed type + 1 or
- conversion from a wider int type to a smaller type is not by truncation of the left most bits which would not fit

EDIT: Alternatively, if there are platforms in the period 1995 to 1998 which influenced the C99 decision to include the above, but which were discontinued, I would be interested in them also.

EDIT: The C rationale has this to say about padding bits:

Padding bits are user-accessible in an unsigned integer type. For example, suppose a machine uses a pair of 16-bit shorts (each with its own sign bit) to make up a 32-bit int and the sign bit of the lower short is ignored when used in this 32-bit int. Then, as a 32-bit signed int, there is a padding bit (in the middle of the 32 bits) that is ignored in determining the value of the 32-bit signed int. But, if this 32-bit item is treated as a 32-bit unsigned int, then that padding bit is visible to the user's program. The C committee was told that there is a machine that works this way, and that is one reason that padding bits were added to C99.

Footnotes 44 and 45 mention that parity bits might be padding bits. The committee does not know of any machines with user-accessible parity bits within an integer. Therefore, the committee is not aware of any machines that treat parity bits as padding bits.

So another question is, what is that machine which C99 mentioned?

EDIT: It seems that C99 was considering removing support for 1's complement and signed magnitude: http://www.open-

std.org/jtc1/sc22/wg14/www/docs/n868.htm
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n873.htm (search for 6.2.6.2)

c standards

Share

Improve this question

Follow

edited May 21, 2014 at 16:15

nobody
20.1k ● 17 ● 58 ● 79

asked Nov 4, 2011 at 10:24

tyty
839 ● 5 ● 12

Seems like some kind of trivia question. What is it for? Homework? – RedX Nov 4, 2011 at 10:28

2 Also, I suspect that the C committee is (deliberately) over-optimistic about people updating compilers for legacy systems. For example see stackoverflow.com/questions/161797/… -- I doubt that the 1s' complement systems listed there have actively-maintained C99 compilers, but I suspect the C committee wants it to be *possible* even though it's unlikely to actually happen (and hence doesn't meet your stated criteria). – Steve Jessop Nov 4, 2011 at 11:18

2 I never programmed such things, but you should look for the oddities on the side of embedded processors and other special architectures. BTW, you should have listed `CHARBIT != 8` to your list of things that you rarely see in the real world nowadays. – Jens Gustedt Nov 4, 2011 at 14:05

1 @Jens: `CHAR_BIT != 8` is too easy, though. There are real modern DSPs that have over-sized bytes, so it's easy to imagine that the C committee would want C to be applicable there, *without* the compiler writer needing to whip up a representation of a "C pointer" that addresses a smaller unit than the hardware's natural addresses do. – Steve Jessop Nov 4, 2011 at 16:00

2 I think in your wishlist you forgot one special case that makes at least 3, 6 trivial: `_Bool`. It is explicitly mentioned among the unsigned integer types. On many architectures the `CHAR_BIT-1` upper bits are padding bits and conversion to that type is not truncation. But this is probably not what you are after. – Jens Gustedt Nov 7, 2011 at 12:48

8 Answers

Sorted by: Highest score (default)

12

+50

I recently worked at a company which still used a version of the PDP-10, and a port of GCC to that platform. The 10 we used had a few of the attributes you list:

- Integers are not 32 or 64 bits they are 36 bits wide.
- Padding bits are used for some representations. For extended precision integers (e.g. of long long type), the underlying representation was 72-bits in which each of the 36-bit words had a sign-bit.

In addition to the above unusual attributes, there was the issue that the machine had several different byte addressing mechanisms. Bytes with widths in the range of 6-12 bits wide could be addressed by using special bits in the address itself which indicated which width and word

alignment was being used. To represent a char* one could use a representation which would address 8-bit bytes, all of which were left-aligned in the word, leaving 4-bits in each 36-bit word which were not addressed at all. Alternatively 9-bit bytes could be used which would fit evenly into the 36-bit word. Both such approaches had there pitfalls for portability, but at the time I left it was deemed more practical to use the 8-bit bytes because of interaction with TCP/IP networking and standard devices which often think in terms of 16, 24, or 32-bit fields which also have an underlying structure of 8-bit bytes.

As far as I know this platform is still being used in products in the field, and there is a compiler developer at this company keeping relatively recent versions of GCC up to date in order to allow for further C development on this platform.

Share Improve this answer

Follow

answered Nov 13, 2011 at 23:12

bockmabe

350 ● 2 ● 7

12

It should be noted that you cannot rely on undefined behaviour even on commonly used platforms, because modern optimizing compilers perform program transformations that only preserve defined behaviour.

In particular, you cannot rely on two's complement arithmetic giving you `INT_MAX+1 == INT_MIN`. For example, gcc 4.6.0 optimizes the following into an infinite loop:

```c
#include <stdio.h>
int main() {
     int i = 0;
     while (i++ >= 0)
          puts(".");
     return 0;
}
```

EDIT: See here for more on signed overflow and GCC optimizations.

Share Improve this answer

Follow

edited Nov 7, 2011 at 19:28

answered Nov 5, 2011 at 7:11

han

755 ● 4 ● 7

@JonathanLeffler: I originally posted the code without the ++ operator but fixed it pretty much immediately - is your comment based on the old version? The fixed version of code does terminate if you use a compiler that supports two's complement arithmetic (in which `0x7fffffff + 1 == -0x80000000` on a 32-bit machine). – han Nov 5, 2011 at 9:51

OK - the `++` makes my comment inaccurate...I'll remove it. (Yes, I saw the version without the `++`, possibly because I didn't refresh before commenting.) – Jonathan Leffler Nov 6, 2011 at 2:12

@han: Interesting. Do you know any more details about this particular case? – undur_gongor Nov 7, 2011 at 8:29

1 @undur_gongor: There's a good GCC-specific overview here. The GCC option -fwrapv forces 2's complement arithmetic, and on the IBM XL C/C++ compiler the optimization above is disabled by -qstrict_induction. GCC developers also maintain that the Intel compiler does similar loop optimizations, but I haven't found an equivalent command line option for it. – han Nov 7, 2011 at 16:41

1 @han: Thanks a lot, wish I could upvote more than once. You should add the link to your answer. – undur_gongor Nov 7, 2011 at 17:16

12

About a decade ago we had to port our C embedded database to a DSP processor that happened to be the main processor of a car stereo. It was a 24-bit machine, in the worst way: `sizeof(char) == sizeof(int) == sizeof(void*) == 1`, which was 24 bits. We named the branch that dealt with this port "24-bit hell".

Since then we ported our library to many platforms, but none as weird as that. They may still be out there (cheap 24-bit DSP chips are even cheaper now), found in low-cost devices where ease of programming is a distant second to a low bill of materials (BOM). Come to think of it, I think we did encounter a machine where a right shift of an unsigned integer did not necessarily insert zero bits. Even so, highly nonstandard arithmetic rules on a platform guarantee challenging, error-prone porting of software to it, which dramatically increases software development costs. At some point sanity prevails and standards are observed.

I suspect that a lot of the motivation for the presence of these rules in C99 is their presence in C89, and earlier iterations of the language. Don't forget that when C was invented, computers were a lot more diverse than they are today. "Bit-slice" processor designs were available where you could add as many bits as you wanted to your processor just by adding chips. And before C you had to code in assembly language or worry about exactly where in RAM your code would reside, etc.

C was a dramatic step forward in terms of portability, but it had to corral a diverse range of systems, hence the very general rules. 20 years later, when Java came around, it had the benefit of history to allow it to declare up-front how big primitive types were to be, which makes everything a lot easier, as long as Java's choices are sound.

I know you are mostly asking about integers, but I have encountered some weirdness when it comes to pointers. Early Macintosh computers had 32-bit processors (Motorola 68000), but only 24-bit memory busses. Thus 0x00123456 and 0xFF123456 referred to the same memory cell, because the processor chopped off the upper 8 bits when accessing RAM. Apple engineers used these bits to store metadata about the memory that the pointer pointed to. Thus when comparing pointers, one had to mask off the upper bits first. And don't get me started on the segmented memory architectures of the x86. :)

Since we are on this topic, take a look at the MISRA coding standard, which is favored by makers of automobiles that need maximum portability and safety. Also look at *Hacker's Delight* by Henry S. Warren, which has tons of useful bit twiddling tricks in it.

Share Improve this answer Follow

answered Nov 9, 2011 at 7:43

Randall Cook

6,776 ● 6 ● 35 ● 68

7

My two cents. Please don't blame hard, this is from my experience, I'm not a theoretic:

- not 2's complement

All of the existing CPU's are 2's complement

- the integer width is not 32 bits or 64 bits

There are 8 and 16 bits architectures too. 8 bit AVR MCU's is a good example.

- some integer types have padding bits

I am not aware of *any* system, that *pads* integers. Floating numbers - is a different story.

- if you worked on a 2's complement machine, the bit pattern with sign bit 1 and all value bits zero is not a valid negative number
- integer conversion from signed to unsigned (and vice versa) is not via verbatim copying of bit patterns

- right shift of integer is not arithmetic shift
- the number of value bits in an unsigned type is not the number of value bits in the corresponding signed type + 1
- conversion from a wider int type to a smaller type is not by truncation of the left most bits which would not fit

All of the above - not aware of any, and I assume there is no such machine.

Share Improve this answer

Follow

answered Nov 4, 2011 at 10:42

Andrejs Cainikovs

28.4k ● 2 ● 78 ● 96

5

Even if these machines are ancient, there's still an active community programming for PDP-8, most but not all using simulations: PDP-8 as an example. And this machine, AFAIK, uses 12-bit integers!

Share Improve this answer

Follow

answered Nov 5, 2011 at 8:28

Johan Bezem

2,585 ● 1 ● 21 ● 47

3

The cc65 compiler for Commodore C64 seem to have had some update as late as last year.

Share Improve this answer

answered Nov 4, 2011 at 10:38

Follow

Some programmer dude
409k ● 35 ● 412 ● 642

Presumably this counts on the basis of "`int` is not 32 or 64 bits"? Does it have any of the other rarities too?
– Steve Jessop Nov 4, 2011 at 11:00

2 No, it has 16-bit ints (and no float, so not fully ISO-compliant), but is pretty close to other common platforms otherwise. Also, C compilers for DOS had 16-bit integers too. – cyco130 Nov 7, 2011 at 10:57

3

An old adage (I forgot the attribution) says that

> **there is no such thing as portable code**

But only that there are some code which have been ported.

You should not care about writing portable code, you should care about writing code that will be easy to port to other platforms.

Besides, using only the C standard gives you not many useful things. Posix standards gives you much more.

Share Improve this answer

Follow

answered Nov 4, 2011 at 10:43

Basile Starynkevitch
1

I prefer Jim McCarthy's rule: Portability is for canoes. :)
– Randall Cook Nov 9, 2011 at 7:52

0

> the integer width is not 32 bits or 64 bits or

(Almost) all platforms do have some integers which are neither 32 not 64 bit. I think you meant `int` ? If so, yes, there are many. There are many Microcontrollers still in production that have an 8 bit, sometimes 16 bit, core. They use a 16 bit `int`. It does not make a lot of sense for something like an electric pressure cooker to have a 32 bit processor in it. Neither PIC10, PIC12, PIC16 and PIC18 and AVRs are 8 bit and many of them are still in production. And this still holds true today, 10 years after you asked that question.

Please do not assume a `int` is at least 32 bit, use `uint_least32_t`, `int_least32_t`, `int_fast32_t`, `uint_fast32_t`, `uint32_t`, `int32_t`, `long` or `unsigned long` for that. It annoys me when i see code that breaks when `sizeof(int)*CHAR_BIT==16`.

Share Improve this answer Follow

answered Jul 8, 2021 at 12:08

12431234123412341234123

1