

Re: Why in stdint.h have both least and fast integer types?

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2004-12/0118.html

From: James Harris (*no.email.please*)

Date: 12/01/04

Date: Wed, 1 Dec 2004 18:35:19 -0000

"Kevin Bracey" <kevin.bracey@tematic.com> wrote in message news:cc6f86154d.kbracey@tematic.com...

> *In message <79062cd3.0411270240.29d3a4f7@posting.google.com>*

> *groupstudy2001@yahoo.co.uk (GS) wrote:*

>

>> *The stdint.h header definition mentions five integer categories,*

>>

>> *1) exact width, eg., int32_t*

>> *2) at least as wide as, eg., int_least32_t*

>> *3) as fast as possible but at least as wide as, eg., int_fast32_t*

>> *4) integer capable of holding a pointer, intptr_t*

>> *5) widest integer in the implementation, intmax_t*

>>

>> *Is there a valid motivation for having both int_least and int_fast?*

>

> *The point you missed is that the _least types are supposed to be the*

> **smallest* types at least as wide, as opposed to the *fastest*, which are*

> *designated by _fast.*

The **smallest** type as least as wide as 16 is of width 16, no? If it is impossible to support an integer of width 16 (18-bit word, for instance) how does the implementation deal with this standard's int16_t?

> *A typical example might be the ARM, which (until ARMv4) had no 16-bit memory*

> *access instructions, and still has only 32-bit registers and arithmetic*

> *instructions. There int_least16_t would be 16-bit, but int_fast16_t might*

> *be*

> *32-bit.*

>

> *How you decide what's "fastest" is the tricky bit.*

Absolutely! There is no point making a data type "fast" if it is to be repeatedly compared with values which are not the same width. Of course, operations are fast or slow, not data values. Is the standard confusing two

orthogonal issues?

> *In a function, code like:*

>

> `uint16_t a, b, c;`

>

> `a = b + c;`

>

> *would be slow on the ARM, because it would have to perform a 32-bit*

> *addition, and then manually trim the excess high bits off. Using*

> *uint_fast16_t would have avoided that. [*]*

Yes, I think I'm coming round to having one that behaves as if it is exactly 16 bits and another that behaves as if it has at least 16 bits.

> *On the other hand, if you had an array of 2000 such 32-bit int_fast_16_ts*

> *you*

> *were working on, having them as 16-bit might actually be faster because*

> *they*

> *fit in the cache better, regardless of the extra core CPU cycles to*

> *manipulate them.*

>

> *That observation is likely to be true for pretty much any cached*

> *processor*

> *where int_fast_XX != int_least_XX, so as a programmer it's probably going*

> *to*

> *be a good idea to always use int_least_XX for arrays of any significant*

> *size.*

I can see your point here. It's a subtlety. I still wonder, though, if I wouldn't prefer to specify that array as `int16_t`. Specifying `int_least16_t` is making me a hostage to the compiler. If I am taking in to account the architecture of the underlying machine (in this case, primary cache size) wouldn't I be better writing more precise requirements than `int_leastX_t`?