

Re: My idea of fully-portable C code

Source: <http://coding.derkeiler.com/Archive/General/comp.arch.embedded/2008-05/msg00648.html>

- *From:* Hans-Bernhard Bröker <HBBroeker@xxxxxxxxxxxxx>
 - *Date:* Wed, 14 May 2008 01:28:22 +0200
-

Tomás Ó hÉilidhe wrote:

On May 13, 7:59 pm, Hans-Bernhard Bröker <HBBroe...@xxxxxxxxxxxxx> wrote:

* Use three bits per LED. 00 = Off. 01 = Red. 10 = Unused.
11 =
Green.

The reasoning behind that choice of encoding is flawed. Putting an unused state in the middle of a list of values will "use the least amount of memory possible" only by luck, but never by design.

I don't know what you're getting at here, but the reason I chose to use 00 and 11 for valid values was so that I could do `SetAllBitsZero` and `SetAllBitsOne`.

If so, then why did you state a completely different reason for it, before? You said you used that encoding "To use the least amount of memory possible for each LED".

```
void SetChunk(uint_fast16_t const i, uint_fast8_t const
state);
```

Here's your first unwarranted assumption (although it's not one of portability). You've validated your self-stated requirement:

^^^^^^^^

Typo. That was supposed to be "violated".

Re: My idea of fully-portable C code

What assumption? Please be specific.

Was it really so hard to read one more line before interjecting? Here it is:

(Of course you can set the chunk size bigger or smaller than 3 bits)

Using `uint8_fast_t` for the datatype disallows setting the chunk size to more than 8 bits.

No, it doesn't. The chunk size is determined by `BITS_PER_CHUNK`. The type, `uint8_fast_t`, can indeed store values outside the "chunk value range" but that doesn't mean you should try to store 467 in a 2-Bit number.

Aha. So you don't even see the problem. You said that of course, I can set the chunk size bigger than 3 bits. I thus choose to set it to 13 bits. Can you see where there's a problem with passing 1234 via a 8-bit function argument?

In addition to that, 'i' should really be `size_t` or something at least as large as that. As-is, you introduce yet another hardcoded limitation.

I went with `uint_fast16_t` because I thought 65536 bits was enough.

What you thought is irrelevant. You failed to specify that restriction, and that makes the program an incomplete solution to the problem on some architectures.

Of course if you want 4 million bits you can always move forward to 32-Bit.

Why should `_I_` have to do that? You claimed you had written a fully portable program. Why should I have to change it to use it on a bigger problem than you thought it should be used for?

```
#define TOTAL_BITS_NEEDED (QUANTITY_CHUNKS  
* BITS_PER_CHUNK)
```

Re: My idea of fully-portable C code

Bad macro. Parentheses missing around QUANTITY_CHUNKS and BITS_PER_CHUNK leave the code vulnerable to funny definitions of those constants (e.g. #define BITS_PER_CHUNK 2+3) The same problem occurs repeatedly later on.

You're confusing things. A macro is a self-contained, self-sufficient entity. If I have a macro called QUANTITY_CHUNKS, then it should be able to be used on its own without parentheses.

Yet your design leaves it up to the *user* of your code to define it. Users should not be trusted with knowing such important rules and adhering to them at all time.

You're betting the well-being of your code on assumptions that you neither documented, nor strengthened the implementation against.

If this wasn't how things worked then you'd see parentheses EVERYWHERE littered throughout code wherever a macro is used.

Guess what. That's exactly what you *do* see in macros written by experienced programmers.

And it has a portability problem, too! Check out what this does for QUANTITY_CHUNKS=60000u, BITS_PER_CHUNK=6u, on a 16-bit host platform.
You lose. Thanks for playing.

It overflows, you're right. Integer types don't have infinite range in C. I consciously chose uint_fast16_t because I consciously chose a limit of 65536 bits.

It's irrelevant what you were or were not conscious of. Talk is cheap.

The only thing that counts is what the code does if used within the limits you actually specified (basically: none), on a random host platform implementing standard C.

```
#define BITS_ALL_ONES(x) ((1u << (x)) - 1u)
```

Will fail as soon as $\text{pow}(2,x) > (\text{UINT_MAX} + 1)$... (e.g. $x=17$ on a 16-bit platform).

Re: My idea of fully-portable C code

You won't have a 17-Bit chunk. I could use plenty of #error directives throughout the code to make sure it's not misused.

You could do that, sure. But you didn't. You claimed it was full-portable as-is. It's not.

The main usage would be for `BITS_PER_CHUNK <= CHAR_BIT`.

So you say now. No mention of such "main usage" occurs in your OP.

.