When should I use type abstraction in embedded systems

Asked 16 years, 4 months ago Modified 7 years ago Viewed 3k times

22

I've worked on a number of different embedded systems. They have all used `typedef`s (or `#defines`) for types such as `UINT32`.

This is a good technique as it drives home the size of the type to the programmer and makes you more conscious of chances for overflow etc.

But on some systems you know that the compiler and processor won't change for the life of the project.

So what should influence your decision to create and enforce project-specific types?

EDIT I think I managed to lose the gist of my question, and maybe it's really two.

With embedded programming you may need types of specific size for interfaces and also to cope with restricted resources such as RAM. This can't be avoided, but you can choose to use the basic types from the compiler.

For everything else the types have less importance. You need to be careful not to cause overflow and may need to watch out for register and stack usage. Which

may lead you to `UINT16`, `UCHAR`. Using types such as `UCHAR` can add compiler 'fluff' however. Because registers are typically larger, some compilers may add code to force the result into the type.

```
i++;
```

can become

```
ADD REG,1
AND REG, 0xFF
```

which is unecessary.

So I think my question should have been :-

given the constraints of embedded software what is the best policy to set for a project which will have many people working on it - not all of whom will be of the same level of experience.

c **embedded**

Share

Improve this question

Follow

edited Dec 22, 2017 at 10:23

gmuraleekrishna

3,433 ● 1 ● 28 ● 46

asked Aug 10, 2008 at 7:50

itj

1,165 ● 1 ● 11 ● 16

9 Answers

Sorted by: Highest score (default)

18

I use type abstraction very rarely. Here are my arguments, sorted in increasing order of subjectivity:

1. Local variables are different from struct members and arrays in the sense that you want them to fit in a register. On a 32b/64b target, a local `int16_t` can make code slower compared to a local int since the compiler will have to add operations to /force/ overflow according to the semantics of `int16_t`. While C99 defines an `intfast_t` typedef, AFAIK a plain int will fit in a register just as well, and it sure is a shorter name.
2. Organizations which like these typedefs almost invariably end up with several of them (`INT32, int32_t, INT32_T`, ad infinitum). Organizations using built-in types are thus better off, in a way, having just one set of names. I wish people used the typedefs from stdint.h or windows.h or anything existing; and when a target doesn't have that .h file, how hard is it to add one?
3. The typedefs can theoretically aid portability, but I, for one, never gained a thing from them. Is there a useful system you can port from a 32b target to a 16b one? Is there a 16b system that isn't trivial to port to a 32b target? Moreover, if most vars are ints, you'll actually gain something from the 32 bits on the new target, but if they are `int16_t`, you won't. And the places which are hard to port tend to require

manual inspection anyway; before you try a port, you don't know where they are. Now, if someone thinks it's so easy to port things if you have typedefs all over the place - when time comes to port, which happens to few systems, write a script converting all names in the code base. This should work according to the "no manual inspection required" logic, and it postpones the effort to the point in time where it actually gives benefit.

4. Now if portability may be a theoretical benefit of the typedefs, *readability* sure goes down the drain. Just look at stdint.h: `{int,uint}{max,fast,least}{8,16,32,64}_t`. Lots of types. A program has lots of variables; is it really that easy to understand which need to be `int_fast16_t` and which need to be `uint_least32_t`? How many times are we silently converting between them, making them entirely pointless? (I particularly like BOOL/Bool/eBool/boolean/bool/int conversions. Every program written by an orderly organization mandating typedefs is littered with that).

5. Of course in C++ we could make the type system more strict, by wrapping numbers in template class instantiations with overloaded operators and stuff. This means that you'll now get error messages of the form "class Number<int,Least,32> has no operator+ overload for argument of type class Number<unsigned long long,Fast,64>, candidates are..." I don't call this "readability", either. Your chances of implementing these wrapper classes

correctly are microscopic, and most of the time you'll wait for the innumerable template instantiations to compile.

Share Improve this answer

Follow

answered Aug 17, 2008 at 19:15

Yossi Kreinin

246 ● 1 ● 3

One further tweak that I have seen and appreciate is use of FIXED types and "USE BEST" types. i.e. typedef unsigned char UINT8 typedef unsigned uint255; uint 255 specifies the range of the values, but lets the optimum size be specified per system – itj Sep 13, 2008 at 21:15

@itj: Instead of uint255, use `uint_fast8_t` from `stdint.h`. It's defined as a fast type that can support an unsigned 8-bit value. On one platform, that may be an `unsigned char`. On another it may simply be an `unsigned int`. – tomlogic Aug 15, 2010 at 21:17

This is all fine until your code has to work on a very limited processor and you want or have to test on a different processor, perhaps because testing and debugging on the real target is difficult/impossible. In this situation, you need to test on your dev host with variables the same size as on the target, and if you havent coded with size-specific typedefs your process will be completely borken.
– DisappointedByUnaccountableMod Dec 26, 2017 at 20:55

8

The C99 standard has a number of standard sized-integer types. If you can use a compiler that supports C99 (gcc does), you'll find these in `<stdint.h>` and you can just use them in your projects.

Also, it can be especially important in embedded projects to use types as a sort of "safety net" for things like unit conversions. If you can use C++, I understand that there are some "unit" libraries out there that let you work in physical units that are defined by the C++ type system (via templates) that are compiled as operations on the underlying scalar types. For example, these libraries won't let you add a `distance_t` to a `mass_t` because the units don't line up; you'll actually get a compiler error.

Even if you can't work in C++ or another language that lets you write code that way, you can at least use the C type system to help you catch errors like that by eye. (That was actually the original intent of Simonyi's Hungarian notation.) Just because the compiler won't yell at you for adding a `meter_t` to a `gram_t` doesn't mean you shouldn't use types like that. Code reviews will be much more productive at discovering unit errors then.

Share Improve this answer

Follow

answered Aug 10, 2008 at 9:03

Chris Hanson

55k ● 8 ● 74 ● 104

4

My opinion is if you are depending on a minimum/maximum/specific size **don't** just assume that (say) an `unsigned int` is 32 bytes - use `uint32_t` instead (assuming your compiler supports C99).

Share Improve this answer

Follow

answered Aug 11, 2008 at 4:29

Bernard
45.5k ● 18 ● 56 ● 70

4

I like using stdint.h types for defining system APIs specifically because they explicitly say how large items are. Back in the old days of Palm OS, the system APIs were defined using a bunch of wishy-washy types like "Word" and "SWord" that were inherited from very classic Mac OS. They did a cleanup to instead say Int16 and it made the API easier for newcomers to understand, especially with the weird 16-bit pointer issues on that system. When they were designing Palm OS Cobalt, they changed those names again to match stdint.h's names, making it even more clear and reducing the amount of typedefs they had to manage.

Share Improve this answer

Follow

answered Aug 20, 2008 at 14:12

Ben Combee
17.4k ● 6 ● 42 ● 42

> +1 for using types in `stdint.h`. Best way to go for portability. If a platform doesn't have it, it's trivial to create it. – tomlogic Aug 15, 2010 at 21:18

3

I believe that MISRA standards suggest (require?) the use of typedefs.

From a personal perspective, using typedefs leaves no confusion as to the size (in bits / bytes) of certain types. I have seen lead developers attempt both ways of

developing by using standard types e.g. int and using custom types e.g. UINT32.

If the code isn't portable there is little *real* benefit in using typedefs, *however* , if like me then you work on both types of software (portable and fixed environment) then it can be useful to keep a standard and use the cutomised types. At the very least like you say, the programmer is then very much aware of how much memory they are using. Another factor to consider is how 'sure' are you that the code will not be ported to another environment? Ive seen processor specific code have to be translated as a hardware engieer has suddenly had to change a board, this is not a nice situation to be in but due to the custom typedefs it could have been a lot worse!

Share Improve this answer

Follow

answered Aug 24, 2008 at 14:48

TK.

47.8k ● 47 ● 121 ● 148

Yes, it's an advisory rule *(#6.3 of MISRA-C 2004 respectively #13 of MISRA-C '98)*. – ollo Jun 10, 2015 at 16:45

1

Consistency, convenience and readability. "UINT32" is much more readable and writeable than "unsigned long long", which is the equivalent for some systems.

Also, the compiler and processor may be fixed for the life of a project, but the code from that project may find new

life in another project. In this case, having consistent data types is very convenient.

Share Improve this answer Follow

answered Aug 10, 2008 at 8:11

Zooba

11.4k ● 3 ● 38 ● 40

1

If your embedded systems is somehow a ***safety critical system*** (or similar), it's strongly *advised* (if not required) to use typedefs over plain types.

As *TK.* has said before, **MISRA-C** has an (advisory) rule to do so:

> **Rule 6.3 (advisory):** typedefs that indicate size and signedness should be used in place of the basic numerical types.

(from MISRA-C 2004; it's Rule #13 (adv) of MISRA-C 1998)

Same also applies to C++ in this area; eg. JSF C++ coding standards:

> **AV Rule 209** A UniversalTypes file will be created to define all sta ndard types for developers to use. The types include: [uint16, int16, uint32_t etc.]

Share Improve this answer

Follow

answered Jun 10, 2015 at 16:43

ollo

25.3k ● 15 ● 110 ● 157

1

Using `<stdint.h>` makes your code more portable for unit testing on a pc.

It can bite you pretty hard when you have tests for everything but it still breaks on your target system because an `int` is suddenly only 16 bit long.

Share Improve this answer

Follow

answered Jun 20, 2016 at 10:07

Pelle

1,252 ● 14 ● 19

> Yes this is one of the most pragmatic demonstrations of when using explicit sized types makes a lot of sense. Of course, if you don't/won't ever do PC-hosted testing then why would you care? – DisappointedByUnaccountableMod Dec 24, 2017 at 14:42

0

Maybe I'm weird, but I use ub, ui, ul, sb, si, and sl for my integer types. Perhaps the "i" for 16 bits seems a bit dated, but I like the look of ui/si better than uw/sw.

Share Improve this answer

Follow

answered Aug 14, 2010 at 3:23

supercat

80.8k ● 9 ● 174 ● 220

I guess this dpeends on context. For embedded programming, size is very important so i & w are effectively "don't care" values. – itj Aug 16, 2010 at 10:01

@itj: I'm not sure quite what you mean by that. I use my two-character type identifiers because they're short and visually clean and distinct. I can't think of any other 2-character identifiers I use for any purpose that start with 's' or 'u', so it seems pretty obvious what the types mean (except, possibly, for 'ui' or 'si' in isolation). – supercat Aug 16, 2010 at 15:42