

Re: "Portability" constructs like UINT32 etc.

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2005-10/msg00521.html

- *From:* Jack Klein <jackklein@xxxxxxxxxxx>
 - *Date:* Thu, 06 Oct 2005 23:33:25 -0500
-

On 05 Oct 2005 14:14:03 +0100, John Devereux
<jdREMOVE@xxxxxxxxxxxxxxxxxxxx> wrote in comp.lang.c:

```
> "Sunil" <sunil.goutham@xxxxxxxxxx> writes:
>
>> Hi all,
>>
>> I am using gcc compiler in linux.I compiled a small program
>> int main()
>> {
>> printf("char : %d\n",sizeof(char));
>> printf("unsigned char : %d\n",sizeof(unsigned char));
>> printf("short : %d\n",sizeof(short));
>
> <SNIP>
>
> This brings to mind something that I have wondered about.
>
> I often see advice elsewhere, and in other peoples programs,
> suggesting hiding all C "fundamental" types behind typedefs such as
>
> typedef char CHAR;
> typedef int INT32;
> typedef unsigned int UINT32;
```

The first one is useless, the second two are worse than useless, they are dangerous, because on another machine int might have only 16 bits and INT32 might need to be a signed long.

```
> typedef char* PCHAR;
```

This is more dangerous yes, never typedef a pointer this way. At least not if the pointer will ever be dereferenced using that alias.

```
> The theory is that application code which always uses these typedefs
> will be more likely to run on multiple systems (provided the typedefs
> are changed of course).
```

More than theory, very real fact.

Re: "Portability" constructs like UINT32 etc.

> I used to do this. Then I found out that C99 defined things like
> "uint32_t", so I started using these versions instead. But after
> following this group for a while I now find even these ugly and don't
> use them unless unavoidable.

Nobody says you have to care about portability if you don't want to. That's between you, your bosses, and your users. If you are writing a program for your own use, the only one you ever have to answer to is yourself.

On the other hand, both UNIXy and Windows platforms are having the same problems with the transition from 32 to 64 bits that they had moving from 16 to 32 bits, if perhaps not quite so extreme.

For more than a decade, the natural integer type and native machine word on Windows has been called a DWORD, and on 64 bit Windows the native machine word is going to be a QWORD.

> What do people here think is best?

On one embedded project we had CAN communications between the main processor, a 32-bit ARM, and slave processors that were 16/32 bit DSPs.

The only types that were identical between the two were signed and unsigned short, and signed and unsigned long. In fact, here are the different integer types for the two platforms:

'plain' char unsigned 8-bit signed 16-bit
signed char signed 8-bit signed 16-bit
unsigned char unsigned 8-bit unsigned 16-bit
signed short signed 16-bit signed 16-bit
unsigned short unsigned 16-bit signed 16-bit
signed int signed 32-bit signed 16-bit
unsigned int unsigned 32-bit unsigned 16-bit
signed long signed 32-bit signed 32-bit
unsigned long unsigned 32-bit unsigned 32-bit

Both processors had hardware alignment requirements. The 32-bit processor can only access 16-bit data at an even address and 32-bit data on an address divisible by four. The penalty for misaligned access is a hardware trap. The DSP only addresses memory in 16-bit words, so there is no misalignment possible for anything but long, and they had to be aligned on an even address (32-bit alignment). The penalty for misaligned access is just wrong data (read), or overwriting the wrong addresses (write).

Now the drivers for the CAN controller hardware are completely off-topic here, but the end result on both systems is two 32-bit words in memory containing the 0 to 8 octets (0 to 64 bits) of packet data.

Re: "Portability" constructs like UINT32 etc.

Re: "Portability" constructs like UINT32 etc.

These octets can represent any quantity of 8-bit, signed or unsigned 16-bit, or 32-bit data values that can fit in 64 bits, and have any alignment.

So your mission, Mr. Phelps, if you decide to accept it, is to write code that will run on both processors despite their different character sizes and alignment requirements, that can use a format specifier to parse 1 to 8 octets into the proper types with the proper values.

The code I wrote runs on both processors with no modifications. And I couldn't even use 'uint8_t', since the DSP doesn't have an 8-bit type. I used 'uint_least8_t' instead.

As for the C99 choice of type definitions like 'uint8_t' and so on, they are not the best I have ever seen, but they are also far from the worst. And they have the advantage of being in a C standard, so with a little luck they will eventually edge out all the others.

--

Jack Klein

Home: <http://JK-Technology.Com>

FAQs for

comp.lang.c <http://www.eskimo.com/~scs/C-faq/top.html>

comp.lang.c++ <http://www.parashift.com/c++-faq-lite/>

alt.comp.lang.learn.c-c++

<http://www.contrib.andrew.cmu.edu/~ajo/docs/FAQ-acllc.html>

.

• **References:**

- ◆ ***On what does size of data types depend?***

◇ From: Sunil

- ◆ ***"Portability" constructs like UINT32 etc.***

◇ From: John Devereux

- Prev by Date: ***Re: References for machines where NULL is not zero.***

- Next by Date: ***Re: if and for statements***

- Previous by thread: ***Re: "Portability" constructs like UINT32 etc.***

- Next by thread: ***Re: "Portability" constructs like UINT32 etc.***

- Index(es):

- ◆ ***Date***

- ◆ ***Thread***