Is there a portable way to define INT_MAX?

Microsoft Azure
Hadoop spoken here.
Java, Hive, Pig, LINQ, .NET...
Hadoop on Azure speaks your language.
Microsoft
Get Azure
AdChoices

I found the following definitions in /usr/include/limits.h:

```
#  define INT_MIN    (-INT_MAX - 1)

#  define INT_MAX    2147483647
```

Also, it seems that all XXX_MAX's in this header file are explicitly defined from a numerical constant.

I wonder if there is a portable way (against different word sizes across platforms) to define a INT_MAX ?

I tried the following:

```
~((int)-1)
```

But this seems incorrect.

A short explanation is also highly regarded.

c gcc int

edited Jul 23 '13 at 2:58
alex
212k 93 531 698

asked Jul 23 '13 at 2:57
Hatrick
55 10

1 `~((int)-1)` is probably zero. – Carl Norum Jul 23 '13 at 3:20

Oh, yes! You are right. But I can't understand why. Could you please explain it a little bit? Thanks a lot! – Hatrick Jul 23 '13 at 5:29

1 The bit pattern for `-1` is all ones in a two's complement system. The one's complement (`~`) of an all-ones word is an all-zeroes word. – Carl Norum Jul 23 '13 at 13:13

Awesome! Thank you! @CarlNorum – Hatrick Oct 21 '14 at 7:10

4 Answers

For the `INT_MAX` in the standard header `limits.h`, the implementor's hands are tied by the fact that it's required to be usable in preprocessor `#if` directives. This rules out anything involving `sizeof` or casts.

If you just want a version that works in actual C expressions, perhaps this would work:

```
(int)-1U/2 == (int)(-1U/2) ? (int)-1U : (int)(-1U/2)
```

The concept here is that `int` may have the same number of value bits as `unsigned`, or one fewer value bit; the C standard allows either. In order to test which it is, check the result of the conversion `(int)-1U`. If `-1U` fits in `int`, its value must be unchanged by the cast, so the equality will be true. If `-1U` does not fit in `int`, then the cast results in an implementation-defined result of type `int`. No matter what the value is, though, the equality will be false merely by the range of possible values.

Note that, technically, the conversion to `int` could result in an implementation-defined signal being raised, rather than an implementation-defined value being obtained, but this is not going to happen when you're dealing with a constant expression which will be evaluated at compile-time.

edited Jul 23 '13 at 5:24

answered Jul 23 '13 at 5:16
R..
106k 11 141 335

1 I don't see that the C standard rules out the unsigned type having more than 1 more value bit than the corresponding signed type, in which case both `(int)-1U/2` and `(int)(-1U/2)` could result in an implementation-defined value, possibly the same one. – caf Jul 23 '13 at 5:59

The 'preprocessor' part is really impressive and helpful, so is the expression. Thanks very much! – Hatrick Jul 23 '13 at 6:21

@caf: Indeed. The test could be fixed by replacing the 2 with a larger value such as `-1U/2+1` , but I don't then have a good approach for the else expression. Of course you could chain a bunch of these up to some sane upper bound on the number of bits you'd expect to see... – R.. Jul 23 '13 at 6:31

1 Also to mention that the preprocessor computes all integer values in `[u]intmax_t` such that any tricks like the expression given here wouldn't work either, because the constants that are used then are not of the right type. – Jens Gustedt Jul 23 '13 at 6:38

If we assume 1 or 2's compliment notation and 8 bit/byte and no padding:

```
#define INT_MAX  ((1 << (sizeof(int)*8 - 2)) - 1 + (1 << (sizeof(int)*8 - 2)))
```

I do not see any overflow in the shifts nor additions. Neither do I see UB. I suppose one could use `CHAR_BIT` instead of 8.

In 1 and 2's compliment the max int would be `power(2,(sizeof(int)*Bits_per_byte - 1) - 1` . Instead of power, we'll use shift, but we can't shift *too* much at once. So we form `power(2, (sizeof(int)*Bits_per_byte - 1)` by doing half of it twice. But overflow is a no-no, so subtract 1 before adding the 2nd half rather then at the end. I've used lots of `()` to emphasize evaluation order.

As pointed out by @caf, this method *fails* is there are padding bits - uncommon but possible.

Computing INT_MIN is a little trickier to make work in 2's *and* 1's compliment, but a similar approach would work, but that is another question.

edited Jul 23 '13 at 4:13 answered Jul 23 '13 at 3:54
chux
22.2k 4 16 44

1 Even if you use `CHAR_BIT` this doesn't take into account the possibility of padding bits, which are uncommon but allowed. – caf Jul 23 '13 at 4:00

@caf I sure you are correct. Do you know of a method or define to determine the number of padding bits? – chux Jul 23 '13 at 4:09

Likely the potential count & locations of the uncommon padding bits theoretically preclude making a general INT_MAX formula. – chux Jul 23 '13 at 4:16

I don't think it is possible without risking undefined behaviour. – caf Jul 23 '13 at 4:18

1 @chux: Padded int can be detected at compile time. `struct { signed int foo:(sizeof(int) *CHAR_BIT-1); };` is a constraint violation if `int` has any padding bits. – R.. Jul 23 '13 at 7:14

You can do that:

```
#define INT_MAX (int) (-1U >> 1)
```

answered Jul 23 '13 at 5:25
Amadeus
2,384 5 16

Well, one can try

```
#define INT_MAX (int) ((unsigned) -1 / 2)
```

which "should" work across platforms with different word size and even with different representations of signed integer values. `(unsigned) -1` will portably produce the `UINT_MAX` value, which is the all-bits-one pattern. Divided it by 2 it should become the expected max value for the corresponding signed integer type, which spends on bit for representing the sign.

But why? The standard header files and definitions made in them are not supposed to be portable.

BTW, the definition of `INT_MIN` you quoted above is not portable. It is specific to 2's complement representation of signed integers.

edited Jul 23 '13 at 5:31 answered Jul 23 '13 at 3:14
AnT
162k 16 226 440

1 It's allowed for the sign bit in the signed type to be a padding bit in the corresponding unsigned type: `int` and `unsigned` may have an equal number of value bits, and in this case `UINT_MAX` and `INT_MAX` will be equal. – caf Jul 23 '13 at 3:53