ISO/IEC JTC 1/SC 22/WG14 **N 2303**

October 12, 2018 **v 1**

intmax_t, a way out

Ease the definition of extended integer types

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The specifications of types [u]**intmax_t** and extended integer types lack to provide the extensibility feature for which they are designed. As a consequence existing "64 bit" implementations are not able to add standard conforming interfaces to their 128 bit or 256 bit integer types without breaking ABI compatibility.
—
A previous version of this proposal has been discussed in message SC22WG14.15569 and the depending thread on the WG14 reflector.

1. PROBLEM DESCRIPTION

The interaction between the definition of extended integer types and [u]**intmax_t** has resulted in a lack of extensibility for existing ABI. Platforms that fixed their specifications for the basic integer types and for [u]**intmax_t** cannot add an extended integer type that is wider than their current [u]**intmax_t** to their specification. As the current text of the C standard stands, such an addition would force a redefintion of [u]**intmax_t** to the wider types. This would have the following consequences:

— The parts of the C library that use [u]**intmax_t** (specific functions but also **printf** and friends) must be rewritten or recompiled with the new ABI and become binary incompatible with existing programs.
— Programs compiled with the new ABI would be binary incompatible on platforms that have not been upgraded.
— The preprocessor of the implementation must be re-engineered to comply to the standard. In particular, there would occur severe specification problems for preprocessor numbers and their evaluation. *E.g* the *value* of **ULLONG_MAX**+1 is not expressible as a literal in the language proper but would be for the preprocessor. The *expression* **ULLONG_MAX**+1 would evaluate to **true** in a preprocessor conditional but to 0 (**false**) in later compilation phases.

As a consequence of these difficulties the concept of "extended integer type" is merely unused by implementations. I have not heard of any implementation that uses this concept. So as it stands this idea of "extended integer type" is basically a failure, nobody uses it, and **intmax_t** is usually just **long** or **long long**.
This has lead to a sensible backlog for platforms such as gcc or clang that provide emulated 128 bit integer types (__[u]int128_t) on 64 bit platforms. They are not able to provide them as "extended integer types" in the sense of the C standard. More and more processor platforms even provide rudimentary support of 128 or 256 bit integers in hardware (*e.g* Intel's AVX vector unit), so it would really be productive to give more slack to implementations to integrate these types into existing ABI.
Generally, we should not block implementations that are able to provide exact-width integer types for $N > 64$. These types can for example be used efficiently for bitsets, UUIDs, cryptography, checksums or networking (ipv6). They have well-defined standard interfaces in the form of ([u]**int**N**_t**) with easy to use feature tests.
Also, current restrictions make it impossible to add bignum integer types as proper integer types to implementations, because their value range can never be covered by a type with a fixed bit representation.

2. SUGGESTED CHANGE

I suggest to change the specification of [u]intmax_t such that they are only at least as wide as any integer type *used* by the standard. Thereby the greatest-width types do not have to cover *all* integer types, in particular not extended integer types that might be added later to an ABI.
The change to the standard can be isolated in 7.20.1.15:

7.20.1.5 Greatest-width integer types

The following type designates a signed integer type capable of representing any value of any required signed integer type:

```
        intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any required unsigned integer type:

```
        uintmax_t
```

These types are required.

Note 1: These types are intended to provide a fallback for applications in situations where they deal with integers for which there are no special provisions for **printf**, **scanf** or similar functions. In particular, they are intended to represent values of all basic integer types as well as **char16_t**, **char32_t**, **ptrdiff_t**, **sig_atomic_t**, **size_t**, **wchar_t**, **wint_t**, the minum-width and fasted integer types for $N = 8, 16, 32, 64$, and, provided they exist, **intptr_t**, **uintptr_t** and exact-width integer types for $N = 8, 16, 32, 64$.

Note 2: It follows from the definitions that greatest-width integer types are at least 64 bit wide.

Note 3: Extended integer types that are not referred by the above list and that have values not representable by **long long** or **unsigned long long**, respectively, need not be representable by the greated-width types.

Recommended practice: Unless some **typedef** in the library clause enforces otherwise, it is recommended to resolve these types to **long** or **long long** and the corresponding **unsigned** counterpart. It is recommended that the same set of integer literals is consistently accepted by all compilation phases, even if greatest-width types are chosen that are wider than **long long**.

3. IMPACT

3.1. Existing implemenations and code

With such a change of the C standard, no existing ABI would have to change, and the preprocessor support for integer expressions could remain unchanged.
Since the concept of extended integer types is basically not yet used by implementations, there would also be no impact on the existing code base on existing implementations, even if they chose to extend their ABI by some wider integer types.

3.2. Extensibility of ABI's

This change allows platforms to add specifications of extended integer types more easily. In particular 128 or 256 bit types can be added to 64 bit ABI as long as a conforming naming

scheme is chosen. Many implementations do so already in various forms and with non-uniform syntax. With this change they could just **`typedef`** their extented type to `uint128_t`, say, and provide the corresponding macros `UINT128_MAX`, `UINT128_C`, `PRId128` etc.
There is no need to extend the language to describe additional integer types (such as **`long long long`**), to add new number literals (`-1ULLL`) or to add **`printf`** conversion characters for these in the C standard. The use of implementation specific names (`__int128` or `__int128_t`) and implementation specific format specifiers (`"%Q"`) is largely sufficient if appropriately mapped by `<stdint.h>` **`typedef`** and macros.
This change can also be seen as a first step such that a bignum type can be added as a new integer type to any implementation that sees need for it. A second would the be to relax constraints about the representation of extended integer types, or to introduce a third category (besides basic and extended) of "*unbounded integer types*", or so.