Proposal for C2x
WG14 n2465

Title:	intmax_t, a way forward
Author, affiliation:	Robert C. Seacord, NCC Group
Date:	2020-02-10
Proposal category:	Defect
Target audience:	Implementers supporting extended integer types

Abstract: Add extended integer types without breaking the ABI

Prior art: C

intmax_t, a way forward

Reply-to: Robert C. Seacord (rcseacord@gmail.com)
Document No: **n2465**
Reference Document: N2425
Date: 2020-02-10

The specifications of types [u]**intmax_t** and extended integer types fail to provide the extensibility feature for which they are designed. As a consequence, existing "64 bit" obsolesinteger types without breaking ABI compatibility.

This paper deprecates the use of [u]**intmax_t** as a misleading and useless type and introduces

- **intbasicmax_t** and **uintbasicmax_t** as integer types that are at least as wide as any basic integer type. These types replace the use of [u]**intmax_t** in APIs and elsewhere.
- **intwidest_t** and **uintwidest_t** as greatest-width integer types that will change width when larger extended integer types are added to the language. These types should not be used in any APIs

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 have anchored 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 redefinition 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 related functions) 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 with the standard. In particular, there would be 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.

Because of these difficulties, the concept of extended integer type is unused by implementations. Consequently, the concept of extended integer type has failed as no implementation uses it under its original design, and **intmax_t** is typically defined as **long** or **long long** even when the implementation supports extended integer types.

This has led 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 be productive to allow 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 be used efficiently for bitsets, UUIDs, cryptography, checksums, networking (ipv6) and so forth. They have well-defined standard interfaces in the form of ([u]**int***N***_t**) with easy to use feature tests.

2. SUGGESTED CHANGES

2.1. New example and recommend practice for greatest-width integers

This proposal introduces the **`intwidest_t`** and **`utinwidest_t`** as greatest-width integer types. These types will change width when larger extended integer types are added to the language. Consequently, these types should not be used in any APIs as they could subsequently lead to ABI breakage.

7.20.1.5 Greatest-width integer types

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

```
intwidest_t
```

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

```
uintwidest_t
```

These types are required.

2 The types **`intwidest_t`** and **`uintwidest_t`** designate a signed and an unsigned integer type capable of representing any value of any signed or unsigned integer type, respectively. These types are required.

3 NOTE 1 The **`intwidest_t`** and **`uintwidest_t`** types are intended to provide a fallback for applications that use integers for which they lack specific type information. This mainly occurs in two different situations. First, for integers that appear in conditional inclusion: (#if expressions, 6.10.1) they provide fallback types that capture the implementation specific capabilities during translation phase 4. Second, for some semantic type definitions that resolve to implementation specific types there are no special provisions for **printf**, **scanf**, or similar functions.

4 **Example** An implementation that has historically fixed its greatest width types to a 64 bit type, and seeks to add a 128 bit integer exact-width type to its extended integer types, may do so by providing types **`uint128_t`**, **`uint_least128_t`**, **`uint_least128_t`** and the corresponding signed types and macros of stdint.h and inttypes.h (7.8.1). Implementations would need to adjust **`intwidest_t`** and **`uintwidest_t`** accordingly which would break binary compatibility with application APIs which use these types.

Application code can then query the type and print it by using the appropriate macros

```
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>
#ifdef UINT128_MAX // ok, because #ifdef
typedef uint128_t bitset;
#else
typedef uint_least64_t bitset;
#endif
int main(void) {
  bitset all = -1;
  printf("the  largest  set  is  %#"PRIXMAX "\n", (uintwidest_t) all);
}
```

Recommended practice

5 It is recommended that the same set of integer literals is consistently accepted by all compilation phases even if **`intwidest_t`** is chosen to be wider than **`signed long long int`**. Implementations and applications should not use the types **`intwidest_t`** and **`uintwidest_t`** to describe application programmable interfaces.[285)]

2.2. New type aliases for the widest type pair

The **intbasicmax_t** and **uintbasicmax_t are** introduced by this proposal as integer types that are at least as wide as any basic integer type in section 7.20.1.6. These types replace the use of [u]**intmax_t** in APIs and elsewhere. The [u]**intmax_t** are misleading and useless types that have been deprecated and are no longer mentioned by the standard.

7.20.1.6 Greatest basic-width integer types

1 The types **intbasicmax_t** and **uintbasicmax_t** designate a signed and an unsigned integer type, respectively, that are at least as wide as any basic integer type, the types

char16_t	**int_least64_t**	**size_t**	**wchar_t**
char32_t	**ptrdiff_t**	**uint_fast64_t**	**wint_t**
int_fast64_t	**sig_atomic_t**	**uint_least64_t**	

and, provided they exist, **intptr_t** and **uintptr_t**. These types are required.

2 **Note 1** The **`intbasicmax_t`** and **`uintbasicmax_t`** types are intended to represent values of all types listed above and also the exact-width integer types for all $N \leq 64$.

3 **Note 2** Extended integer types that are not referred by the above list and that are wider than **signed long long int** may also be wider than **`intbasicmax_t`**. The **`intbasicmax_t`** and **`intwidest_t`** types may then be different.

4 **Note 3** The **`intwidest_t`** and **`uintwidest_t`** types are intended to provide a fallback for applications that deal with extended integer types that are potentially wider than **`intbasicmax_t`** or **`uintbasicmax_t`**.

Recommended practice

5 Unless some **typedef** in the library clause otherwise enforces, it is recommended to resolve **`intbasicmax_t`** to **long** or **long long int** and **`uintbasicmax_t`** to 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 int**.

2.3. Tighten the rules for least and fast minimum-width integer types

To be fully operational, some macros (**_MAX** etc) must be added to the text for <stdint.h>. These additions are straightforward and can be seen in the Appendix

2.4. Eliminating [u]intmax_t from standard interfaces

To avoid such situations where implementations get stuck because of early ABI choices, this proposal replaces the use of [u]**intmax_t** in all interfaces that use these types with the use of [u]**intbasicmax_t** typedefs.

This works because **intmax_t** is currently defined as either **long int** or **long long int** in existing implementations. Recommended practice is to typedef [u]**intbasicmax_t** to these same real types as to not alter the ABI. This leaves implementations free to add support for extended integer types and change the widths of [u]**intwidest_t** to accommodate these changes.

These interfaces in the C standard are

imaxabs	**strtoumax**	**compoundn**	fromfp	ufromfp
imaxdiv	**wcstoimax**	**pown**	fromfpx	ufromfpx
strtoimax	**wcstoumax**	**rootn**		

Only the first six appear already in a published version of the standard. Because [u]**intmax_t** has never been used in the field with types other than **long int** or **long long int**, these functions are all code duplication and the **long long int** interface could clearly have worked all along.

This proposal does not include the inclusion of type-generic functions but does nothing to prevent them from being proposed in the future.

2.5. Formatted input/output functions

Another goal of this proposal is to preserve developer's ability to perform formatted I/O with programmer-defined integer types by converting signed programmer-defined integer types and unsigned programmer-defined integer types to the appropriately signed greatest width type. This is a useful feature and common idiom on which substantial existing code depends.

The `j` length modifier will be used to represent the size of the `[u]intbasicmax_t` types going forward to retain the same size as existing implementations. This again means that no existing code needs to change.

This code can be rewritten to can be rewritten to replace `[u]intmax_t` with `[u]intbasicmax_t` at the developer's leisure.

```
uint_least64_t bitset all = -1;
printf("the  largest  set  is  %ju\n", (uintbasicmax_t) all);
```

Greatest-width types can be used in formatted input/output functions by using the format specifiers defined in 7.8.1. For example:

```
uint128_t bitset all = -1;
printf("the  largest  set  is  %#"PRIXWIDESTMAX"\n", (uintwidest_t) all);
```

The PRI and SCN macros can specify the exact width of the type by using specific bit-width conversion specifiers in a manner that can be understood by both the implementation and the library. If the library doesn't support the specified extended type, the formatted input or output function can return an error.

This is accomplished using a length modified. It uses a lowercase letter because uppercase letters are reserved for implementation extensions, and avoiding the letters used in the standard and various TRs leaves bqvwy. In this case, we decided to use 'w' might be better than 'b' in that it would leave 'b' available for supporting binary output in future. 128-bit integers, for example, will look like this:

```
uint128_t bitset all = -1;
printf("the largest set is %w128d\n", all);
```

A w followed by a decimal number following d, i, o, u, x, or X conversion specifier specifies that the conversion specifier applies to an exact-width integer type argument of exactly *N* bits; or that a following n conversion specifier applies to a pointer to an exact-width integer type argument of exactly *N* bits.

There is some relevant implementation experience. Microsoft printf has I32 and I64 for this purpose. The use of I is in the space reserved for implementation extensions and other implementations use I for other things, but it's still relevant experience should we wish to support w<width> for that purpose. Microsoft also uses 'w' in extensions, but only with string and character formats so that wouldn't conflict in any way with a standard use of 'w'.

3. IMPACT

3.1. Existing code

Existing code that invokes a standard library function that use a **`[u]intmax_t`** type will continue to work unchanged as the interfaces that use these types will continue to use the same real underlying types.

3.1. Existing implementations

With such a change of the C standard, no existing ABIs would have to change, and the preprocessor support for integer expressions could remain unchanged. Because 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. Existing 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 nonuniform syntax. With this change they could **typedef** their extended 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.

3.3 C++ Compatibility

C++ is that they mangle type in their external symbols. However, **`[u]intmax_t`** are still typedefs in C++ and mangle the same as their underlying types.

3.3 Fundamental Type for N-bit integers

This proposal could potentially have some interactions with *N2472 Adding Fundamental Type for N-bit integers*. Supporting these types as extended integer types as proposed would likely mean that the greatest width types would need to be as wide as these types. We're currently in the process of adding a parameterized family of integer types of any bit-width up to our implementation limit (currently 16M).

4.0 Acknowledgements

I would like to recognize the following people for their help with this work: Jens Gustedt, Joseph S. Myers, Aaron Ballman, Richard Smith, Erich Keane, Jeff Dileo, and David Keaton.

5.0 References

N2425 intmax_t, a way out v.2.
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2425.pdf

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

3 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

4 Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see the following URL www.iso.org/iso/foreword.html.

6 This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

7 This fifth edition cancels and replaces the fourth edition, ISO/IEC 98992018. Major changes from the previous edition include

— remove obsolete sign representations and integer width constraints

added the **intwidest_t** and **uwidestmax_t** greatest-width integer types and deprecated the use of intmax_t and uintmax_t

— added the **intbasicmax_t** and **uintbasicmax_t** and allow extended integer types to be wider than **these types**

— added a one-argument version of **_Static_assert** —

harmonization with ISO/IEC 9945 (POSIX)

- extended month name formats for **strftime**
- • integration of functions **memccpy**, **strdup**, **strndup** — harmonization with floating point

standard IEC 60559

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of decimal floating-point technical specification TS 18661-4a

— the macro **DECIMAL_DIG** is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added **nodiscard**, **maybe_unused** and **deprecated** attributes 8A

complete change history can be found in Annex M.

7.8 Format conversion of integer types <inttypes.h>

1 The header <inttypes.h> includes the header <stdint.h> and extends it with additional facilities provided by hosted implementations.

2 It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the type

```
imaxdiv_t
```

which is a structure type that is the type of the value returned by the **imaxdiv** function. For each type declared in <stdint.h>, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[1)]

Forward references integer types <stdint.h> (7.20), formatted input/output functions (7.21.6), formatted wide character input/output functions (7.29.2).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of **PRI** (character string literals for the **fprintf** and **fwprintf** family) or **SCN** (character string literals for the **fscanf** and **fwscanf** family),[2)] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.20.1. In these names, *N* represents the width of the type as described in 7.20.1. For example, **PRIdFAST32** can be used in a format string to print the value of an integer of type **int_fast32_t**.

2 The fprintf macros for signed integers are:

PRIdN PRIdLEASTN PRIdFASTN PRIdMAX PRIdWIDESTMAX:PRIdPTR
PRIiN PRIiLEASTN PRIiFASTN PRIiMAX PRIiWIDESTMAX:PRIiPTR

3 The fprintf macros for unsigned integers are:

PRIoN PRIoLEASTN PRIoFASTN PRIoMAX PRIoWIDESTMAX:PRIoPTR
PRIuN PRIuLEASTN PRIuFASTN PRIuMAX PRIuWIDESTMAX:PRIuPTR
PRIxN PRIxLEASTN PRIxFASTN PRIxMAX PRIxWIDESTMAX:PRIxPTR
PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXWIDESTMAX:PRIXPTR

4 The fscanf macros for signed integers are:

SCNdN SCNdLEASTN SCNdFASTN SCNdMAX SCNdWIDESTMAX:SCNdPTR
SCNiN SCNiLEASTN SCNiFASTN SCNiMAX SCNiWIDESTMAX:SCNiPTR

5 The fscanf macros for unsigned integers are:

SCNoN SCNoLEASTN SCNoFASTN SCNoMAX SCNoWIDESTMAX:SCNoPTR
SCNuN SCNuLEASTN SCNuFASTN SCNuMAX SCNuWIDESTMAX:SCNuPTR
SCNxN SCNxLEASTN SCNxFASTN SCNxMAX SCNxWIDESTMAX:SCNxPTR

1) See "future library directions" (7.31.6).

2) Separate macros are given for use with **fprintf** and **fscanf** functions because, in the general case, different format specifiers might be required for **fprintf** and **fscanf**, even when the type is the same.

6 For each type that the implementation provides in <stdint.h>, the corresponding **fprintf** macros shall be defined and the corresponding **fscanf** macros shall be defined unless the implementation does not have a suitable **fscanf** length modifier for the type.

7 **EXAMPLE**

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
  uintbasicmax_t i = UINTBASICMAX_MAX;        // this type always exists
  wprintf(L"The largest integer value is %020" PRIxMAX "\n", i);
  wprintf(L"The largest extended integer value is %#" PRIxMAX "\n", i);
  uintwidest_t j = UINTWIDEST_MAX;    // this type always exists
  wprintf(L"The largest extended integer value is %#" PRIxWIDESTMAX "\n", j);
  return 0;
}
```

7.8.2 Functions for greatest-width integer types

7.8.2.1 The imaxabs function

Synopsis

1
```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
intbasicmax_t imaxabs(intbasicmax_t j);
```

Description

2 The **imaxabs** function computes the absolute value of an integer j. If the result cannot be represented, the behavior is undefined.[233)]

Returns

3 The **imaxabs** function returns the absolute value.

7.8.2.2 The imaxdiv function

Synopsis

1
```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
imaxdiv_t imaxdiv(intbasicmax_t numer, intbasicmax_t denom);
```

Description

2 The **imaxdiv** function computes numer / denom and numer % denom in a single operation.

Returns

3 The **imaxdiv** function returns a structure of type **imaxdiv_t** comprising both the quotient and the remainder. The structure shall contain (in either order) the members quot (the quotient) and rem (the remainder), each of which has type **intbasicmax_t**. If either part of the result cannot be represented, the behavior is undefined.

7.8.2.3 The strtoimax and strtoumax functions Synopsis

1
```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
intbasicmax_t strtoimax(const char *restrictnptr,  char ** restrict endptr, int base);
uintbasicmax_t strtoumax(const char *restrictnptr,  char ** restrict endptr, int base);
```

---

233) The absolute value of the most negative number may not be representable.

2 The **strtoimax** and **strtoumax** functions are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions, except that the initial portion of the string is converted to **intbasicmax_t** and **uintbasicmax_t** representation, respectively.

Returns

3 The **strtoimax** and **strtoumax** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, ~~**INTMAX_MAX**~~ **INTBASICMAX_MAX**, ~~**INTMAX_MIN**~~ **INTBASICMAX_MIN**, or ~~**UINTMAX_MAX**~~ **UINTBASICMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.7).

7.8.2.4 The wcstoimax and wcstoumax functions Synopsis

1

```
#include <stddef.h>                              // for wchar_t
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
intbasicmax_t wcstoimax(const wchar_t *restrictnptr, wchar_t **restrict endptr, int base);
uintbasicmax_t wcstoumax(const wchar_t *restrictnptr,  wchar_t **restrict endptr, int base);
```

Description

2 The **wcstoimax** and **wcstoumax** functions are equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions except that the initial portion of the wide string is converted to **intbasicmax_t** and **uintbasicmax_t** representation, respectively.

Returns

3 The **wcstoimax** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, ~~**INTMAX_MAX**~~ **INTBASICMAX_MAX**, ~~**INTMAX_MIN**~~ **INTBASICMAX_MIN**, or ~~**UINTMAX_MAX**~~ **UINTBASICMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions (7.29.4.1.3).

7.20 Integer types <stdint.h>

1 The header <stdint.h> declares sets of integer types having specified widths, and defines corresponding sets of macros.[3] It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

2 Types are defined in the following categories

— integer types having certain exact widths;

— integer types having at least certain specified widths;

— fastest integer types having at least certain specified widths;

— integer types wide enough to hold pointers to objects; —

integer types having greatest width.

(Some of these types may denote the same type.)

3 Corresponding macros specify limits of the declared types and construct suitable constants. 4 For each type described herein that the implementation provides,[4] <stdint.h> shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, <stdint.h> shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as "required", but need not provide any of the others (described as "optional").

5 The feature test macro **__STDC_VERSION_STDINT_H__** expands to the token *yyyymm*L.

7.20.1 Integer types

1 When typedef names differing only in the absence or presence of the initial u are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

2 In the following descriptions, the symbol *N* represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

7.20.1.1 Exact-width integer types

1 The typedef name **int*N*_t** designates a signed integer type with width *N* and no padding bits. Thus, **int8_t** denotes such a signed integer type with a width of exactly 8 bits.

2 The typedef name **uint*N*_t** designates an unsigned integer type with width *N* and no padding bits. Thus, uint24_t denotes such an unsigned integer type with a width of exactly 24 bits.

3 These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, and no padding bits, it shall define the corresponding typedef names.

7.20.1.2 Minimum-width integer types

1 The typedef name **int_least*N*_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width. Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits.

2 The typedef name **uint_least*N*_t** designates an unsigned integer type with a width of at least *N*, such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.

---

[3] See "future library directions" (7.31.12).

[4] Some of these types might denote implementation-defined extended integer types.

3 If either of the types **int_least** N**_t** or **uint_least**N**_t** are provided, the other is provided, too, and they are the corresponding signed and unsigned types of each other. If the types **int_least**N**_t** and **int_least**M**_t** are provided for $N < M$, the width of the former is less than or equal to the width of the latter.

4 The following types are required

int_least8_t int_least16_t
int_least32_t int_least64_t

uint_least8_t
uint_least16_t
uint_least32_t
uint_least64_t

as are the types **int_least**N**_t** and **uint_least**N**_t** for all N for which the exact-width types **int**N**_t** and **uint**N**_t** are provided. All other types of this form are optional.

7.20.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[5)] to operate with among all integer types that have at least the specified width.

2 The typedef name **int_fast**N**_t** designates the fastest signed integer type with a width of at least N. The typedef name **uint_fast**N**_t** designates the fastest unsigned integer type with a width of at least N.

3 If either of the types **int_fast**N**_t** or **uint_fast**N**_t** are provided, the other is provided, too, and they are the corresponding signed and unsigned types of each other. If the types **int_fast**N**_t** and **int_fast**M**_t** are provided for $N < M$, the width of the former is less than or equal to the width of the latter.

4 The following types are required

int_fast8_t int_fast16_t
int_fast32_t int_fast64_t

uint_fast8_t
uint_fast16_t
uint_fast32_t
uint_fast64_t

as are the types **int_fast**N**_t** and **uint_fast**N**_t** for all N for which the exact-width types **int**N**_t** and **uint**N**_t** are provided. All other types of this form are optional.

7.20.1.4 Integer types capable of holding object pointers

1 The following type designates a signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer

```
uintptr_t
```

These types are optional.

7.20.1.5 Greatest-width integer types

[5)] The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

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

```
~~intmax_t~~

intwidest_t
```

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

```
~~uintmax_t~~

uintwidest_t
```

These types are required.

7 The types **intwidest_t** and **uintwidest_t** designate a signed and an unsigned integer type capable of representing any value of any unsigned integer type: respectively that are at least as wide as any integer type defined by the header **<stdint.h>**. These types are required.

8 NOTE 1 The **intwidest_t** and **uintwidest_t** types are intended to provide a fallback for applications that use integers for which they lack specific type information. This mainly occurs in two different situations. First, for integers that appear in conditional inclusion: (#if expressions, 6.10.1) they provide fallback types that capture the implementation specific capabilities during translation phase 4. Second, for some semantic type definitions that resolve to implementation specific types there are no special provisions for **printf**, **scanf**, or similar functions.

9 **Example** An implementation that has historically fixed its type **intwidest_t** and **uintwidest_t** to a 64 bit type, and seeks to add a 128 bit integer exact-width type to its extended integer types, may do so by providing types **uint128_t**, **uint_least128_t**, **uint_least128_t** and the corresponding signed types and macros of stdint.h and inttypes.h (7.8.1). Implementations would need to adjust **intwidest_t** and **uintwidest_t** accordingly which would break binary compatibility with application APIs which use these types.

Application code can then query the type and print it by using the appropriate macros

```
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>
#ifdef UINT128_MAX // ok, because #ifdef
typedef uint128_t bitset;
#else
typedef uint_least64_t bitset;
#endif
int main(void) {
  bitset all = -1;
  printf("the  largest  set  is  %#"PRIXMAX "\n", (uintwidest_t) all);
}
```

Recommended practice

10 It is recommended that the same set of integer literals is consistently accepted by all compilation phases even if **intwidest_t** is chosen to be wider than **signed long long int**. Implementations and applications should not use the types **intwidest_t** and **uintwidest_t** to describe application programmable interfaces.[285)]

7.20.1.6 Greatest basic-width integer types

7 The types **intbasicmax_t** and **uintbasicmax_t** designate a signed and an unsigned integer type, respectively, that are at least as wide as any basic integer type, the types

char16_t	**int_least64_t**	**size_t**	**wchar_t**
char32_t	**ptrdiff_t**	**uint_fast64_t**	**wint_t**
int_fast64_t	**sig_atomic_t**	**uint_least64_t**	

and, provided they exist, **intptr_t** and **uintptr_t**. These types are required. Additionally defined are the types **intmax_t** and **uintmax_t** which are obsolescent aliases for **intbasicmax_t** and **uintbasicmax_t**.

8 **Note 1** The **intbasicmax_t** and **uintbasicmax_t** types are intended to represent values of all types listed above and also the exact-width integer types for all $N \leq 64$.

9 **Note 2** Extended integer types that are not referred by the above list and that are wider than **signed long long int** may also be wider than **intbasicmax_t**. The **intbasicmax_t** and **intwidest_t** types may then be different.

10 **Note 3** The **intwidest_t** and **uintwidest_t** types are intended to provide a fallback for applications that deal with extended integer types that are potentially wider than **intbasicmax_t** or **uintbasicmax_t**.

Recommended practice

11 Unless some **typedef** in the library clause otherwise enforces, it is recommended to resolve **intbasicmax_t** to **long** or **long long int** and **uintbasicmax_t** to 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 int**.

7.20.2 Widths of specified-width integer types

1 The following object-like macros specify the width of the types declared in <stdint.h>. Each macro name corresponds to a similar type name in 7.20.1.

2 ~~Each~~ Unless specified otherwise, each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives. Its implementation-defined value shall be equal to or greater than the value given below, except where stated to be exactly the given value. An implementation shall define only the macros corresponding to those typedef names it actually provides.[286)]

7.20.2.1 Width of exact-width integer types

1

INT*N*_WIDTH	exactly *N*
UINT*N*_WIDTH	exactly *N*

285) This document does not use them further in any of its clauses.

286) The exact-width and pointer-holding integer types are optional.

7.20.2.2 Width of minimum-width integer types

1

INT_LEAST*N*_WIDTH	exactly UINT_LEAST*N*_WIDTH
UINT_LEAST*N*_WIDTH	N

7.20.2.3 Width of fastest minimum-width integer types

1

INT_FAST*N*_WIDTH	exactly UINT_FAST*N*_WIDTH
UINT_FAST*N*_WIDTH	N

7.20.2.4 Width of integer types capable of holding object pointers

1

INTPTR_WIDTH	exactly UINTPTR_WIDTH
UINTPTR_WIDTH	16

7.20.2.5 Width of greatest-width integer types

1

~~INTMAX_WIDTH~~ INTWIDEST_WIDTH	exactly ~~UINTMAX_WIDTH~~ UINTWIDEST_WIDTH
~~UINTMAX_WIDTH~~ UINTWIDEST_WIDTH	64

7.20.2.6 Width of greatest basic-width integer types

1

INTBASICMAX_WIDTH	exactly UINTBASICMAX_WIDTH
UINTBASICMAX_WIDTH	64
INTMAX_WIDTH	INTBASICMAX_WIDTH
UINTMAX_WIDTH	UINTBASICMAX_WIDTH

2 The INTMAX_WIDTH and UINTMAX_WIDTH macros are obsolescent features.

7.20.3 Width of other integer types

3 The following object-like macros specify the width of integer types corresponding to types defined in other standard headers.

4 Each instance of these macros shall be replaced by a constant expression suitable for use in **#if** preprocessing directives. Its implementation-defined value shall be equal to or greater than the corresponding value given below. An implementation shall define only the macros corresponding to those typedef names it actually provides.[6)]

[6)] A freestanding implementation need not provide all of these types.

7.20.3.1 Width of ptrdiff_t

1

PTRDIFF_WIDTH	17

7.20.3.2 Width of sig_atomic_t

1

SIG_ATOMIC_WIDTH	8

7.20.3.3 Width of size_t

1

SIZE_WIDTH	16

7.20.3.4 Width of wchar_t

1

WCHAR_WIDTH	8

7.20.3.5 Width of wint_t

1

WINT_WIDTH	16

7.20.4 Macros for integer constants

1 The following function-like macros expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in <stdint.h>. Each macro name corresponds to a similar type name in 7.20.1.2 or 7.20.1.5. For types wider than **uintbasicmax_t**, the macros shall only be defined if the implementation provides integer literals for the type that are suitable to be used in **#if** preprocessing directives. Otherwise, the definition of these macros is mandatory for any of the types that are provided by the implementation.

2 The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

3 Each invocation of one of these macros shall expand to an integer constant expression suitable for use in **#if** preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

7.20.4.1 Macros for minimum-width integer constants

1 ~~The~~ If defined, the macro **INT*N*_C**(*value*) expands to an integer constant expression corresponding to the promoted type **int_least*N*_t**. ~~The~~ If defined, the macro **UINT*N*_C**(*value*) expands to an integer constant expression corresponding to the promoted type **uint_least*N*_t**. ~~For example, if~~

2 **EXAMPLE** If **uint_least64_t** is a name for the type **unsigned long long int**, then **UINT64_C**(0x123) might expand to the integer constant 0x123ULL.

7.20.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type ~~intmax_t~~ intwidest_t

~~INTMAX_C~~ INTWIDEST_C(*value*)

The following macro expands to an integer constant expression having the value specified by its argument and the type ~~uintmax~~_t uintwidest_t

```
UINTMAX_C UINTWIDEST_C (value)
```

The INTMAX_C and UINTMAX_C macros are obsolescent features.

7.20.4.3 Macros for greatest basic-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type **intbasicmax_t**

```
INTBASICMAX_C(value)
```

The following macro expands to an integer constant expression having the value specified by its argument and the type **uintbasicmax_t**

```
UINTBASICMAX_C(value)
```

7.20.5 Maximal and minimal values of integer types

1 For all integer types for which there is a macro with suffix **_WIDTH** holding the width, maximum macros with suffix **_MAX** and, for all signed types, minimum macros with suffix **_MIN** are defined as by 5.2.4.2. If it is unspecified if a type is signed or unsigned and the implementation has it as an unsigned type, a minimum macro with extension **_MIN**, and value 0 of the corresponding type is defined.

2 The INTMAX_MAX, INTMAX_MIN, and UINTMAX_MAX macros are obsolescent aliases for INTBASICMAX_MAX, INTBASICMAX_MIN, and UINTBASICMAX_MAX, respectively.

7.21.6.1 The fprintf function

…

j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an ~~intmax_t~~ **intbasicmax_t** or ~~uintmax_t~~ **intbasicmax_t** argument; or that a following n conversion specifier applies to a pointer to an ~~intmax_t~~ **intbasicmax_t** argument.

wN Specifies that a following d, i, o, u, x, or X conversion specifier applies to an exact-width integer type argument of exactly *N* bits; or that a following n conversion specifier applies to a pointer to an exact-width integer type argument of exactly *N* bits.

…

7.21.6.2 The fscanf function

…

j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ~~intmax_t~~ **intbasicmax_t** or ~~uintmax_t~~ **intbasicmax_t**.

wN Specifies that a following d, i, o, u, x, or X, or n conversion specifier applies to a pointer to an exact-width integer type argument of exactly *N* bits.

…

7.29.2.1 The fwprintf function

…

j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an ~~intmax_t~~ **intbasicmax_t** or ~~uintmax_t~~ **intbasicmax_t** argument; or that a following n conversion specifier applies to a pointer to an ~~intmax_t~~ **intbasicmax_t** argument.

wN Specifies that a following d, i, o, u, x, or X conversion specifier applies to an exact-width integer type argument of exactly *N* bits; or that a following n conversion specifier applies to a pointer to an exact-width integer type argument of exactly *N* bits.

…

7.29.2.2 The fwscanf function

…

j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ~~intmax_t~~ **intbasicmax_t** or ~~uintmax_t~~ **intbasicmax_t**.

wN Specifies that a following d, i, o, u, x, or X, or n conversion specifier applies to a pointer to an exact-width integer type argument of exactly *N* bits.

…

7.31.17 Greatest-width integer types

1 The **intmax_t** and **uintmax_t** types are obsolescent aliases for **intbasicmax_t** and **uintbasicmax_t.** Using **intmax_t** and **uintmax_t** types in an API is an obsolescent feature.

2 The **INTMAX_WIDTH** and **UINTMAX_WIDTH** macros are obsolescent aliases for **INTBASICMAX_WIDTH** and **UINTBASICMAX_WIDTH**.

3 The **INTMAX_C** and **UINTMAX_C** macros are obsolescent aliases for **INTBASICMAX_C** and **UINTBASICMAX_C**.

4 The **INTMAX_MAX**, **INTMAX_MIN**, and **UINTMAX_MAX** macros are obsolescent aliases for **INTBASICMAX_MAX**, **INTBASICMAX_MIN**, and **UINTBASICMAX_MAX**, respectively.

Annex M

(informative)

Change History

M.1 Fifth Edition

1 Major changes in this fifth edition (**__STDC_VERSION__** *yyyymm*L) include

— remove obsolete sign representations and integer width constraints

— added the **intbasicmax_t** and **uintbasicmax_t** and allow extended integer types to be wider than **these types**

— added a one-argument version of **_Static_assert** —

harmonization with ISO/IEC 9945 (POSIX)

- extended month name formats for **strftime** • integration of functions **memccpy**, **strdup**,

strndup — harmonization with floating point standard IEC 60559

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of decimal floating-point technical specification TS 18661-4a

— the macro **DECIMAL_DIG** is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added **nodiscard**, **maybe_unused** and **deprecated** attributes

M.2 Fourth Edition

1 There were no major changes in the fourth edition (**__STDC_VERSION__**201710L), only technical corrections and clarifications.

M.3 Third Edition

1 Major changes in the third edition (**__STDC_VERSION__**201112L) included

— conditional (optional) features (including some that were previously mandatory)

— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (<stdatomic.h> and <threads.h>)

— additional floating-point characteristic macros (<float.h>)

— querying and specifying alignment of objects (<stdalign.h>, <stdlib.h>)

— Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 197692004)

— type-generic expressions

— static assertions

— anonymous structures and unions

— no-return functions

— macros to create complex numbers (<complex.h>)

— support for opening files for exclusive access