Adding a Fundamental Type for N-bit integers

Committee: ISO/IEC JTC1 SC22 WG14

Document Number: N2763
Date: 2021-06-21
Revises document number: N2709
Authors: Aaron Ballman, Melanie Blower, Tommy Hoffner, Erich Keane
Reply to: Aaron.Ballman@intel.com, Melanie.Blower@intel.com,
thoffner@acm.org, Erich.Keane@intel.com

Contents

Summary of Changes

N2763

- Removed alternative wording section.
- Editorial fix to 6.2.5p4 for use of “signed”.
- Added straw poll results.

N2709

- Corrected the integer conversion ranks and added addition explanation of intent to the prose.

N2646

- Removed the _BitwidthOf, file IO, literal suffixes, and atomic functionality and added a Future Directions section to the prose.
- Renamed _ExtInt to _BitInt.
- Corrected the changes to stdint.h to include extended integer types in addition to standard.
- Added alternate proposed wording for an annex, added prose to discuss it.
- Clarified that the types defined in <stdint.h> cannot be defined in terms of bit-precise integers.
- Minor editorial tweaks.
- Added straw poll questions.

N2590

- Significantly revised the proposed wording.
- Replaced previous inttypes.h macros with the %qN length modifier for I/O support.
- Removed the _STR_TO_EXTINT macro and supported oversized constants directly.
- Made the _Bitwidthof operator apply in constant expressions and apply to bit-fields.
- Added a helper macro for querying lock-free atomic support and a type mapping to an atomic type name.
- Added a section about impacts on generic programming.
- Added more rationale and prior art information.
- Exempted _ExtInt from impacting the definition of [u]intmax_t and the exact-width integer types.

N2534

- Proposed wording

N2501

- High level introduction to the topic without proposed wording, with corrections to the original document.

N2472

- Original report.

Introduction and Rationale

We propose adding a family of integer types spelled as _BitInt(N), where N is an integral constant expression representing the exact number of bits to be used to represent the type. In most application code, the usual 8-, 16-, 32-, 64-bit width integer types provide satisfactory expressiveness for common integer uses. However, there are use cases for integer types with a specific bit-width in application domains, such as using 256-bit integer values in various cryptographic symmetric ciphers like AES, when calculating SHA-256 hashes, representing a 24-bit color space, or when describing the layout of network or serial protocols.

Further, in the case of FPGA hardware, using normal integer types for small value ranges where the full bit-width isn't used is extremely wasteful and creates severe performance/space concerns. At the other extreme, FPGA's can support really wide integers, essentially providing arbitrary precision, and existing FPGA applications make use of really large integers, for example up to 2031 bits. The clang implementation of _BitInt provides support for bit widths up to 10^{24} (N.B., the clang feature is spelled _ExtInt, which was an earlier proposed spelling for _BitInt).

An integer type with an explicit bit-width allows programmers to portably express their intent in code more clearly. Today, the programmer must pick an integer datatype of the next larger size and manually perform mask and shifting operations. However, this is error prone because integer widths vary from platform to platform and exact-width types from stdint.h are optional, it is not always obvious from code inspection when a mask or a shift operation is incorrect, implicit conversion can lead to surprising behavior, and the resulting code is harder for static analyzers and optimizers to reason about. By allowing the programmer to express the specific bit-width they need for their problem domain directly from the type system, the programmer can write less code that is more likely to be correct and easier for tooling and humans to reason about.

Proposed solution

A set of bit-precise integer types using the syntax _BitInt(N) where N is an integer that specifies the number of bits that are used to represent the type, including the sign bit. The keyword _BitInt followed by a parenthesized integer constant is a type specifier, thus it can be used in any place a type can; whether it can be the type of a bit-field is implementation defined. When polled on whether WG14 would like something along these lines at the Oct 2020 meeting, the results were 16/1/4 (consensus).

The original proposal was to name the feature _ExtInt and refer to it in standards text as a "special extended integer type". However, during committee discussion, the feature was changed to be called a "bit-precise integer type" and so the name _ExtInt was a misnomer. During the Nov 2020 meeting, WG14 polled on whether to change the name and the results were 12/0/8 (consensus). The name has been changed to _BitInt to more closely match with the notion of a bit-precise integer type.

At the Nov 2020 meeting, the question was raised about whether this feature should be surfaced via macros within a header file instead of fully specified types in the language portion of the standard. The question was not polled, but this version of the paper proposes alternative wording for an annex that exposes the type via a new header file so that the committee can compare the approaches. (N.B., the prose of this proposal has not been updated to reflect the annex wording beyond what's found in this paragraph and the alternative proposed wording.) The annex uses the macros `_BitInt(N)` and `_UnsignedBitInt(N)`, which expand to an implementation-defined type. However, our belief is that exposing the type via macros makes a less compelling feature. Not having a portable name for the type by hiding it behind macros makes it harder to write a C library with APIs using _BitInt that interoperates with other languages that do not have a C preprocessor (which is why the annex requires the underlying type to be implementation-defined rather than unspecified). Also, it introduces the novel concept of a non-typedef integer datatype that is signed by default but that cannot be spelled with the `unsigned` keyword. e.g., `unsigned _BitInt(4) j;` is a reasonable construct for a programmer to write given the common prevalence of integer types like `unsigned int`, but will instead be a constraint violation. Further, this approach may introduce a QoI challenge for implementations that have an existing implementation of bit-precise integer types which is not spelled `_BitInt`, like Clang. It may seem plausible that such an implementation could define the proposed macros as:

```
#define _BitInt(N) _ExtInt(N)
#define _UnsignedBitInt(N) unsigned _ExtInt(N)
```

However, if a user wrote a declaration like `unsigned _BitInt(4) j;`, it would expand to `unsigned _ExtInt(4) j;` and appear as a valid declaration of an unsigned bit-precise integer that is indistinguishable from use of the `_UnsignedBitInt` macro. Alternatively, if the macro was defined to expand to `signed _ExtInt(N)` explicitly, the diagnostics produced by the implementation may talk about duplicate type specifiers rather than use of an invalid type specifier as in: https://godbolt.org/z/1nPEca. To conform to the C standard requirements for type specifiers, such an implementation might be forced to introduce parser changes or introduce another type in the type system to be able to properly diagnose the invalid use of the `unsigned` type specifier with the quality expected for a production C compiler.

We are not proposing this feature as a Technical Specification because we believe there is sufficient motivation for a mandatory, portable, exact bit-width integer datatype in the international standard. Significant prior art exists for such a datatype and demonstrates that the industry sees a need in this space, and we do not know what further information or experience would be gained by introducing the feature in a technical specification.

A _BitInt can be explicitly declared as either signed or unsigned by using the signed/unsigned keywords. If no sign specifier is used or if the signed keyword is used, the _BitInt type is a signed integer.

The N expression is an integer constant expression, which specifies the number of bits used to represent the type, following normal integer representations for both signed and unsigned types. Both a signed and unsigned _BitInt of the same N value will have the same number of bits in its representation. Many architectures don't have a way of representing non power-of-2 integers, but these architectures can emulate these types using larger integers.

In order to be consistent with the C language and make the _BitInt types useful for their intended purpose, _BitInt types follow the usual C standard integer conversion ranks. A _BitInt type has a greater rank than any standard integer type with less width. However, they have lower rank than any of the standard integer types with the same width, which means that a binary expression with a 32-bit `int` and a `_BitInt(32)` will use `int` as a the common type. (cf 6.3.1.1 "The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.") When the widths of the integer types are identical, a conversion to standard integer type is preferred over a _BitInt type. However, to give more consistent semantics for existing extended integer types, the rank of an extended integer type and bit-precise integer type of the same width is implementation-defined. Usual arithmetic conversions also work the same, where the smaller ranked integer is converted to the larger. When polled on whether the committee agrees with the chosen integer ranks at the Oct 2020 meeting, the results were 7/2/10 (weak consensus).

There is one crucial exception to the C rules for integer promotion: _BitInt types are excepted from the integer promotions. Operators typically will promote operands smaller than the width of an int to an int. Doing these promotions would inflate the size of required hardware on some platforms, so _BitInt types aren't subject to the integer promotion rules. For example, in a binary expression involving a _BitInt(12) and an unsigned _BitInt(3), the usual arithmetic conversions would not promote either operand to an int before determining the common type. Because one type is signed and one is unsigned and because the signed type has greater rank than the unsigned type (due to the bit-widths of the types), the unsigned _BitInt(3) will be converted to _BitInt(12) as the common type. We do not propose promoting to a _BitInt wide enough to perform the operation without loss of precision because empirical evidence suggests that mildly complex arithmetic expressions can quickly cause promotion to surprisingly wide _BitInt types with poor performance characteristics. Further, integer promotions do not always yield intuitive results. While a case like promoting an expression SomeBitInt12 * SomeBitInt8 to use ints in order to avoid overflow may seem attractive, a case like SomeBitInt28 * SomeBitInt8 where integer promotions would promote to a (potentially) 32-bit type that's still insufficient to represent the resulting value shows that this implicit conversion doesn't always help. By exempting _BitInt from the integer promotion rules, users are given a more expressive model that allows them to express the bit-precise semantics of expressions in a way that is easier for tooling like static analyzers to reason about. When polled on whether the committee is in favor of excepting _BitInt from the integer promotions, the results were 14/0/6 (consensus).

Prior Art

Intel has introduced this functionality in our FPGA targeting compilers; the High Level Synthesis (HLS) compiler and FPGA OpenCL compiler under the name "Arbitrary Precision Integer" (ap_int for short). See References section below for details about these compilers. This feature has been extremely useful and effective for our users, permitting them to optimize their storage and operation space on an architecture where both can be extremely expensive. The Intel Compilers for the Intel FPGA have many users that rely on the ap_int type to achieve efficient programs on a daily basis.

The ac_int type is another demonstration of prior art as a de facto standard for bit-precise integer types in C++ and was originally developed by Mentor Graphics: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf.

Another implementation of this feature, implemented from scratch, is available in Intel's oneAPI product: https://software.intel.com/en-us/oneapi.

_BitInt has been implemented directly in the Clang 11 release as a language extension, spelled _ExtInt: https://clang.llvm.org/docs/LanguageExtensions.html#extended-integer-types.

The XiLinx FPGA compiler for HLS also provides users a similar solution: a C++ "arbitrary precision" integer type so that solutions can be optimized. Note that the naming scheme for C types builds the integer width into the type name, a la int9, and for XiLinx the maximum width is limited to 1024 bits. More information about this implementation can be found at: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/integer_types.html

ABI Considerations

_BitInt(N) types align with existing calling conventions. They have the same size and alignment as the smallest basic type that can contain them. Types that are larger than __int64_t are conceptually treated as struct of register size chunks. The number of chunks is the smallest number that can contain the type.

With the Clang implementation on Intel64 platforms, _BitInt types are bit-aligned to the next greatest power-of-2 up to 64 bits: the bit alignment A is min(64, next power-of-2(>=N)). The size of these types is the smallest multiple of the alignment greater than or equal to N. Formally, let M be the smallest integer such that A*M >= N. The size of these types for the purposes of layout and sizeof is the number of bits aligned to this calculated alignment, A*M. This permits the use of these types in allocated arrays using the common sizeof(Array)/sizeof(ElementType) pattern. The authors will discuss the ABI requirements with the different ABI groups.

Safety

Overflow occurs when a value exceeds the allowable range of a given data type. For example, `(_BitInt(3))7 + (_BitInt(3))2` overflows, and the result is undefined as with other signed integer types. To avoid the overflow, the operation type can be widened to 4 bits by casting one of the operands to _BitInt(4). As with other unsigned integer types, overflow of an unsigned _BitInt is well-defined and the value wraps around with twos complement semantics.

Interaction with _Generic

_BitInt(N) is a distinct type from _BitInt(M) when N != M, which means that use within a generic selection expression requires the developer to name the exact bit-width they would like to match. This could feel like a “hole” in the generic programming capabilities of C because it does not provide a direct way to match the entire family of _BitInt types. However, supporting the ability to have a single association that matches any size _BitInt is unlikely to meet the user's expectations because C does not support a way to write type generic statements or function declarations to match the actual type.

We believe that any improvements to generic programming should be explored as an orthogonal topic to the introduction of _BitInt. The proposed behavior gives a defensible model within generic selection expressions that should be familiar to C programmers who write generic function interfaces in that _BitInt(32) being distinct from _BitInt(31) should be no more difficult to understand than int and long int being unique types.

Future Directions

Once the committee is content with the basic functionality of the bit-precise integer type, we intend to introduce proposals to extend or polish the feature. For reference, we plan to propose the following extensions to this functionality:

- Adding a format specifier so that a bit-precise integer can be used directly with the `fprintf` and `fscanf` family of functions.
- Adding atomic support for atomic bit-precise integers.
- Adding the `_BitwidthOf` unary operator to query the width (rather than size) of an integer.
- Adding literal suffixes to form integer constants with a bit-precise type.

Proposed Straw Polls

We would like to see bit-precise integers added into C23 with sufficient time for subsequent, related proposals to make progress. To that end, we’d like to poll adopting one of the two wording approaches (with any necessary minor wording corrections being applied after the meeting).

Does WG14 wish to adopt N2646, proposed wording alternative one, into C2x?

Does WG14 wish to adopt N2646, proposed wording alternative two, into C2x?

If neither of the polls gains consensus for adoption, we would like WG14 to pick a preferred wording approach to reduce the editing burden on subsequent revisions of the paper.

Does WG14 prefer the specification approach taken in wording alternative one from N2646?

(For the above poll, a Yes vote is support for something along the lines of wording alternative one and a No vote is support for something along the lines of wording alternative two.)

Straw Poll Results

At the June 2021 virtual meeting, a straw poll was taken; does WG14 wish to adopt N2709, wording alternative one, into C23: 14/2/5 (consensus). This paper only makes editorial changes requested by the committee and removes the alternative wording proposal for clarity.

Proposed Wording

The wording proposed is a diff from WG14 N2573. Green text is new text, while ~~red~~ text is deleted text.

Modify 5.2.4.2.1p1 (add to the end of the list):

> — width for an object of type `_BitInt` or `unsigned _BitInt`
> `BITINT_MAXWIDTH /* see below */`
>
> The macro `BITINT_MAXWIDTH` represents the maximum width *N* supported in the declaration of a bit-precise integer type (6.2.5) in the type specifier `_BitInt(`*N*`)`. The value BITINT_MAXWIDTH shall expand to a value that is greater than or equal to the value of ULLONG_WIDTH.

Modify 6.2.5p4: *Drafting note: this moves the existing text into a new paragraph.*

> There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) ~~There may also be implementation-defined *extended signed integer types*.[N] The standard and extended signed integer types are collectively called *signed integer types*.[M]~~

Insert a new paragraph immediately after 6.2.5p4: *Drafting note: the footnotes in the new paragraph are the same as the ones from the preceding paragraph.*

> A *bit-precise signed integer type* is designated as **_BitInt**(N) where N is an integer constant expression that specifies the number of bits that are used to represent the type, including the sign bit. Each value of N designates a distinct type. [Footnote: Thus, _BitInt(3) is not the same type as _BitInt(4).] There may also be implementation-defined *extended signed integer types*.[41] The standard signed integer types, bit-precise signed integer types, and extended signed integer types are collectively called *signed integer types*.[42]

Modify footnotes 41 and 42, respectively:

41) ~~Therefore, a~~Any statement in this document about signed integer types also applies to the bit-precise signed integer types and the extended signed integer types, unless otherwise noted.

42) ~~Therefore, a~~Any statement in this document about unsigned integer types also applies to the bit-precise unsigned integer types and the extended unsigned integer types, unless otherwise noted.

Modify the existing 6.2.5p6:

> For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword unsigned) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type _Bool and the unsigned integer types that correspond to the standard signed integer types are the standard unsigned integer types. The unsigned integer types that correspond to the bit-precise signed integer types are the *bit-precise unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard unsigned integer types, bit-precise unsigned integer types, and extended unsigned integer types are collectively called *unsigned integer types*.

Modify the existing 6.2.5p7:

> The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the bit-precise signed integer types and bit-precise unsigned integer types are collectively called the *bit-precise integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

Modify 6.3.1.1p1:

> …
> — The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.
> — The rank of a bit-precise signed integer type shall be greater than the rank of any standard integer type with less width or any bit-precise integer type with less width.
> — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
> — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width or bit-precise integer type with the same width.
> — The rank of any bit-precise integer type relative to an extended integer type of the same width is implementation-defined.
> …

Modify 6.3.1.1p2:

> The following may be used in an expression wherever an int or unsigned int may be used:
> — An object or expression with an integer type (other than int or unsigned int) whose integer conversion rank is less than or equal to the rank of int and unsigned int.
> — A bit-field of type _Bool, int, signed int, or unsigned int.

If the original type is not a bit-precise integer type (6.2.5) and if~~If~~ an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int. These are called the integer promotions. 61) All other types are unchanged by the integer promotions.

Insert a new example after 6.3.1.8p1:

EXAMPLE 1 One consequence of _BitInt being exempt from the integer promotion rules (6.3.1.1) is that a `_BitInt` operand of a binary operator is not always promoted to an `int` or `unsigned int` as part of the usual arithmetic conversions. Instead, a lower-ranked operand is converted to the higher-rank operand type and the result of the operation is the higher-ranked type.

```
_BitInt(2) a2 = 1;
_BitInt(3) a3 = 2;
_BitInt(33) a33 = 1;
char c = 3;

a2 * a3 /* As part of the multiplication, a2 is converted to
           _BitInt(3) and the result type is _BitInt(3). */

a2 * c /* As part of the multiplication, c is promoted to int,
          a2 is converted to int and the result type is int. */
a33 * c /* As part of the multiplication, c is promoted to int,
           then converted to _BitInt(33) and the result type
           is _BitInt(33). */


void func(_BitInt(8) a1, _BitInt(24) a2) {
  /* Cast one of the operands to 32-bits to guarantee the
     result of the multiplication can contain all possible
     values. */
  _BitInt(32) a3 = a1 * (_BitInt(32))a2;
}
```

Modify 6.4.1p1 to add a new keyword:

`_BitInt`

Modify 6.7.2p1 to add a new entry to the *type-specifier* list:

`_BitInt (` *constant-expression* **`)`**

Modify 6.7.2p2 to add two new items to the list immediately below **unsigned long long**:

— `_BitInt(`*constant-expression*`)`, or `signed _BitInt(`*constant-expression*`)`
— `unsigned _BitInt(`*constant-expression*`)`

Insert a new paragraph after 6.7.2p3 to the constraints section:

The parenthesized constant expression that follows the **_BitInt** keyword shall be an integer constant expression **N** that specifies the width (6.2.6.2) of the type. The value of **N** for `unsigned _BitInt` shall be greater than or equal to 1. The value of **N** for `_BitInt` shall be greater than or equal to 2. The value of N shall be less than or equal to the value of BITINT_MAXWIDTH (see 5.2.4.2.1).

Modify 7.20p4:

> For each type described herein that the implementation provides,[284] `<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"). None of the types shall be defined as a synonym for a bit-precise integer type.

Modify 7.20.1.1p3:

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

Modify Annex E p2, Adding a new line below `ULLONG_WIDTH`:

```
#define  BITINT_MAXWIDTH      64    // ULLONG_WIDTH
```

Acknowledgements

The authors would like to recognize the following people for their help with this work: Jens Gustedt, JeanHeyd Meneide, Joseph S. Myers, Richard Smith, and Martin Uecker.

References

1. The HLS compiler: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html refer to "Arbitrary Precision Integer"
2. The OpenCL FPGA compiler: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
3. The ac_int datatype: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf
4. The current clang review: https://reviews.llvm.org/D73967
5. https://reviews.llvm.org/D59105 An earlier version of this feature was proposed for acceptance into clang/llvm, the code review is here.
6. The XiLinx HLS compiler arbitrary precision data types https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf
7. The bitWidth type property in swift: https://developer.apple.com/documentation/swift/int/2885648-bitwidth