N3862

Removing the Implementation-Defined Signedness of Bit-Fields, v2

Document	N3862
Date	2026-03-16
Reply to	Robert C. Seacord <rcseacord@gmail.com>
Target	C2y
Abstract	This paper resolves ambiguities concerning bit-field signedness by removing the implementation-defined reinterpretation of `int` bit-fields as `unsigned int`.
Reference	N3783

Change Log

2026-02-19:

- Initial version

2026-03-16:

- Updated EXAMPLE 3 in subclause 6.7.9, “Type definitions”, paragraph 6
- Added Section 8 on Previous C Versions
- UPdated historical motivation

1 Introduction

N3783 Subclause 6.7.3.1, paragraph 7 specifies:

> Each of the comma-separated multisets designates the same type, **except that for bit-fields, it is implementation-defined whether the specifier `int` designates the same type as `signed int` or the same type as `unsigned int`**.

This exception, inherited from C90, has become a persistent source of ambiguity as the language has added features including `typedef` redefinition (C11), `typeof` specifiers (C23), and `_BitInt` types (C23).

Footnote 138 in subclause 6.7.3.2, paragraph 12 states this exception extends to `typedef` names and `typeof` specifiers:

> As specified in 6.7.3, if the actual type specifier used is `int` or a typedef-name defined as `int`, then it is implementation defined whether the bit-field is signed or unsigned. This includes an `int` type specifier produced using the typeof specifiers (6.7.3.6).

DR 315 asks and answers the following question:

> Must bit-fields of type char nevertheless have the same signedness as ordinary objects of type **`char`**, and similarly for those of types **`short`** (or **`short int`**), **`long`** (or **`long int`**), **`long long`** (or **`long long int`**)?
>
> These are all determined by the implementation-defined behavior specified in 6.7.2.1#4.

This answer further extends the Subclause 6.7.3.1, paragraph 7 exception to other implementation-defined bit-field types such as `char`, `short`, `long`, and `long long` — but the normative text does not clearly support these extensions

The C++ committee addressed a closely related set of issues through CWG 739 (*Signedness of bit-field with typedef or template parameter type*), which was applied as a defect resolution against C++11 in 2012, and so was always part of C++14 from its initial publication. In practice, a DR gets applied to all relevant older standards, so C++ implementations use the same rule for C++98 as well.

This paper proposes eliminating the implementation-defined signedness of bit-fields entirely. Under this change, the specifier `int` always designates `signed int`, including when used as the type of a bit-field. This:

- resolves all five open questions raised by Issue 1007 without case-by-case special rules,
- eliminates a long-standing portability trap,
- improves compatibility with C++, and
- aligns with the MISRA C:2025 safety-critical coding standard (e.g., R.6.1 "Bit-fields shall only be declared with an appropriate type").

1.1 Alternative Approach

An alternative approach would be to make `int` a constraint violation for bit-fields (i.e., requiring `signed int` or `unsigned int` explicitly). This approach has the following drawbacks:

- It breaks existing code. Every `struct { int flags : 4; }` in every C codebase becomes a constraint violation. The proposed approach changes the behavior of zero programs on any major compiler, because GCC, Clang, MSVC, and ICC all already default to treating `int` bit-fields as signed.
- It creates a new inconsistency. Today, `int` is a valid type specifier everywhere. The bit-field signedness exception is the only place where `int` and `signed int` diverge. Removing that exception makes the language more uniform. A constraint violation would create a different anomaly: "`int` is valid everywhere except bit-fields." It would replace one special case with another.
- It diverges from C++. C++ permits `int` bit-fields. Making `int` a constraint violation in C would break common-subset code and create a new C/C++ incompatibility — the opposite of the paper's stated goal. The proposed approach makes C a subset of C++: code valid under the new C rule is valid under every C++ implementation.
- No precedent. C has never made a previously valid type specifier into a constraint violation in a specific declarator context. Doing so here would be a novel kind of breaking change with no comparable precedent in the standard's history. Removing implementation-defined behavior, by contrast, has ample precedent (e.g., C99 requiring `int` to be at least 16 bits narrowed previous implementation freedom without breaking code).
- The transition cost is asymmetric. Under the proposed approach, implementations that currently default to `signed` (all of them) need to do nothing. Implementations that use `-funsigned-bitfields` can keep it as a non-conforming extension. Under the constraint violation approach, every translation unit containing an `int` bit-field must be edited, across all implementations, regardless of what behavior they already had.

In short: the constraint violation approach solves the same problem but at far greater cost, with a new inconsistency, and a new C++ incompatibility. The proposed approach solves it for free on every existing implementation.

2 Problem Description

2.1 Question 1 — Standard Library Typedefs

```
#include <stddef.h>
#include <stdint.h>

struct s1 {
  ptrdiff_t b1a : 20;
  int32_t   b1b : 20;
};
```

`ptrdiff_t` is required to be a signed integer type, and `int32_t` is required to be a signed integer type with width exactly 32. If the underlying type is `int`, the current rule permits an

implementation to treat the bit-field as unsigned — contradicting the guarantee that these types are signed.

Despite its name, `typedef` only creates a type alias and not a new type, so any implied semantic in the type alias name is only for the benefit of the reader and has no impact on what the compiler does after the alias is replaced with the underlying type.

Moreover, what is the implied semantic of using a `uint32_t` type to define a 20-bit wide bit-field? It's just contradictory and that creates confusion to readers of the source code. MISRA C:2025 has the long standing rule "Bit-fields shall only be declared with an appropriate type", R.6.1, that limits the types to be used with bit-fields to `unsigned int`, `signed int`, and `bool` and R.6.2 further states that "Single-bit named bit-fields shall not be of a signed type" to prevent such confusion.

2.2 Question 2 — Typedef Redefinition

```
typedef int T;
typedef signed int T;  // valid in C11+: same type

struct s2 {
  T b2 : 20;
};
```

C11 permits redefinition of a typedef with the same type. Outside of bit-fields, `int` and `signed int` are the same type, so this redefinition is valid. But as a bit-field, `int` might be unsigned while `signed int` is signed. Is `T` explicitly signed after the second definition? The current wording does not address this.

2.3 Question 3 — `typeof` Specifiers

```
int i;
signed int si;

struct s3a {
  typeof(int)             b3a : 20;  // int → impl-defined?
  typeof(signed int)      b3b : 20;  // signed int → signed
  typeof(i)               b3c : 20;  // int → impl-defined?
  typeof(si)              b3d : 20;  // signed int → signed
  typeof((int)si)         b3e : 20;  // cast to int → impl-defined?
  typeof((signed int)i)   b3f : 20;  // cast to signed int → signed
};
```

Some cases are clear, but others involving composite types, usual arithmetic conversions, and standard-specified result types are not:

```
int i;
signed int si;
extern int x;
extern signed int x;       // same type, but which "spelling" wins?

signed short s;

struct s3b {
  typeof(x)     b3g : 20;  // composite type of int and signed int
  typeof(i+si)  b3h : 20;  // usual arithmetic conversions → int
};

struct s3c {
  typeof(+s)    b3i : 20;  // integer promotions → int
  typeof(s==s)  b3j : 20;  // comparison → int
```

```
  typeof(0)      b3k : 20;  // literal → int
  typeof(-1)     b3l : 20;  // negation of literal → int
};
```

The standard specifies many of these result types as `int`, not `signed int`, because bit-field signedness wasn't considered. The distinction between `int` and `signed int` should be invisible at the type level; the current bit-field exception makes it visible.

2.4 Question 4 — `_BitInt`

```
struct s4 {
  _BitInt(32) b4 : 20;  // may this be unsigned?
};
```

The normative text in 6.7.3.1 mentions only `int`. `_BitInt(N)` is a distinct type, and support for `_BitInt` bit-fields is not implementation-defined (unlike `short` or `long` bit-fields). There is a reasonable reading that `_BitInt(N)` is never subject to signedness reinterpretation, but the standard should be explicit.

2.5 Question 5 — `_BitInt(1)` Constraint Interaction

```
struct s5 {
  _BitInt(1) b5 : 1;  // valid if reinterpreted as unsigned?
};
```

Yes. `_BitInt(1)` is valid in N3783 as a result of the adoption of N3747 Seacord, Integer Sets, v5.

3 Analysis

3.1 Historical Motivation

The original C90 rule reflected real hardware: some architectures stored bit-fields in unsigned containers regardless of the declared type, making signed extraction expensive. On architectures such as the PDP-11 or the 6502, bit-field operations had to be synthesized using multiple instructions, making the signed variant notably more taxing due to the lack of barrel shifters or dedicated extraction logic.

Modern architectures universally support efficient signed bit-field extraction. For example, modern architectures such as AArch64 include specific instructions like SBFX (Signed Bitfield Extract) and UBFX (Unsigned Bitfield Extract) to ensure both operations are equally efficient. AArch32 has had SBFX since ARMv7 (from about 2004 onwards). AArch64 (ARM64) also has the SBFM (Signed Bitfield Move) instruction. SBFM is a versatile instruction used for bitfield

manipulation, specifically for extracting, sign-extending, and inserting bitfields. Older, pre-2004, CPUs such as ARM7TDMI (ARMv4T) have to use logical shift left and arithmetic shift right. On small embedded systems,even the signed-extension of integers can be more expensive than zero-extension.

The implementation freedom is no longer motivated by the limitations of modern hardware; it exists only for backward compatibility with implementations that chose unsigned.

3.2 C++ Precedent

C++ core issue CWG 739 (Signedness of plain bit-fields) addressed an equivalent set of ambiguities involving typedef names and template parameters. The proposed resolution (February 2012), moved to DR at the October 2012 meeting and applied as a defect resolution against C++11, deleted the implementation-defined signedness rule from [class.bit] paragraph 3 entirely. The sentence:

It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int`, `long`, or `long long` bit-field is signed or unsigned.

was removed. Since C++11, plain `int` bit-fields are always `signed int` in C++.

The Annex C compatibility note added by CWG 739 (C.7.7 [diff.class]) documents this as a deliberate divergence from C:

11.4.10 [class.bit]

Change: Bit-fields of type plain `int` are signed.

Rationale: Leaving the choice of signedness to implementations could lead to inconsistent definitions of template specializations. For consistency, the implementation freedom was eliminated for non-dependent types, too.

Effect on original feature: The choice is implementation-defined in C, but not so in C++.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

The change proposed in this paper would eliminate this documented C/C++ incompatibility by adopting the same rule in C.

3.3 Safety-Critical Coding Standards

Both MISRA C:2025 Rule 6.1 and MISRA C++:2023 Rule 6.1.4 require explicit signedness for all bit-field declarations. Code that complies with these rules is unaffected by this change, because every bit-field already uses either `signed int` or `unsigned int` (never plain `int`).

Removing the implementation-defined behavior benefits code that does *not* follow these rules — which is the majority of general-purpose C code.

3.4 Implementation Survey

In practice, major implementations such as GCC, Clang, MSVC, and ICC default to treating `int` bit-fields as signed. GCC provides `-funsigned-bitfields` to select unsigned behavior but Clang does not; this flag would become a non-conforming extension under the proposed change. No implementation defaults to unsigned `int` bit-fields.

SDCC defaults to unsigned bit-fields of type `int` up to the latest release. In the next SDCC release, bit-fields of type `int` will be signed.

4 Proposed Direction

Remove the implementation-defined signedness exception for bit-fields entirely. After this change:

- `int` designates `signed int` unconditionally, including as a bit-field type.
- `char` bit-fields (where `char` is an implementation-defined bit-field type) retain the separately implementation-defined signedness of `char` (6.2.5), which is a property of the `char` type itself, not of bit-fields.
- `short`, `long`, and `long long` bit-fields (where these are implementation-defined bit-field types) are always signed, as they are in non-bit-field contexts.
- `_BitInt(N)` bit-fields are always signed; `unsigned _BitInt(N)` must be used for unsigned bit-fields.

`_BitInt(N)` is not mentioned in subclause 6.7.3.1, paragraph 7 and is never subject to signedness reinterpretation.

5 Proposed Wording

Text in green is added to the C2Y working draft N3783. ~~Text in red~~ that has been struck through is removed from the C2Y working draft N3783.

Subclause 6.7.3.1 Type specifiers, paragraph 7

Change:

> Each of the comma-separated multisets designates the same type~~, except that for bit-fields, it is implementation-defined whether the specifier int designates the same type as signed int or the same type as unsigned int~~.

Subclause 6.7.3.2, Structure and union specifiers, footnote 138

In paragraph 12, delete footnote 138 in its entirety:

> ~~As specified in 6.7.3, if the actual type specifier used is int or a typedef-name defined as int, then it is implementation-defined whether the bit-field is signed or unsigned. This includes an int type specifier produced using the typeof specifiers (6.7.3.6).~~

Subclause 6.7.9 Type definitions, paragraph 6

Change EXAMPLE 3:

> declare a typedef name `t` with type `signed int`, a typedef name `plain` with type `int`, and a structure with three bit-field members, one named `t` that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it can be accessed) would contain values in the range [−16, +15], and one named `r` that contains values in ~~one of~~ the range~~s [0, 31] or~~ [−16, +15]. ~~(The choice of range is implementation-defined.)~~

Subclause Annex J.3 Implementation-defined behavior

In J.3.11 (Structures, unions, enumerations, and bit-fields), remove:

> ~~(1) Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (6.7.3, 6.7.3.2).~~

Subclause 6.7.3.2 Structure and union specifiers, paragraph 5 (bit-field constraint)

No change required. The existing constraint:

> A bit-field shall have a type that is a qualified or unqualified `bool`, `signed int`, `unsigned int`, a bit-precise integer type, or other implementation-defined type. It is implementation-defined whether atomic types are permitted.

remains correct. With the change to 6.7.3.1, a bit-field declared with type specifier `int` now unambiguously has type `signed int`, which satisfies this constraint.

6 Answers to Questions

#	Question	Answer Under Proposed Wording

1	Must standard library typedefs that are required to be signed integer types remain signed as bit-fields?	**Yes.** The type specifier `int` always designates `signed int`. A typedef for a signed integer type produces a signed bit-field. `ptrdiff_t` and `int32_t` bit-fields are always signed.
2	Is a bit-field using a typedef after `typedef int T; typedef signed int T;` explicitly signed?	**Moot.** `int` and `signed int` designate the same type unconditionally, so both typedef definitions produce the same type and the bit-field is signed.
3	Which `typeof` cases are explicitly signed?	**All of them**, when the resulting type is `int`. Since `int` is always `signed int`, every case in `s3a`, `s3b`, and `s3c` produces a signed bit-field. The `typeof` specifiers need no special treatment.
4	Does the signedness reinterpretation include `_BitInt`?	**No.** `_BitInt(N)` was never mentioned in the implementation-defined signedness rule, and support for `_BitInt` bit-fields is not implementation-defined. The proposed change makes this even clearer by eliminating the only signedness reinterpretation mechanism.
5	Does the `_BitInt(1)` constraint apply before or after signedness reinterpretation?	**No reinterpretation occurs.** `_BitInt(1) b : 1;` is a constraint violation because N ≥ 2 is required for signed `_BitInt`. If an unsigned single-bit field is desired, `unsigned _BitInt(1)` or `bool` must be used.

7 Compatibility Considerations

Aspect	C++ (since CWG 739 / C++11 DR)	C23 (current)	C2y (proposed)
Plain `int` bit-field	Always `signed int`	Implementation-defined	Always `signed int`
Typedef for `signed int`	Always signed	Ambiguous (fn. 132)	Always signed
`typeof` producing `int`	Always signed	Ambiguous (fn. 132)	Always signed

`_BitInt(N)`	N/A	Ambiguous (Q4)	Always signed

Backward compatibility with C23 and earlier: Code that explicitly uses `signed int` or `unsigned int` for bit-fields is unaffected. Code that uses plain `int` and relies on it being unsigned on a particular implementation will change behavior. However:

- No major compiler defaults to unsigned `int` bit-fields.
- Code relying on unsigned `int` bit-fields is already non-portable.
- Safety-critical coding standards already prohibit implicit signedness.
- Compilers can provide `-funsigned-bitfields` as a non-conforming extension for legacy code.
- A non-exhaustive independent survey found that 6 out of 13 build platforms have both signed and unsigned bit-field configurations.

Forward compatibility with C++: Code written under the new C rule (where `int` bit-fields are signed) is valid under *any* C++ implementation, because `signed int` is always one of the permitted interpretations. The change makes C a subset of C++ in this regard. A corresponding change in C++ is encouraged but not required for compatibility.

Feature-test macro: No feature-test macro is proposed. The change can be detected portably at compile-time as follows:

```
struct signed_test { int b : 1; };
constexpr struct signed_test test = { -1 };
```

8 Previous C Versions

For older C versions, there could be a significant compatibility issue for users of implementations that make plain `int` bit-fields unsigned. It may make sense to just say (a) the ambiguous cases are explicitly signed (without removing the option for plain `int` bit-fields to be unsigned in older versions), or (b) leave everything, or at least questions 2 and 3, unspecified.

It might be reasonable to answer question 4 (`_BitInt`) as "no", and so question 5 as "not applicable", for older versions of C. Question 5 is moot for C2Y because we've removed the constraint against `_BitInt(1)`.

9 Poll Questions

1. Adopt this paper for C2Y?
2. Apply this paper to previous versions of the standard?

10 References

- N3783 C2Y working draft
- ISO/IEC 9899:2024 (C23)
- ISO/IEC 14882:2011 (C++11), [class.bit]
- WG14 DR 0315 — *Implementation-defined bit-field types*
- WG21 CWG 739 — *Signedness of bit-field with typedef or template parameter type*
- MISRA C:2025, Rule 6.1
- MISRA C++:2023, Rule 6.1.4
- Defect Report #315, https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_315.htm