N3795: Named Address Space Type Qualifiers for C2Y

Philipp Klaus Krause

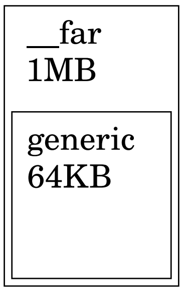
January 22, 2026

We propose bringing named address space type qualifiers from the Embedded C TR into the C2Y standard. This proposal replaces the earlier N3651 and N3723. Compared to N3723 we added highlighting in the proposed changes to make them more readable, fixed an the proposed wording for library changes, acknowledged further alternatives in the "Motivation and Alternatives" section, and provided a wording alternative for only introducing the "access qualifier" concept (the proposed wording for that is a subset of the wording for introducing named address spaces).

0 Introduction

Small architectures often have different address spaces that need to be accessed in different ways. Embedded C (N1275) proposed named address space type qualifiers to designate the address space that an object resides in. When no named address space qualifier is used, the corresponding address space is the generic address space. There might be implementation-defined intrinsic named address spaces, and there might be user-defined named address spaces.

0.1 Example: One larger intrinsic named address space containing the generic address space

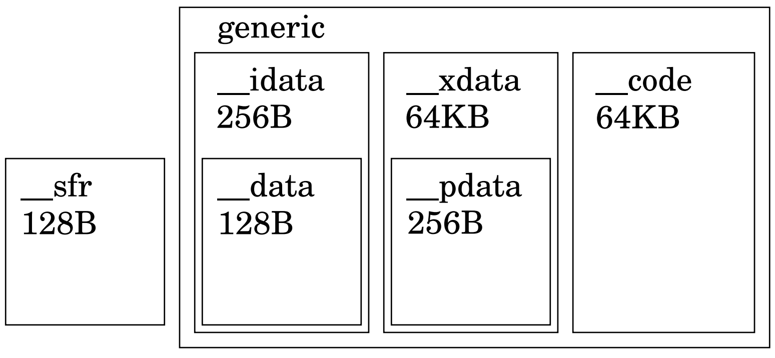


A common case is a small, easy-to-access generic address space suitable for most uses, and a larger address space that is harder to access (e.g. only a few instructions can access it - often, all instructions have an addressing mode for the generic address space, but the larger address space has only a few load/store instructions. Typically, the generic address space is used for most objects (including the stack), while the larger address space holds few large objects explicitly placed there by the programmer using a named address space type qualifier.

Generic pointers tend to be 16 bits wide, while pointers into the intrinsic named address space tend to be 24 or 32 bits wide.

- GCC (a C compiler supporting ISO C23) uses this for the M8C, M16C, and RL78. `__far` is a named address space type qualifier as in N1275.
- SDCC (a C compiler supporting ISO C23 with few gaps) will use this for the Toshiba TLCS-90 and Rabbit microcontrollers in SDCC 4.6.0. In both cases we have a generic address space of 64KB, and a larger 1MB or 16MB `__far` address space. The same approach will potentially be used by SDCC for STM8 in the future. `__far` is a named address space type qualifier as in N1275.
- Dynamic C (a C compiler supporting C90 with few gaps) uses this for the Rabbit. `far` is a type qualifier.
- Cosmic C (a C compiler supporting C99 with gaps) uses this for STM8. `@far` is a type qualifier.
- Raisonance (a C compiler supporting C11 with gaps) uses this for STM8. `far` is a type qualifier.
- IAR (a C compiler supporting C99 with few gaps) uses this for STM8. `__far` and `__huge` can be syntactically placed like a type qualifier, however, they are called "type attributes", and when placed at the very beginning of a declaration, apply to the declared object (i.e. `__far int *p` declares p in `__far` pointing into the generic space, while `int __far *p` declares p in the generic address space pointing into `__far`). `__far` and `__huge` are essentially the same space, but they have different semantics for pointer arithmetic, resulting in `__far` only being usable for objects up to 64KB.

0.2 Example: Complex arrangement of intrinsic named address spaces



The very common MCS-51 architecture has a large number of address spaces that are accessed in different ways. Objects could be anywhere, so generic pointers contain tag bits to designate the address space they point into. The arrangement is so complex that reads and writes via pointers are implemented as support function calls (unless the optimizer can prove that a particular pointer points into a specific address space at the point of access), and thus expensive.

Generic pointers tend to be 24 bits wide, while pointers into any fixed intrinsic named address spaces tend to be 8 or 16 bits wide.

- SDCC (a C compiler supporting ISO C23 with few gaps) uses this for MCS-51. `__data`, `__idata`, `__pdata`, `__xdata`, `__code` are named address space type qualifiers as in N1275.
- IAR (a C compiler supporting ISO C99 with few gaps) uses this for MCS-51. `__data`, `__idata`, `__pdata`, `__xdata`, `__code` can be placed like type qualifiers, however, they are called "type attributes", and when placed at the very beginning of a declaration, apply to the declared object (i.e. `__data int *p` declares p in `__data` pointing into the generic space, while `int __data *p` declares p in the generic address space pointing into `__data`).
- Keil (a C compiler supporting ISO C90 with gaps) uses this for MCS-51. `data`, `idata`, `pdata`, `xdata`, `code` can be placed like type qualifiers, however, they are called "memory type specifiers", and when placed at the very beginning of a declaration, apply to the declared object (i.e. `data int *p` declares p in `data` pointing into the generic space, while `int data *p` declares p in the generic address space pointing into `data`). However, the documentation notes that this non-compliant aspect is an obsolescent feature, and will not be supported in future compiler versions.
- OpenCL, a programming language for heterogeneous computing systems based on ISO C99 with some extensions (including parts of ISO C11) uses this. `__global`, `__local`, `__constant`, `__private` are named address space type qualifiers as in N1275.

0.3 Further examples

Further examples tend to fall in between the two given above, e.g. GCC for AVR, and SDCC for Padauk, HC08, S08, MOS6502 and 65C02. In addition to intrinsic named address spaces, N1275 annex B also mentions user-defined named address spaces. While SDCC does support user-defined named address spaces with the N1275 syntax and semantics at their point of use, the syntax and semantics for declaring them are different.

0.4 Summary of current situation

GCC and SDCC support intrinsic named address space type qualifiers as in N1275. Dynamic C, Cosmic C, and Raisonance support intrinsic named address space type qualifiers as in N1275, except for the identifiers used to designate them (N1275 requires them to start with two underscores or an underscore followed by a capital letter). IAR supports intrinsic named address space type qualifiers as in N1275, except for different semantics when they are placed at the very beginning of a declaration. Keil supports intrinsic named address space type qualifiers, but both exceptions apply. OpenCL has named address space type qualifiers as in N1275.

1 Motivation and Alternatives

Intrinsic named address spaces are the one widely supported part of N1275. There are notable differences between named address space type qualifiers and some of the other qualifiers. Having them in the standard helps avoid changes to qualifiers that would break named address space type qualifiers by not considering them sufficiently (e.g. N3510 with its "all qualifiers except `_Atomic`" wording).

A program where all named address space qualifiers are removed would still be a valid C program, with the same behavior in the abstract machine, though it typically will no longer compile for the original target. Thus, attributes could be considered as an alternative, similar to how tail-call elimination could be expressed via a standard attribute.

Technically, ignoring attributes could even improve portability of the program here: while the program would no longer compile for the original target, it would compile for larger targets with more memory.

On the other hand, qualifiers are what existing implementations and the Embedded C TR use. So bringing named address space type qualifiers into C2Y standardizes existing practice, and helps prevent unintentional breaking of the existing practice. The named address spaces only affect how the hardware accesses objects, but does not affect which values they can have, which goes well with qualifiers being dropped on lvalue conversion (but being kept on pointer targets). However, there is one difference to other qualifiers currently in the standard: for other qualifiers, it is either possible to convert a pointer to a non-qualified object to the qualified one (most qualifiers) or not (`_Atomic`). For named address space qualifiers, this depends on the nesting of the corresponding named address spaces.

Even when going with qualifiers, there are multiple possible options:

- Integrating named address space type qualifiers into the main body of the standard (wording for this option is provided below)
- Integrating named address space type qualifier as a normative annex.
- Minimal approach: leave the named address space type qualifiers themselves out of the standard, but do put the wording about "access qualifiers" into it to acknowledge that there can be qualifiers that are not access qualifiers.

If the named address space qualifiers go into the standard, we again have multiple options:

- Stick closely to the Embedded C TR N1275 (wording for this option is provided below).
- Make some changes. In particular, this has been suggested on the reflector by Joseph Myers (wrt. the behavior of null pointer constants) and Robert Seacord (wording improvements).

2 Named Address Space Qualifiers in the main body of the standard - proposed changes vs. N3685

This is essentially the bits of N1275 for named address spaces rebased from the C99 standard to the C2Y draft N3685.

6.2.1p1: In the list of things an identifier can denote add "- an address space".

6.2.3: Change "- all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants)." to "- all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants, and intrinsic address spaces)."

Insert new section after current 6.2.4:

"Objects are allocated in one or more *address spaces.* A unique *generic address space* always exists. Every address space other than the generic one has a unique name in the form of an identifier. Address spaces other than the generic one are called *named address spaces.* An object is always completely allocated into at least one address space. Unless otherwise specified, objects are allocated in the generic address space.

Some (possibly empty) implementation-defined set of named address spaces are *intrinsic.* The name of an intrinsic address space shall begin with an underscore and an uppercase letter or with two underscores (and hence is a reserved identifier as defined in 7.1.3). There is no declaration for the name of an intrinsic address space in a translation unit; the identifier is *implicitly declared* with a scope covering the entire translation unit.

An implementation may optionally support an implementation-defined syntax for declaring other (non-intrinsic) named address spaces. Each address space (intrinsic or otherwise) exists for the entire execution of the program. If an address space A *encloses* an address space B, then every location (address) within B is also within A. (Either A or B may be the generic address space.) The property of "enclosing" is transitive: if A encloses B and B encloses a third address space C, then A also encloses C. Every address space encloses itself. For every pair of distinct address spaces A and B, it is implementation-defined whether A encloses B.

If one address space encloses another, the two address spaces *overlap*, and their *combined address space* is the one that encloses the other. If two address spaces do not overlap, they are *disjoint*, and no location (address) within one is also within the other. (Thus if two address spaces share a location, one address space must enclose the other.)"

6.2.5p32: Change "Each unqualified type has several *qualified versions* of its type,40) corresponding to the combinations of one, two, or all three of the `const`, `volatile`, and `restrict` qualifiers." to "Each unqualified type has several qualified versions of its type,40) corresponding to the combinations of one, two, or all three of the `const`, `volatile`, and `restrict` qualifiers and all combinations of these three qualifiers with one address space qualifier. (Syntactically, an address space qualifier is an address space name, so there is an address space qualifier for each visible address space name.) The qualifiers `const`, `volatile`, and `restrict` are *access qualifiers*."

Insert new paragraph after 6.2.5p33: "If type T is qualified by the address space qualifier for address space A, then "T is in A". If type T is not qualified by an address space qualifier, then T is in the generic address space. If type T is in address space A, a pointer to T is also a "pointer into A", and the *referenced address space* of the pointer is A."

6.2.6.3p1: Change "A pointer to `void` shall have the same representation and alignment requirements as a pointer to a character type.41) Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types may not have the same representation or alignment requirements." to "A pointer to `void` shall have the same representation and alignment requirements as a pointer to a character type in the same address space.41) Similarly, pointers to differently access-qualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types in the same address space shall have the same representation and alignment requirements as each other. All pointers to union types in the same address space shall have the same representation and alignment requirements as each other. Pointers to other types may not have the same representation or alignment requirements."

6.3.3.3p1: Change "A pointer to `void` can be converted to or from a pointer to any object type. A pointer to any object type can be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer." to "6.3.3.3 A pointer to `void` in any address space can be converted to or from a pointer to any object type. A pointer to any object type in an address space can be converted to a pointer to `void` in any enclosing address space and back again; the result shall compare equal to the original pointer."

6.3.3.3p2: Change "For any qualifier q, a pointer to a non-q-qualified type can be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal." to "For any access qualifier q, a pointer to a non-q-qualified type can be converted to a pointer to the q-qualified version of the type (but with the same address space qualifier, if any); the values stored in the original and converted pointers shall compare equal.

6.3.3.3p4: Change "Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal." to "Conversion of a null pointer to another pointer type yields a null pointer of that type. If the referenced address spaces of the original and converted pointers are disjoint, the behavior is undefined. Any two null pointers whose referenced address spaces overlap shall compare equal."

Insert new paragraph after 6.3.3.3p8: "If a pointer into one address space is converted to a pointer into another address space, then unless the original pointer is a null pointer or the location referred to by the original pointer is within the second address space, the behavior is undefined. (For the original pointer to refer to a location within the second address space, the two address spaces must overlap.)"

6.5.1p7: Change "*The effective* type of an object that is not a byte array, for an access to its stored value, is the declared type of the object.84) If a value is stored into a byte array through an lvalue having a type that is not a byte type, then the type of the lvalue becomes the effective type of the object" to "The *effective type* of an object that is not a byte array, for an access to its stored value, is the declared type of the object, without any address space qualifier.84) If a value is stored into a byte array through an lvalue having a type that is not a byte type, then the type of the lvalue, without any address space qualifier, becomes the effective type of the object"

Insert new paragraph after 6.5.3.6p4: "If the compound literal has automatic storage duration, the type name shall not be qualified by an address space qualifier."

Insert new paragraph after 6.5.7p4: "For subtraction, if the two operands are pointers into different address spaces, the address spaces must overlap."

Insert new paragraph after 6.5.9p3: "If the two operands are pointers into different address spaces, the address spaces must overlap."

Insert new paragraph after 6.5.9p4: "If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same referenced type except with any address-space qualifier removed and any address-space qualifier of the other pointer's referenced type added. (After this conversion, both operands are pointers into the same address space.)"

Insert new paragraph after 6.5.10p3: "If the two operands are pointers into different address spaces, the address spaces must overlap."

6.5.10p6: Change "Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant or has type nullptr_t, they compare equal if the former is a null pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`, the former is converted to the type of the latter." to "Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant or has type nullptr_t, they compare equal if the former is a null pointer. If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same referenced type except with any address space qualifier removed and any address space qualifier of the other pointer's referenced type added. (After this conversion, both operands are pointers into the same address space.) Then, if one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`, the former is converted to the type of the latter."

Insert new paragraph after 6.5.16p4: "If the second and third operands are pointers into different address spaces, the address spaces must overlap."

6.5.16p8: Change "If both the second and third operands are pointers, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands;" to "'If both the second and third operands are pointers, the result type is a pointer to a type qualified with all the access qualifiers of the types referenced by both operands, and qualified with the address-space qualifier for the combined address space of the referenced address spaces of the two operands or with no address-space qualifier if the combined address space is the generic one;"

6.5.17.2p1: Change "both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;" to "both operands are pointers to qualified or unqualified versions of compatible types, the referenced address space of the left encloses the referenced address space of the right, and the referenced type of the left has all the access qualifiers of the referenced type of the right;"

6.5.17.2p1: Change "one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of `void`, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;" to "one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of `void`, the referenced address space of the left encloses the referenced address space of the right, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;"

Insert new paragraph after 6.7.3.2p5: "Within a structure or union specifier, the type of a member shall not be qualified by an address space qualifier."

6.7.4.1: Add to the syntax for type-qualifier: "- address-space-name" and "address-space-name: identifier".

Insert new paragraphs after 6.7.4.1p4: "No type shall be qualified by qualifiers for two or more different address spaces.

The type of an object with automatic storage duration shall not be qualified by an address-space qualifier.

A function type shall not be qualified by an address-space qualifier."

7.16.2.2: Replace both occurrences of "qualified or unqualified" by "access-qualified or unqualified".

3 Alternative: Just introduce the "access qualifier" concept - proposed changes vs. N3685

This is essentially a small subset of the above wording to just introduce the "access qualifier" concept into the main body of the standard, to make it easier for named address space qualifiers to survive (in an annex or a TR/TS or just as implementation extensions).

6.2.5p32: Change "Each unqualified type has several *qualified versions* of its type,40) corresponding to the combinations of one, two, or all three of the `const`, `volatile`, and `restrict` qualifiers." to "Each unqualified type has several *qualified versions* of its type,40) corresponding to the combinations of one, two, or all three of the `const`, `volatile`, and `restrict` qualifiers. The qualifiers `const`, `volatile`, and `restrict` are *access qualifiers*."

6.2.6.3p1: Change "Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements." to "Similarly, pointers to differently access-qualified versions of compatible types shall have the same representation and alignment requirements."

6.3.3.3p2: Change "For any qualifier q, a pointer to a non-q-qualified type can be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal." to "For any access qualifier q, a pointer to a non-q-qualified type can be converted to a pointer to the q-qualified version of the type (but with the same address space qualifier, if any); the values stored in the original and converted pointers shall compare equal.

6.5.16p8: Change "If both the second and third operands are pointers, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands;" to "'If both the second and third operands are pointers, the result type is a pointer to a type qualified with all the access qualifiers of the types referenced by both operands;"

7.16.2.2: Replace both occurrences of "qualified or unqualified" by "access-qualified or unqualified".

J: Add "**J.5.20 Named address spaces**

Identifiers can denote address spaces. Objects are allocated in one or more *address spaces.* Named address spaces are supported via address space qualifiers (which are qualifiers, but not not access qualifiers)."

4 Further changes: library functions - proposed changes vs. N3685

The current wording for varying pointer arguments to some library functions is problematic wrt. pointers to qualified types. Adding named address space type qualifiers to the standard results in more problems, but the embedded C TR contains no wording to deal with the issue. Pointers to `_Atomic` or to a named address space are obviously problematic (since the standard does not require their representations to be compatible). But the same is true for other qualifiers, since these functions are not required to respect any `volatile` or `restrict`. The only qualifier that makes sense on the pointer target in some cases is `const`. The changes proposed here go further than just dealing with named address space type qualifiers, and disallow the use of pointers to further qualifiers.

7.24.6.2p8: Change "the argument shall be a pointer to storage of character type." to "the argument shall be a pointer to storage of unqualified or `const`-qualified character type.", change "the argument shall be a pointer to storage of `wchar_t` type" to "the argument shall be a pointer to storage of unqualified or `const`-qualified `wchar_t` type", "The argument shall be a pointer to `void` or a pointer to a character type." to "The argument shall be a pointer to unqualified or `const`-qualified `void` or a pointer to an unqualified or `const`-qualified character type.", change "The argument shall be a pointer to signed integer" to "The argument shall be a pointer to unqualified or `const`-qualified signed integer".

7.24.6.3p10: Change "If this object does not have an appropriate type" to "If this object does not have an appropriate unqualified type".

7.33.2.2p8: Same changes as 7.24.6.2p8.

7.33.2.3p10: Same changes as 7.24.6.3p10.

5 Straw Polls

- Do we want named address space type qualifiers in C2Y?
- If yes: do we want them in the main body of the standard (as opposed to a new Annex N)?
- If yes: do we want to use the wording above?
- If no: do we want the "access qualifier" concept in C2Y?
- If yes: do we want to use the wording above?

- Do we want to make the changes to the library functions using the wording above?