MISRA C:2012 Amendment 4

Updates for ISO/IEC 9899:2011/2018
Phase 3 — Multi-threading and atomics

March 2023

First published March 2023 by The MISRA Consortium Limited
1 St James Court
Whitefriars
Norwich
Norfolk
NR3 1RU
UK

www.misra.org.uk

ISBN 978-1-911700-03-6 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

MISRA C:2012 Amendment 4

Updates for ISO/IEC 9899:2011/2018 Phase 3 — Multi-threading and atomics

March 2023

MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

Disclaimer

Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and re-use.

Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations.

Foreword

An updated edition of the C Standard, ISO/IEC 9899:2011, commonly referred to as C11, was released just as MISRA C:2012 was being prepared for publication, meaning it arrived too late for the MISRA C Working Group to take it into consideration. Subsequently a further edition, ISO/IEC 9899:2018, commonly referred to as C18, followed.

As the adoption of C11 and then C18 became more widespread, the MISRA C Working Group decided that it was time to address these new editions of the C Standard, support for which is being implemented by means of a series of amendments to MISRA C:2012. To date, the following have been published:

- MISRA C:2012 Amendment 2 *C11 Core* (published February 2020), and
- MISRA C:2012 Amendment 3 *C11/C18 New features* (published October 2022).

This document further amends MISRA C:2012 as required to introduce support for most of the remaining new features introduced by C11 and C18, as well as some additional guidance on existing language features.

We trust that this amendment will be welcomed by the community at large, and will offer confidence to projects and organizations who have held off migrating to C11 or C18.

Andrew Banks FBCS CITP
Chairman, MISRA C Working Group

Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Andrew Banks	LDRA Ltd. and Intuitive Consulting
Dave Banham	BlackBerry Ltd.
Alex Gilding	Perforce Software Inc.
Daniel Kästner	AbsInt Angewandte Informatik GmbH

The MISRA consortium also wishes to acknowledge contributions from the following members of the MISRA C Working Group during the development and review process:

Roberto Bagnara	BUGSENG and University of Parma
Jill Britton	Perforce Software Inc.
Gerlinde Kettl	Vitesco Technologies GmbH
Michal Rozenau	Parasoft Corp.
Chris Tapp	LDRA Ltd. and Keylevel Consultants Ltd.

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

Gaétan Noël	BlackBerry Ltd.
David Ward	HORIBA MIRA Ltd.
Liz Whiting	LDRA Ltd.

DokuWiki was used extensively during the drafting of this document. Our thanks go to all those involved in its development.

Contents

1 Overview

1.1 Applicability

This amendment is intended to be used with MISRA C:2012 (Third Edition, First Revision) [2] as revised and amended by

- MISRA C:2012 Amendment 2 [6],
- MISRA C:2012 Amendment 3 [7], and
- MISRA C:2012 Technical Corrigendum 2 [4]

This amendment is also compatible with MISRA C:2012 (Third Edition) [1] as revised and amended by:

- MISRA C:2012 Amendment 1 [5],
- MISRA C:2012 Amendment 2 [6],
- MISRA C:2012 Amendment 3 [7],
- MISRA C:2012 Technical Corrigendum 1 [3], and
- MISRA C:2012 Technical Corrigendum 2 [4]

1.2 C language updates

This document further amends MISRA C:2012 as follows:

1. To permit the use, with restrictions, of the following ISO/IEC 9899:2011 [12] features:
   - Atomic functions (`<stdatomic.h>`)
   - Multi-threading (`<threads.h>`)
2. To provide further guidance on the use of the following:
   - Small integer constants
   - Unused objects
   - Chained initialization (also revises Rule 9.4)
   - Variably-modified arrays (also revises Rule 18.10)

When using ISO/IEC 9899:2011 [12], use of the following features remains prohibited without the support of a deviation against Rule 1.4:

- Bounds-checking interfaces (Annex K)

Notes:

1. ISO/IEC 9899:2018 [13] incorporates corrigenda applicable to ISO/IEC 9899:2011 [12]. As such, it is functionally equivalent to ISO/IEC 9899:2011 and is therefore also supported through this amendment.

2 New guidance

2.1 Section 7 — Directives

2.1.1 Create new section 7.5 — Concurrency Considerations

Amendment

Add new section 7.5 and associated directives.

AMD4.1 : Add new Section 7.5 for Concurrency Considerations

7.5 Concurrency considerations

AMD4.2 : Add the following new directives in the new section 7.5:

Dir 5.1	There shall be no data races between threads

C11 [Undefined 5, *]

Category Required

Applies to C11

Amplification

Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other, i.e. there is no fixed ordering between the two actions. To prevent data races, objects shared between different threads shall be protected by an appropriate synchronization mechanism.

Rationale

Data races are caused by simultaneous accesses to the same non-atomic object from two different threads `T1` and `T2` where at least one of them is a write access and where the program semantics does not impose a fixed ordering between `T1` and `T2`. There may be legitimate program executions where the access from `T1` is executed before the access from thread `T2`, and vice versa, or where a given access itself is interrupted. Any such data race results in undefined behaviour.

There are several critical scenarios:

- Depending on the timing of the threads, sometimes in a given context the wrong value might be used, leading to unexpected results.
- If a read or write access is implemented by several machine instructions, a pre-emption might occur between these instructions such that inconsistent values might be read or written. As an example, a 64-bit variable read implemented as two 32-bit load instructions might be interrupted after reading the first 32 bits. Then another thread might change the variable value. When the first thread resumes, it reads the second 32-bit half, which now contains a different value than when the first 32 bits of the variable were read.

In general, a data race can cause memory corruption and lead to unexpected, erroneous or erratic behaviour. Data races typically manifest sporadically and are very hard to reproduce.

To prevent such situations, when an object is shared between different threads, it shall be protected by an appropriate synchronization mechanism. To ensure consistent access within a single shared object it can be declared as atomic. A more general solution to ensure consistency of accesses is to introduce critical sections with mutex locks or condition variables.

Note: C library functions may access objects with static or thread storage duration directly or indirectly via the function's arguments. The C library functions *setlocale*, *tmpnam*, *rand*, *srand*, *getenv*, *getenv_s*, *strtok*, *strerror*, *asctime*, *ctime*, *gmtime*, *localtime*, *mbrtoc16*, *c16rtomb*, *mbrtoc32*, *c32rtomb*, *mbrlen*, *mbrtowc*, *wcrtomb*, *mbsrtowcs*, *wcsrtombs* are not guaranteed to be reentrant and may modify objects with static or thread storage duration. To prevent data races explicit synchronization may be required.

Example

The following example exhibits data races on the global variables `x` and `a`. Functions `t1`, `t2`, `t3` and `t4` are executed as concurrent threads `T1`, `T2`, `T3` and `T4` respectively.

Variable `x` is accessed without synchronization, by function `t1` in thread `T1` and by function `t2` in thread `T2`. If executed on a 16-bit machine writing 32-bit values in two chunks of 16 bits, threads `T1` and `T2` might interrupt one another after the first 16 bits of the variable have been written. As a consequence, the two 16-bit halves of variable `x` might be written by different threads, causing unexpected values.

```
int32_t x;
int32_t a=1;
int32_t b;


int32_t t1( void *ignore )  /* Thread T1 entry */
{
  while ( 1 )
  {
    x = -1;  /* Write-write data race with t2. Possible values of x: 0xFFFF0000,
                0x0000FFFF, 0x00000000, 0xFFFFFFFF */
  }
  return 0;
}


int32_t t2( void *ignore )  /* Thread T2 entry */
{
  while ( 1 )
  {
    x = 0;  /* Write-write data race with t1. Possible values of x: 0xFFFF0000,
               0x0000FFFF, 0x00000000, 0xFFFFFFFF */
  }
  return 0;
}
```

A data race on `a` is caused by unprotected accesses by function `t3` in thread `T3` and by function `t4` in thread `T4`. If thread `T3` sees the value of 1 in variable `a`, it will enter the then-part of the conditional statement. At that point, it might be interrupted by thread `T4`, which sets `a` to 0. After resuming, thread `T3` will run into a division by zero.

```
int32_t t3( void *ignore )  /* Thread T3 entry */
{
  while ( 1 )
  {
    if ( a != 0 )   /* Read-write  data race with T4 */
    {
      b += 1/a;     /* Read-write  data race with T4 */
      a = 1;        /* Write-write data race with T4 */
    }
  }
  return 0;
}


int32 t4( void *ignore )  /* Thread T4 entry */
{
  while ( 1 )
  {
    a = 0;          /* Read-write data race with T3  */
  }
  return 0;
}
```

See also

Rule 9.7, Rule 12.6

Dir 5.2 There shall be no deadlocks between threads

C11 [Undefined *]

Category Required

Applies to C11

Amplification

A deadlock occurs when there is a circular chain of threads each of which holding a locked synchronization resource and trying to lock a synchronization resource held by the next element in the chain. To prevent deadlocks, synchronization mechanisms between threads shall not introduce cyclic dependencies.

Rationale

An example for a deadlock between two threads `T1` and `T2` is when `T1` enters the waiting state because it requests a mutex `Ra` which is locked by thread `T2`, and `T2` in turn is waiting for another mutex `Rb` held by thread `T1`.

Possible solutions to avoid deadlocks include locking/unlocking synchronization resources in a fixed global non-cyclic order, or associating synchronization resources with appropriate priorities.

Example

Assume that in the following example functions `t1` and `t2` are executed as concurrent threads `T1` and `T2`. Thread `T1` locks mutex `Ra`, then executes some other code in which it might be interrupted by thread `T2`. Thread `T2` locks mutex `Rb`, executes some other code, and is blocked when attempting to lock mutex `Ra`, which is currently held by thread `T1`. Hence thread `T1` resumes, and eventually reaches the call to `mtx_lock(&Rb)` on which it blocks, because `Rb` is held by `T2`. Then execution is stuck indefinitely because thread `T1` is waiting for thread `T2` and vice versa.

```
mtx_t   Ra;
mtx_t   Rb;


int32_t t1( void *ignore ) /* Thread T1 entry         */
{
  mtx_lock( &Ra );
  ...
  mtx_lock( &Rb );         /* Deadlock may occur here */
  ...
  mtx_unlock( &Rb );
  mtx_unlock( &Ra );
  return 0;
}


int32_t t2(void* ignore)   /* Thread T2 entry         */
{
  mtx_lock( &Rb );
  ...
  mtx_lock( &Ra );         /* Deadlock may occur here */
  ...
  mtx_unlock( &Ra );
  mtx_unlock( &Rb );
  return 0;
}
```

Dir 5.3	There shall be no dynamic thread creation
Category	Required
Applies to	C11

Amplification

Thread creation shall only occur in a well-defined program start-up phase.

Rationale

Uncertainty about the number of threads running at a particular point in time is error prone and reduces analysability. Also the overhead in thread creation and destruction is hard to predict.

Usage of a static thread pool is common practice in operating systems for safety-related systems, e.g. ARINC-653 [45], AUTOSAR [46] and OSEK [47].

Example

```
thrd_t id1;
thrd_t id2;


int32_t t1( void *ignore)         /* Thread T1 entry                            */
{
  ...
  thrd_create( &id2, t2, NULL );  /* Non-compliant, not constrained to start-up */
  ...
}


int32_t t2( void* ignore )        /* Thread T2 entry                            */
{
  ...
}
```

```
void main(void)
{
  thrd_create( &id1, t1, NULL );   /* Compliant                                                     */
  ...
}
```

See also

Dir 4.7

2.2 Section 8 — Rules

2.2.1 New Rule 2.8 — Unused objects

Amendment

Add restrictions on unused objects.

AMD4.3 : Add the following new rule after Rule 2.7:

Rule 2.8 A project should not contain unused object definitions

Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99, C11

Amplification

An object is unused if the definition (and any declarations) can be removed, and the program still compiles.

Rationale

If an object is defined but unused, then it is unclear to a reviewer if the object is redundant or it has been left unused by mistake.

See also

Rule 8.6

2.2.2 New Rule 7.6 — Small integer constants

Amendment

Restrict the use of the small integer constants

AMD4.4 : Add the following new rule after Rule 7.5 (added by AMD3):

Rule 7.6 The small integer variants of the minimum-width integer constant macros shall not be used

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99, C11

Amplification

The minimum-width integer constant macros are of the form `INTn_C(value)` and `UINTn_C(value)`, where *n* is a value corresponding to a type `int_leastn_t`.

Small integer refers to any integer type with width less than that of type *int*.

Rationale

The Standard requires that the minimum-width integer constant macros expand to an integer constant expression suitable for use in `#if` pre-processing directive, and that the type of the expression has the same type as would result from integer promotion. Consequentially many implementations of the small integer macros have opted to simply substitute the macro for the argument. This results in an expression with type *int* and not the type that may have been anticipated by the use of the macro.

Example

```
int main( void )
{
  uint8_t a = UINT8_C( 100 );  /* Non-compliant - typically expands as plain 100
                                                  i.e. as a signed int           */
}
```

The following example shows the impact of the typing conflict:

```
#define M(x) _Generic( (x), uint8_t: fu8, default: fi )(x)

int main( void )
{
  M( UINT8_C( 100 ) );         /* Non-compliant - selects fi, not fu8            */
}
```

See also

Rule 7.5

2.2.3 New Rule 9.6 — Chained initialization

Amendment

Add guidance on chained initialization.

AMD4.5 : Add the following new rule after Rule 9.5:

Rule 9.6	An initializer using chained designators shall not contain initializers without designators
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99, C11

Amplification

A chained designator is a *designator list* that has more than one item, thus specifying an element of a sub-object within the *current object*.

If an aggregate initializer uses designators to specify elements, and any designator in the initializer is chained, every initializer in the entire containing initializer list shall specify an element explicitly using a designator.

This rule applies to initializers for both objects and sub-objects.

Rationale

Using chained designators for selective sub-object designation can make the intent of the initializer clear for some constructs such as sparse matrices. However, combining chained designators with positional initialization is extremely unclear – a human reader cannot easily tell whether the intended *next object* is within the sub-object, or within the same object level from which the designator started lookup. The syntactic brace structure of the initializer list may also no longer match the depth of the selected element, adding to the confusion.

Exception

A braced sub-object initializer may omit designators to specify elements if it does not contain any chained designators, and no chained designators in the containing initializer list specify an element inside it as the *current object*.

Example

```
struct S
{
    int x;
    int y;
};

struct T
{
    int      w;
    struct S s;
    int      z;
};

/* Non-compliant - chained designators and implicit positional initializers mixed */
struct T tt = {
    1,
    .s.x = 2,   /* To a human reader, this looks like .z is being initialized     */
    3,          /* tt is actually initialized as { 1, { 2, 3 }, 0 }               */
};              /* This also violates Rule 9.2                                    */

/* Compliant - allow the y dimension to implicitly initialize to zero             */
struct S aa[5] = {
    [0].x = 1,
    [1].x = 2,
    [2].x = 3,
    [3].x = 4,
    [4].x = 5,
};

/* Compliant - the initializer for [1] is not chained, but is explicit            */
struct S ab[2] = {
    [0].x = 1,
    [1] = { 2, 3 }, /* Compliant by exception:                                    */
};                  /* the positional initializers are inside a braced sub-object */
```

See also

Rule 9.2, Rule 9.4

2.2.4 New Rule 9.7 — Atomic initialization

Amendment

Add guidance on the initialization of atomic objects.

AMD4.6 : Add the following new rule after new Rule 9.6:

Rule 9.7	Atomic objects shall be appropriately initialized before being accessed

C11 [Undefined 5, *]

Category Mandatory

Analysis Undecidable, System

Applies to C11

Amplification

Initialization of atomic objects shall be completed before accessing them.

For objects that do not have static storage duration, initialization shall be included in their declaration using the assignment operator =, or using the Standard Library function *atomic_init()* before any other access.

For objects of static storage duration, the default initialization is sufficient.

Rationale

An atomic object is to be initialized before it is accessed. Concurrent access to the object being initialized, even via an atomic operation, constitutes a data race.

The *atomic_init()* function initializes atomic objects, including any additional state that the implementation might need to carry for the atomic object. However, it does not avoid data races.

Because of the potential initialization of the implementation state, *atomic_init()* cannot be replaced by other access functions, e.g. *atomic_store()*. Initialization of atomic objects inside of threads would impose constraints on thread ordering which are hard to ensure or verify. An explicit protection, e.g. by use of a mutex, would make atomicity unnecessary.

Example

```
_Atomic int32_t g_ai1;          /* Compliant     - default initialization         */

void main( void )
{
  _Atomic int32_t ai1 = 22;     /* Compliant     - directly initialized           */

  _Atomic int32_t ai2;
  ai2 = 777;                    /* Non-compliant - not initialized by atomic_init */

  _Atomic int32_t ai3;
  atomic_init( &ai3, 333);      /* Compliant     - Initialized by atomic_init     */

  /* ------------ */

  _Atomic int32_t ai4;
  thrd_create( &id1, t1, &ai4);

  atomic_init( &ai4, 666);      /* Non-compliant - Initialized after user-thread
                                                   T1 is created                  */

  thrd_join  ( id1, NULL);
}

int32_t t1( t1_paramtype *ptr )
{
  /* accesses g_ai1, ai1, ai2, ai3, ai4 */
}
```

See also

Dir 5.1, Rule 1.5, Rule 9.1, Rule 12.6

2.2.5 Amend Rule 11.3

Amendment

Amend Exception in case of _*Atomic* qualification.

AMD4.7 : Amend the “Headline”:

A cast shall not ...

to

A conversion shall not ...

AMD4.8 : In the first sentence of the “Rationale”, replace:

Casting ...

to

Conversion of ...

AMD4.9 : Amend the “Exception”:

It is permitted to convert a pointer to object type into a pointer to one of the object types *char*, *signed char* or *unsigned char*.

to

It is permitted to convert a pointer to a non-atomic qualified object type into a pointer to one of the object types *char*, *signed char* or *unsigned char*.

2.2.6 Amend Rule 11.8

Amendment

Extend the Rule to cover _*Atomic* qualification.

AMD4.10 : Amend the “Headline”:

A cast shall not remove any *const* or *volatile* from the type pointed to by a pointer

to

A conversion shall not remove any *const*, *volatile* or _*Atomic* qualification from the type pointed to by a pointer

AMD4.11 : In the first sentence of the “Rationale” section remove:

... by using casting ...

AMD4.12 : Add an additional bullet point to the “Rationale”:

Removing an _*Atomic* qualifier might circumvent the lock status of an object and potentially result in memory corruption.

AMD4.13 : In the “Example” section add additional examples:

```
typedef struct s {
  uint8_t a;
  uint8_t b;
} s_t;

int main( void )
{
  _Atomic s_t  astr;
          s_t  lstr = { 7U, 42U };
          s_t *sptr = &astr;         /* Non-compliant - removes _Atomic qualifier */
}
```

AMD4.14 : In the “See also” section append:

Rule 11.10

2.2.7 New Rule 11.10 — The _Atomic qualifier

Amendment

Add restrictions on the *_Atomic* qualifier.

AMD4.15 : Add the following new rule after Rule 11.9

Rule 11.10 The *_Atomic* qualifier shall not be applied to the incomplete type *void*

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C11

Rationale

The C Standard does not explicitly prohibit usage of the type void with the *_Atomic* qualifier. However, it does not provide a guarantee that a pointer to *_Atomic void* has any particular size or alignment requirement, so it cannot be assumed that is the same as for a pointer to an arbitrary type *_Atomic T*, and the behaviour of type conversion between them may be undefined.

Example

```
struct A {
  int32_t _Atomic x;
  int32_t _Atomic y;
};

void main (void)
{
  struct A a1 = { 6, 7 };

  void _Atomic * pav = &a1;     /* Non-compliant */
  void _Atomic * pax = &a1.x;   /* Non-compliant */
}
```

See also

Rule 11.8

2.2.8 New Rule 12.6 — Atomic structures and unions

Amendment

Add restrictions on the use of atomic-related structures and unions.

AMD4.16 : Add the following new rules after Rule 12.5:

Rule 12.6 Structure and union members of atomic objects shall not be directly accessed

C11 [Undefined 42]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C11

Amplification

The C Standard defines the following access functions for atomic objects: *atomic_init()*, *atomic_store()*, *atomic_load()*, *atomic_exchange()*, *atomic_compare_exchange()*.

Accesses to atomic objects of structure and union types shall only be made to the object as a whole, and only using these functions and the assignment operator =. In particular, the `.` and `->` operators shall not be used on atomic objects of structure and union type.

Rationale

The Standard guarantees absence of data races when performing atomic operations on data shared between threads without requiring explicit protection via mutex or condition variables. The operations have to be performed by dedicated access functions which provide an appropriate built-in protection. Direct access to structure or union members of atomic objects circumvents this protection, thus making them vulnerable to data races.

Note: The *atomic_init()* functions does not avoid data races. Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

Example

```
typedef struct s {
  uint8_t a;
  uint8_t b;
} s_t;
_Atomic s_t astr;


sint32_t main(void)
{
  s_t lstr = { 7U, 42U };

  astr.b = 43U;                   /* Non-compliant */

  lstr   = atomic_load( &astr );
  lstr.b = 43U;
  atomic_store( &astr, lstr );   /* Compliant     */

  lstr.a = 8U;
  astr   = lstr;                  /* Compliant     */
}
```

See also

Dir 5.1, Rule 11.4, Rule 9.7

2.2.9 Amend Rule 13.2 — Concurrency

Amendment

Extend the Rule to cover concurrency aspects.

AMD4.17 : Amend the "Headline":

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

to

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders and shall be independent from thread interleaving

AMD4.18 : In the first line of the "Amplification" section, remove:

or within any full expression

AMD4.19 : In the "Amplification" section, amend bullet point 4:

There shall be no more than one modification access with volatile-qualified type;

to

There shall be no more than one modification access with volatile-qualified or atomic type;

AMD4.20 : In the "Amplification" section, add a new bullet point 6:

There shall be no more than one read access to an object with atomic type.

AMD4.21 : In the "Amplification" section, delete the final sentence:

Full expressions are defined in the statements and blocks section of the C Standard.

AMD4.22 : In the "Rationale" section, add a new paragraph after the existing bullet point list:

The atomic types provide assurance that a single read or write access to an atomic object is not subject to interruption or potential interference from other threads. However, that does not prevent two distinct atomic accesses to the same variable by a thread being pre-empted by another thread modifying that variable. On non-atomic variables such interference can only be caused by data races and constitute undefined behaviour. By definition, although there are no data races on atomic variables, such interference is still undesirable.

AMD4.23 : In the "Example" section, append a new example:

In the following example, Thread `T2` might interrupt Thread `T1` while the expression *a - a* is evaluated. Then the first load instruction for `a` loads the value 10, but the second load operation loads the value 7. The compliant solution avoids the problem by storing the value of `a` in a local variable.

```
_Atomic int32_t a;

int32_t t1( void* ignore )    /* Thread T1 entry */
{
   int32_t v1, v2;
   int32_t acopy;

   a     = 10;
   acopy = a;          /* acopy may be either 10 or 7      */

   v1 = a - a;         /* Non-compliant - v1 may be 0 or 3 */
   v2 = acopy - acopy; /* Compliant     - v2 is always 0   */

   return v1 + v2;
}


int32_t t2( void* ignore )    /* Thread T2 entry */
{
  a = 7;

  return a;
}
```

2.2.10 Amend Rule 18.6

Amendment

Extend the scope of the rule to include thread-local objects.

AMD4.24 : Amend the "Headline":

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

to

The address of an object with automatic or thread-local storage shall not be copied to another object that persists after the first object has ceased to exist.

2.2.11 Rule 18.8 and new Rule 18.10 — Guidance on VLAs

Amendment

Focus Rule 18.8 on just variable-length arrays, and exclude variably-modified arrays.

AMD4.25 : In the "Headline" section, replace the headline:

Variable-length array types ...

to

Variable-length arrays ...

AMD4.26 : In the first line of the first paragraph of the "Rationale" section, replace:

Variable-length array types ...

to

Variable-length arrays ...

AMD4.27 : In the third line of the third paragraph of the "Rationale" section, replace:

... in which it is required to be compatible with another array type, possibly itself variable-length, then ...

to

... in which its type is required to be compatible with the type of another array, then ...

AMD4.28 : In the first line of the fifth paragraph of the "Rationale" section, replace:

... variable-length array type ...

to

... variable-length array ...

AMD4.29 : Update the "Example" section:

Delete function `h()` (which now forms part of new Rule 18.10)

AMD4.30 : Update the "See also" section to add:

, Rule 18.10

Amendment

Add guidance on the use of variably-modified array types

AMD4.31 : Add the following new rule after Rule 18.9 (added by AMD3):

Rule 18.10 Pointers to variably-modified array types shall not be used

C99 [Undefined 69, 70], C11 [Undefined 75, 76]

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C99, C11

Amplification

A pointer to a variably-modified array type shall not be used in the declaration of any object or parameter.

A parameter declared to have an array type is not a pointer-to-array type (unless it is an array of arrays), because it is rewritten to a pointer to the element type.

Rationale

Compatibility between array types requires the size specifiers for the pointed-to arrays to have equal values. However, for variably-modified array types this cannot be determined at compile-time.

If two pointers to array types are used in any way that requires them to be compatible (such as assignment), and the size specifiers for the pointed-to array are not the same, the behaviour is undefined. This is undecidable in general, effectively leaving *all* such operations untyped.

Example

```
/* Non-compliant */
void f1 (uint16_t n, uint16_t (* a) [n])
{
  uint16_t ( *p )[ 20 ];
  p = a;   /* undefined unless n == 20, but types always assumed compatible */
}

/* Compliant */
void f2 (uint16_t n, uint16_t a[n])
{
  uint16_t * p;
  p = a;   /* pointed-to type is not variably-modified, always well-defined */
}
```

See also

Rule 18.8

2.2.12 New Rule 21.25 — Atomic functions

Amendment

Add restrictions on the use of atomic-related Standard Library functions.

AMD4.32 : Add the following new rule after Rule 21.24 (added by Amendment 3):

Rule 21.25 All memory synchronization operations shall be executed in sequentially consistent order

C11 [Undefined *]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C11

Amplification

The Standard provides an enumerated type *memory_order* to specify the behaviour of memory synchronization operations. Only the memory order *memory_order_seq_cst* shall be used.

The following library functions implicitly use memory ordering *memory_order_seq_cst*:

- *atomic_store()*
- *atomic_load()*
- *atomic_exchange()*
- *atomic_compare_exchange_strong()*
- *atomic_compare_exchange_weak()*
- *atomic_fetch_add()*
- *atomic_fetch_sub()*
- *atomic_fetch_or()*
- *atomic_fetch_xor()*
- *atomic_fetch_and()*
- *atomic_flag_test_and_set()*
- *atomic_flag_clear()*

For each of these functions, there exists an alternate version with the function name ending in *_explicit()*, which takes an explicit *memory_order* parameter. The functions ending in *_explicit()* shall only be called with the enumeration *memory_order_seq_cst* as the *memory_order* parameter.

Also the following functions shall only be called with the enumeration *memory_order_seq_cst* as the *memory_order* parameter:

- *atomic_thread_fence()*
- *atomic_signal_fence()*

Rationale

The Standard defines *memory_order_seq_cst* as the default memory order for objects with atomic types. This ordering is fully defined in the C Standard and enables sequential consistency. The behaviour of other memory orders is non-portable, as it depends on hardware architecture and compiler.

For *memory_order_relaxed*, no operation orders memory. Usage of *memory_order_relaxed* can cause unintuitive behaviour and is error-prone.

Many of those library functions listed above impose restrictions on the memory order allowed, e.g. it is undefined behaviour if the *atomic_store* generic function is called with a *memory_order_acquire*, *memory_order_consume*, or *memory_order_acq_rel* order argument. In case of non-compliant usage, compilers may show warnings but still generate code.

Example

```
typedef struct s {
  uint8_t a;
  uint8_t b;
} s_t;
_Atomic s_t astr;


void main( void )
{
  s_t lstr = {7, 42};

  atomic_init( &astr, lstr );

  lstr = atomic_load( &astr );                                /* Compliant     */
  lstr = atomic_load_explicit( &astr, memory_order_relaxed ); /* Non-compliant */

  lstr.b = 43;
  atomic_store_explicit( &astr, lstr, memory_order_release ); /* Non-compliant */
}
```

See also

Dir 4.13

2.2.13 New Rules 21.26 — Mutex functions

Amendment

Add restrictions on the use of mutex Standard Library functions.

AMD4.33 : Add the following new rule after new Rule 21.25:

Rule 21.26 The Standard Library function *mtx_timedlock()* shall only be invoked on mutex objects of appropriate mutex type

C11 [Undefined *]

Category Required

Analysis Undecidable, System

Applies to C11

Amplification

The first argument of the Standard Library function *mtx_timedlock()* shall be a mutex object of mutex type `mtx_timed` or `( mtx_timed | mtx_recursive )`.

Rationale

Calling the function *mtx_timedlock()* on a mutex object that does not support timeout is undefined behaviour.

Example

```
mtx_t Ra;
mtx_t Rb;
mtx_t Rc;
struct timespec *ts;


void main( void )
{
  mtx_init( &Ra, mtx_plain                 );
  mtx_init( &Rb, mtx_timed                 );
  mtx_init( &Rc, mtx_timed | mtx_recursive );
  ...
}


int32_t t1( void* ignore )
{
  ...
  mtx_timedlock( &Ra, ts );  /* Non-compliant */
  mtx_timedlock( &Rb, ts );  /* Compliant     */
  mtx_timedlock( &Rc, ts );  /* Compliant     */
  ...
}
```

2.2.14 New Rules 22.11-22.20 — Threads

Amendment

Add guidance on the use of threads.

AMD4.34 : Add the following new rules after Rule 22.10:

Rule 22.11 A thread that was previously either joined or detached shall not be subsequently joined nor detached

C11 [Undefined *]

Category Required

Analysis Undecidable, System

Applies to C11

Rationale

Invoking *thrd_detach()* or *thrd_join()* on a thread that has been previously detached or joined is undefined behaviour.

Example

```
void main( void )
{
  thrd_t id1, id2, id3, id4;

  thrd_create( &id1, t1, NULL );
  thrd_create( &id2, t2, NULL );
  thrd_create( &id3, t3, NULL );
  thrd_create( &id4, t4, NULL );

  thrd_join  ( id1, NULL ); /* Compliant                           */
  thrd_join  ( id1, NULL ); /* Non-compliant - already joined      */

  thrd_detach( id2 );       /* Compliant                           */
  thrd_detach( id2 );       /* Non-compliant - already detached */

  thrd_join  ( id3, NULL ); /* Compliant                           */
  thrd_detach( id3 );       /* Non-compliant - already joined      */

  thrd_detach( id4 );       /* Compliant                           */
  thrd_join  ( id4, NULL ); /* Non-compliant - already detached */
}
```

Rule 22.12 Thread objects, thread synchronization objects, and thread-specific storage pointers shall only be accessed by the appropriate Standard Library functions

C11 [Undefined *]

Category Mandatory

Analysis Undecidable, System

Applies to C11

Amplification

Thread objects shall exclusively be accessed via the Standard Library functions *thrd_create()*, *thrd_detach()*, *thrd_join()*, and *thrd_equal()*.

Mutex objects shall exclusively be accessed via the Standard Library functions *mtx_destroy()*, *mtx_init()*, *mtx_lock()*, *mtx_trylock()*, *mtx_timedlock()*, *mtx_unlock()*, *cnd_wait()*, and *cnd_timedwait()*.

Condition variables shall exclusively be accessed via the Standard Library functions *cnd_broadcast()*, *cnd_destroy()*, *cnd_init()*, *cnd_signal()*, *cnd_wait()*, and *cnd_timedwait()*.

Thread-specific storage pointers shall exclusively be accessed by the Standard Library functions *tss_create()*, *tss_delete()*, *tss_get()*, and *tss_set()*.

Rationale

Thread objects and thread synchronization objects are expected to be unique for the corresponding thread and synchronization resources.

Thread-specific storage pointers are identified by unique keys. Any direct manipulation (copy, assignment, etc.) may result in undefined behaviour. The *tss_delete()*, *tss_get()* and *tss_set()* functions shall only be called with a value for key that was returned by a call to *tss_create()*, otherwise the behaviour is undefined.

Example

```
mtx_t  Ra;
mtx_t  Rb;
thrd_t id1;
thrd_t id2;
tss_t  key;


int32_t t1( void *ignore )
{
  mtx_lock( &Ra );
  int32_t val;
  if ( id1 == id2 )                    /* Non-compliant - use thrd_equal()          */
  {
    Rb = Ra;                           /* Non-compliant                             */
    memcpy(&Rb, &Ra, sizeof(mtx_t));   /* Non-compliant                             */
  }

  if ( thrd_equal( id1, id2 ) )        /* Compliant                                 */
  {
    ...
  }
  key++;                               /* Non-compliant - explicit manipulation of
                                                          TSS pointer                */
  tss_set( key, &val );                /* Undefined, value of key not returned by
                                                          tss_create()               */
}


void main( void )
{
  mtx_init    ( &Ra,  mtx_plain );
  mtx_init    ( &Rb,  mtx_plain );
  tss_create  ( &key, NULL      );
  thrd_create( &id1, t1, NULL   );
  thrd_create( &id2, t1, NULL   );
  ...
}
```

See also

Rule 11.5, Rule 22.20

Rule 22.13 Thread objects, thread synchronization objects and thread-specific storage pointers shall have appropriate storage duration

C11 [Undefined 9, 10, 11]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C11

Amplification

Objects of type *thrd_t*, *mtx_t*, *cnd_t*, and *tss_t* shall not have automatic storage duration nor thread storage duration.

Rationale

Determining the lifetime of non-static objects which depend on thread execution state is difficult and error-prone. In particular, sharing objects of automatic storage duration between threads and using them to control concurrent execution can cause undefined behaviour due to accessing them outside of their lifetime.

Usage of a static pool of synchronization resources is common practice in many safety-related operating systems, including ARINC-653 [45], AUTOSAR [46] and OSEK [47].

Example

```
mtx_t Ra;                    /* Compliant    */


int32_t t1( void *ptr )        /* Thread entry */
{
  ...
  mtx_lock   ( &Ra);
  mtx_lock   ( (mtx_t*)ptr );    /* Lifetime of Rb might have ended
                                    ... pointer might be dangling    */
  ...
  mtx_unlock( (mtx_t*)ptr );    /* Lifetime of Rb might have ended
                                    ... pointer might be dangling    */
  mtx_unlock( &Ra);
}


void main( void )
{
  thrd_t  id1;                 /* Non-compliant                    */
  mtx_t   Rb;                  /* Non-compliant                    */

  mtx_init   ( &Ra, mtx_plain );
  mtx_init   ( &Rb, mtx_plain );
  thrd_create( &id1, t1,  &Rb  );
}
```

Rule 22.14 Thread synchronization objects shall be initialized before being accessed

C11 [Undefined 9]

Category Mandatory

Analysis Undecidable, System

Applies to C11

Amplification

Before being accessed, mutex objects shall be initialized by calling *mtx_init()*, and condition variables by calling *cnd_init()*.

The second parameter of *mtx_init()* shall be either *mtx_plain*, *mtx_timed*, *(mtx_plain | mtx_recursive)*, or *(mtx_timed | mtx_recursive)*.

Rationale

Mutex objects have to be explicitly created by calling function *mtx_init()*, and condition variables have to be explicitly created by calling function *cnd_init()*.

Invoking *mtx_init()* with a different value of its type parameter than listed above is undefined behaviour.

Initializing all synchronization objects before creating the threads accessing them is a deterministic way to prevent threads from accessing synchronization objects with indeterminate state.

Example

```
mtx_t Ra;
mtx_t Rb;
mtx_t Rc;


int32_t t1( void *ignore )      /* Thread T1 entry                               */
{
  mtx_init( &Rb, mtx_plain );   /* Non-compliant - T2 may have already accessed Rb */
  ...
  /* Subsequently locks/unlocks Ra, Rb, Rc */
}


int32_t t2( void *ignore )
{
  /* locks/unlocks Ra, Rb, Rc */
}


thrd_t id1, id2;

void main(void)
{
  mtx_init   ( &Ra, mtx_plain ); /* Compliant */

  thrd_create( &id1, t1, NULL );
  thrd_create( &id2, t2, NULL );

  mtx_init   ( &Rc, mtx_plain );  /* Non-compliant - T1/T2 may have already
                                                     accessed Rc            */

  thrd_join  ( id1, NULL );
  thrd_join  ( id2, NULL );

  mtx_destroy( &Ra );
  mtx_destroy( &Rb );
  mtx_destroy( &Rc );
}
```

See also

Dir 4.7

Rule 22.15 Thread synchronization objects and thread-specific storage pointers shall not be destroyed until after all threads accessing them have terminated

C11 [Undefined 9, 10, *]

Category Required

Analysis Undecidable, System

Applies to C11

Rationale

The Standard Library function *mtx_destroy(mtx)* releases all resources used by the mutex pointed to by *mtx*. Destroying a mutex which is still locked by some thread results in undefined behaviour, as the C Standard expects no threads to be blocked by a mutex when it is destroyed.

The Standard Library function *tss_delete(key)* releases all resources used by the thread-specific storage identified by *key*. Calling the *tss_delete()*, *tss_get()* or *tss_set()* functions after the thread commenced executing destructors results in undefined behaviour.

Calling the Standard Library function *cnd_destroy()*, on a condition variable on which a thread is currently waiting, results in undefined behaviour.

These problems are avoided by only destroying synchronization resources and deleting thread-specific storage after all threads accessing them have terminated (or not at all).

Example

```
mtx_t   Ra;
mtx_t   Rb;
tss_t   key1;
tss_t   key2;
thrd_t  id1;
thrd_t  id2;


int32_t t1( void *ignore )   /* Thread T1 entry */
{
  /*
  ** locks/unlocks Ra, Rb
  ** accesses thread-specific storage pointed to by key1, key2
  */

  tss_delete( key1 );        /* Non-compliant - might still be accessed from T2   */
}


int32_t t2( void *ignore )   /* Thread T2 entry */
{
  /*
  ** locks/unlocks Ra, Rb
  ** accesses thread-specific storage pointed to by key1, key2
  */

  mtx_destroy( &Rb );        /* Non-compliant - T1 might still access Rb          */
}


void main( void )
{
  mtx_init    ( &Ra, mtx_plain );
  mtx_init    ( &Rb, mtx_plain );

  tss_create  ( &key1, NULL    );
  tss_create  ( &key2, NULL    );

  thrd_create( &id1, t1, NULL );
  thrd_create( &id2, t2, NULL );

  spendSomeTime();

  tss_delete  ( key2 );       /* Non-compliant - might still be accessed by t1, t2 */

  thrd_join   ( id1, NULL );
  thrd_join   ( id2, NULL );

  mtx_destroy( &Ra       );  /* Compliant                                         */
  tss_delete  ( key1 );       /* Compliant                                         */
}
```

See also

Rule 22.1

Rule 22.16 All mutex objects locked by a thread shall be explicitly unlocked by the same thread

C11 [Undefined *]

Category Required

Analysis Undecidable, System

Applies to C11

Amplification

If a mutex object *mtx* is locked by *mtx_lock(mtx)* at a program point *p* there shall be an explicit *mtx_unlock(mtx)* for mutex object *mtx* on all programs paths reachable from *p* before exiting the thread.

Rationale

When a thread terminates without releasing a lock, that lock may be held for indeterminate time. If the life range of a mutex object ends while there are threads waiting for it the behaviour is undefined.

Destroying a mutex on which threads are waiting is undefined behaviour.

Note: it is good practice to unlock mutexes in the same function and under the same control dependences in which they have been locked.

Example

```
mtx_t Ra;
mtx_t Rb;


int32_t t1( void *ignore )  /* Thread 1 */
{
  bool_t b;

  mtx_lock  ( &Ra );  /* Compliant                                  */
  mtx_unlock( &Ra );

  mtx_lock  ( &Rb );  /* Non-compliant - unlock missing on one path */
  if ( b )
  {
    mtx_unlock( &Rb );
  }
  return 0;
}
```

See also

Dir 4.13, Rule 22.1

Rule 22.17 No thread shall unlock a mutex or call *cnd_wait()* or *cnd_timedwait()* for a mutex it has not locked before

C11 [Undefined *]

Category	Required
Analysis	Undecidable, System
Applies to	C11

Amplification

A mutex shall only be unlocked by a thread if it has been locked by that thread before.

The *cnd_wait()* and *cnd_timedwait()* functions shall only be called by a thread on a mutex that is locked by that thread.

Rationale

Unlocking a mutex which has not been locked by the calling thread is undefined behaviour. Calling *cnd_wait()* or *cnd_timedwait()* with mutex argument *mtx* requires that the mutex pointed to by *mtx* be locked by the calling thread.

Example

```
mtx_t Ra;
mtx_t Rb;
cnd_t Cnd1;
cnd_t Cnd2;


int32_t t1( void *ignore )    /* Thread 1 */
{
  mtx_lock  ( &Ra );
  mtx_unlock( &Ra );          /* Compliant                                      */

  mtx_unlock( &Ra );          /* Non-compliant - mutex is not locked            */

  cnd_wait  ( &Cnd1, &Ra );   /* Non-compliant - mutex is not locked            */

  mtx_unlock( &Rb);           /* Non-compliant - mutex either not locked, or
                                                ... is locked by different thread */

  cnd_wait  ( &Cnd2, &Rb );   /* Non-compliant - mutex either not locked, or
                                                ... is locked by different thread */

  return 0;
}


int32_t t2( void *ignore )    /* Thread 2 */
{
  mtx_lock   ( &Rb );
  doSomething();
  mtx_unlock ( &Rb );         /* Compliant                                      */
  return 0;
}
```

See also

Dir 4.13, Rule 22.1, Rule 22.18

Rule 22.18 Non-recursive mutexes shall not be recursively locked

C11 [Undefined *]

Category Required

Analysis Undecidable, System

Applies to C11

Amplification

A non-recursive mutex shall only be locked by a thread if it has not already been locked by that before.

Rationale

It is undefined behaviour if a non-recursive mutex is recursively locked by the calling thread. If the thread also attempts to unlock the mutex twice, the second call to *mtx_unlock()* will also result in undefined behaviour, since the mutex then will already be unlocked.

Example

```
mtx_t Ra;
mtx_t Rb;


int32_t t1( void *ignore )  /* Thread 1 */
{
  mtx_lock  ( &Rb );  /* Compliant                                        */
  mtx_lock  ( &Rb );  /* Compliant - Rb is recursive                      */
  mtx_unlock( &Rb );  /* Rb still locked                                  */
  mtx_unlock( &Rb );  /* Rb gets unlocked                                 */

  mtx_lock  ( &Ra );  /* Compliant                                        */
  mtx_lock  ( &Ra );  /* Non-compliant - undefined behaviour, deadlock possible */
  mtx_unlock( &Ra );  /* If reachable (i.e. no deadlock), Ra gets unlocked      */
  mtx_unlock( &Ra );  /* Undefined behaviour if reachable                 */

  return 0;
}


thrd_t id1;
thrd_t id2;

int32_t main(void)
{
  mtx_init    ( &Ra, mtx_plain     );
  mtx_init    ( &Rb, mtx_recursive );
  thrd_create( &id1, t1, NULL      );
  ...
}
```

See also

Dir 4.13, Rule 22.1, Rule 22.17

Rule 22.19 A condition variable shall be associated with at most one mutex object

C11 [Undefined *]

Category Required

Analysis Undecidable, System

Applies to C11

Rationale

If the same condition variable is used with different mutex objects by two threads, it is undefined which mutex will be unlocked upon signalling the condition variable.

Example

```
mtx_t Ra;
mtx_t Rb;
cnd_t Cnd;


int32_t t1(void *ignore )
{
  mtx_lock  ( &Ra       );
  cnd_wait  ( &Cnd, &Ra );    /* Non-compliant - t2 uses Cnd with Rb  */
  mtx_unlock( &Ra       );
  return 0;
}


int32_t t2(void *ignore )
{
  mtx_lock  ( &Rb       );
  cnd_wait  ( &Cnd, &Rb );    /* Non-compliant - t1 uses Cnd with Ra  */
  mtx_unlock( &Rb       );
  return 0;
}


int32_t t3(void* ignore)
{
  cnd_signal( &Cnd );         /* Unblocks one of Ra and Rb...
                                 ... unclear whether t1 or t2 resumes */
  return 0;
}
```

Rule 22.20 Thread-specific storage pointers shall be created before being accessed

C11 [Undefined 9, *]

Category Mandatory

Analysis Undecidable, System

Applies to C11

Amplification

Objects of type *tss_t* shall be explicitly created by *tss_create()* before being accessed.

Rationale

Thread-specific storage pointers have to be explicitly created before accessing them. Creating them inside of threads creates dependencies on thread execution and ordering which are hard to maintain and check. Creating them before creating the threads accessing them is a deterministic way to prevent threads from accessing thread-specific storage pointers with indeterminate state.

Example

```
tss_t   key1;
tss_t   key2;
thrd_t  id1;
thrd_t  id2;
int32_t g1;
int32_t g2;



int32_t t2( void *ignore )      /* Thread t2 entry */
{
  tss_create( &key1, NULL );    /* Non-compliant - thread t1 might already have
                                                   tried to access key1             */
}


int32_t t1( void *ignore )      /* Thread t1 entry */
{
  tss_set    ( key1, &g1 );   /* Non-compliant - might not yet be created */
  tss_set    ( key2, &g2 );   /* Compliant                                 */

  int32_t *v1 = tss_get( key1 );
  int32_t *v2 = tss_get( key2 );

  *v1 = computeG1();
  *v2 = computeG2();
}


void main( void )
{
  int32_t i;

  tss_create( &key2, NULL );    /* Compliant */

  thrd_create( &id1, t1, NULL );
  thrd_create( &id2, t2, NULL );

}
```

See also

Dir 4.13

3 Technical Corrigenda

3.1.1 Update section 6.9 — Presentation of the guidelines

Amendment

Add new explanations of the “Example” and “See also” sections:

AMD4.35 : Immediately before the paragraph commencing “The supporting text is not...” insert:

Within the supporting text, there may be a heading titled “Example”, followed by code snippets demonstrating the application of the guideline. These code snippets may be incomplete, for the sake of brevity (for example, an *if* statement without its body, or the omission of function call return value checking).

Within the supporting text, there may be a heading titled “See also”, followed by a list of other guidelines which are related to or interact with the guideline.

AMD4.36 : Remove the existing Note:

Note: where code is quoted ... brevity.

3.1.2 Amend Rule 2.2 and Rule 2.7 — Inconsistent headlines

Amendment

Amend rule headlines to align with other Rule 2.x headlines

AMD4.37 : Amend the Rule 2.2 “Headline”:

There shall be no dead code

to

A project shall not contain dead code

AMD4.38 : Amend the Rule 2.7 “Headline”:

There should be no unused parameters in functions

to

A function should not contain unused parameters

3.1.3 Amend Rule 3.1 — URIs in comments

Amendment

Add an explicit exception for Uniform Resource Identifiers (URIs)

AMD4.39 : In the “Exception” section, number the existing exception as 2

AMD4.40 : In the “Exception” section, add a new exception:

1. *Uniform resource identifiers*, of the form *{scheme}*`://`*{path}*, are permitted within comments.

AMD4.41 : In the “Example” section, add a new example:

The following example demonstrates the use of a URI in a comment, and is compliant by exception 1.

```
/*
** The MISRA C:2012 example suite can be found at
** https://gitlab.com/MISRA/MISRA-C/MISRA-C-2012
*/
```

3.1.4 Amend Rule 8.6 — Missing "See also"

Amendment

Add a “See also” omitted from AMD3

AMD4.42 : Add a new “See also” section:

See also

Rule 8.15

3.1.5 Amend Rule 8.9 — "Declared" not "defined"

Amendment

“Declared” should be used instead of “defined”

AMD4.43 : In the “Headline” section, replace:

... defined ...

with

... declared ...

AMD4.44 : At the start of the first paragraph of the “Rationale” section, replace:

Defining ...

with

Declaring ...

AMD4.45 : In the second paragraph of the “Rationale” section, replace:

... defined ...

with

... declared ...

AMD4.46 : In the preamble to the second example in the “Example” section, replace:

... defined ...

with

... declared ...

3.1.6 Amend Rule 9.4 — Designated initializers

Amendment

Clarify the guidance on the use of *designated initializers*.

AMD4.47 : In the "Amplification" section, replace the second paragraph with:

An aggregate initializer shall not contain two designators that refer to the same sub-object. An aggregate initializer shall not allow the *current object* to implicitly initialize an element that has been initialized previously in the initializer list.

AMD4.48 : In the "Rationale" section, replace the first paragraph of with:

The provision of *designated initializers* allows the naming of the components of an aggregate (structure or array) or of a union to be initialized within an initializer list and allows the object's elements to be initialized in any order by specifying the array indices or structure member names they apply to (elements having no initialization value assume the default for uninitialized objects).

A designator can specify elements to be initialized in a different syntactic sequence from their order within the object layout. An initializer without a designator will always initialize the *next subobject* within the object layout.

Care is required when using *designated initializers* since the initialization of object elements can be inadvertently repeated. The C Standard specifies that the value produced by the syntactically-last initializer referring to an element in the list is used, overriding any preceding initializers for that element. The Standard leaves unspecified whether overridden initializers are evaluated, and therefore whether or not any *side effects* in the initializing expressions occur or not. This is not listed in Annex J of the C Standard.

AMD4.49 : In the "Example" section, append the following additional example:

```
/*
 * Positional initializer element values can overwrite earlier ones
 * if preceded by a designated element out of sequence
 * Non-compliant - s4 is 1, 4, 3, 0
 */
struct mystruct s4 = { .b = 2, .c = 3, .a = 1, /* b */ 4 };
```

AMD4.50 : Add a "See also" section:

See also

Rule 9.6

3.1.7 Amend Rule 10.1 and Rule 18.3 — "Expressions" not "objects"

Amendment

"Expressions" should be used instead of "objects".

AMD4.51 : In the "Exception" section of Rule 10.1, in Exception 2, replace:

objects

with

expressions

AMD4.52 : In the “Headline” section of Rule 18.3, replace:

objects of pointer type

with

expressions of pointer type

4 Consequential amendments

4.1 Section 8 — Rules

4.1.1 Amend Rule 1.4 — General restrictions

Amendment

Remove the general restriction on features covered by this amendment.

AMD4.53 : Delete the bullet point relating to the `<stdatomics.h>` header file.

AMD4.54 : Delete the bullet point relating to the `<threads.h>` header file.

4.1.2 Amend Rule 7.5 — Small integer constants

Amendment

Add a "See also" section, with a reference to new Rule 7.6

AMD4.55 : Add a "See also" section:

See also

Rule 7.6

4.1.3 Rule 9.1 — Initialization

Amendment

Extend rule to exclude *atomic* initialization.

AMD4.56 : In the "Amplification" section, append a new paragraph:

This rule does not apply to *_Atomic* qualified objects, which are covered by Rule 9.7.

AMD4.57 : Update the "See also" section to add in sequence:

Rule 9.7,

4.1.4 Amend Rule 9.2 — Aggregate initializers

Amendment

Add a "See also" section, with a reference to new Rule 9.6

AMD4.58 : Add a "See also" section:

See also

Rule 9.6

4.2 Section 9 — References

4.2.1 Insert new references

Amendment

Insert new references to the end of the existing references list.

AMD4.59 : Insert Reference 44 (RFC 3986):

44: RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*,
The Internet Society, 2005
Available from https://www.ietf.org/rfc/rfc3986.txt

AMD4.60 : Insert Reference 45 (ARINC 653):

45: ARINC 653, *Avionics Application Software Standard Interface*,
Aeronautical Radio Inc., https://aviation-ia.sae-itc.com/standards/

AMD4.61 : Insert Reference 46 (AUTOSAR):

46: *AUTomotive Open System ARchitecture* (AUTOSAR), https://www.autosar.org

AMD4.62 : Insert Reference 47 (OSEK/VDX):

47: *OSEK/VDX Operating System*,
Version 2.2.3., 2005

4.3 Appendix A — Summary of Guidelines

AMD4.63 : Update existing entries, as follows:

Guideline	Category	Headline
Rule 2.2	Required	A project shall not contain dead code
Rule 2.7	Advisory	A function should not contain unused parameters
Rule 8.9	Advisory	An object should be declared at block scope if its identifier only appears in a single function
Rule 11.3	Required	A conversion shall not be performed between a pointer to object type and a pointer to a different object type
Rule 11.8	Required	A conversion shall not remove any *const*, *volatile* or *_Atomic* qualification from the type pointed to by a pointer
Rule 13.2	Required	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders and shall be independent from thread interleaving
Rule 18.6	Required	The address of an object with automatic or thread-local storage shall not be copied to another object that persists after the first object has ceased to exist
Rule 18.8	Required	Variable-length arrays shall not be used

AMD4.64 : Insert new entries, in the appropriate places, as follows:

Guideline	Category	Headline
Dir 5.1	Required	There shall be no data races between threads
Dir 5.2	Required	There shall be no deadlocks between threads
Dir 5.3	Required	There shall be no dynamic thread creation
Rule 2.8	Advisory	A project should not contain unused object definitions
Rule 7.6	Required	The small integer variants of the minimum-width integer constant macros shall not be used
Rule 9.6	Required	An initializer using chained designators shall not contain initializers without designators
Rule 9.7	Mandatory	Atomic objects shall be appropriately initialized before being accessed
Rule 11.10	Required	The *_Atomic* qualifier shall not be applied to the incomplete type *void*
Rule 12.6	Required	Structure and union members of atomic objects shall not be directly accessed
Rule 18.10	Mandatory	Pointers to variably-modified array types shall not be used
Rule 21.25	Required	All memory synchronization operations shall be executed in sequentially consistent order
Rule 21.26	Required	The Standard Library function *mtx_timedlock()* shall only be invoked on mutex objects of appropriate mutex type
Rule 22.11	Required	A thread that was previously either joined or detached shall not be subsequently joined nor detached
Rule 22.12	Mandatory	Thread objects, thread synchronization objects, and thread-specific storage pointers shall only be accessed by the appropriate Standard Library functions
Rule 22.13	Required	Thread objects, thread synchronization objects and thread-specific storage pointers shall have appropriate storage duration
Rule 22.14	Mandatory	Thread synchronization objects shall be initialized before being accessed
Rule 22.15	Required	Thread synchronization objects and thread-specific storage pointers shall not be destroyed until after all threads accessing them have terminated
Rule 22.16	Required	All mutex objects locked by a thread shall be explicitly unlocked by the same thread
Rule 22.17	Required	No thread shall unlock a mutex or call *cnd_wait()* or *cnd_timedwait()* for a mutex it has not locked before
Rule 22.18	Required	Non-recursive mutexes shall not be recursively locked
Rule 22.19	Required	A condition variable shall be associated with at most one mutex object
Rule 22.20	Mandatory	Thread-specific storage pointers shall be created before being accessed

4.4 Appendix B — Guidelines attributes

AMD4.65 : Insert new entries, in the appropriate places, as follows:

Guideline	Category	Applies to	Analysis
Dir 5.1	Required	C11	
Dir 5.2	Required	C11	
Dir 5.3	Required	C11	
Rule 2.8	Advisory	C90, C99, C11	Decidable, System
Rule 7.6	Advisory	C99, C11	Decidable, Single Translation Unit
Rule 9.6	Required	C99, C11	Decidable, Single Translation Unit
Rule 9.7	Mandatory	C11	Undecidable, System
Rule 11.10	Required	C11	Decidable, Single Translation Unit
Rule 12.6	Required	C11	Decidable, Single Translation Unit
Rule 18.10	Mandatory	C99, C11	Decidable, Single Translation Unit
Rule 21.25	Required	C11	Decidable, Single Translation Unit
Rule 21.26	Required	C11	Undecidable, System
Rule 22.11	Required	C11	Undecidable, System
Rule 22.12	Mandatory	C11	Undecidable, System
Rule 22.13	Required	C11	Decidable, Single Translation Unit
Rule 22.14	Mandatory	C11	Undecidable, System
Rule 22.15	Required	C11	Undecidable, System
Rule 22.16	Required	C11	Undecidable, System
Rule 22.17	Required	C11	Undecidable, System
Rule 22.18	Required	C11	Undecidable, System
Rule 22.19	Required	C11	Undecidable, System
Rule 22.20	Mandatory	C11	Undecidable, System

4.5 Appendix H — Undefined and critical unspecified behaviour

4.5.1 Appendix H.1 — Undefined behaviour

AMD4.66 : Replace the the following rows in the table:

Id			Decidable	Guidelines	Notes
C90	C99	C11			
		5	No	Dir 5.1, Rule 9.7	
	8	9	No	Dir 4.12, Rule 18.6, Rule 18.9, Rule 21.3, Rule 22.13, Rule 22.14, Rule 22.15, Rule 22.20	
	9	10	No	Dir 4.12, Rule 18.6, Rule 21.3, Rule 22.15	
	10	11	No	Rule 22.13	Compliance with Rule 9.1 also avoids a common cause of this undefined behaviour but it is not sufficient to avoid all situations in which an indeterminate value might arise.
		42	Yes	Rule 12.6	
	69	75	No	Rule 18.10	
	70	76	No	Rule 18.10	
		71	No	Rule 17.9	
	112	118	No	Dir 4.11, Rule 21.12	
	185	196	Yes	Rule 21.11	
		*	No	Rule 22.18	Added by C18
		*	No	Rule 21.26	Added by C18
		*	No	Rule 22.16, Rule 22.17, Rule 22.18	Added by C18
		*	No	Rule 22.11	Added by C18
		*	Yes	Rule 22.20	Added by C18
		*	No	Rule 22.12, Rule 22.15, Rule 22.20	Added by C18
		197	No	Rule 21.10	

4.5.2 Appendix H.2 — Critical unspecified behaviour

AMD4.67 : Replace the entire table as follows:

This also addresses table layout corruption found in Amendment 2.

Id			Critical	Guidelines	Notes
C90	C99	C11			
1	1	1	No		
	2	2	No		
		3	No		
2	3	4	No	Rule 21.6	
3	4	5	No	Rule 21.6	
4	5	6	No	Rule 21.6	
5	6	7	No	Rule 21.6	
6			Yes		
	7	8	Yes	Rule 5.1	
	8	9	Yes		
	9	10	Yes		Compliance with Rule 21.16 avoids this unspecified behaviour in respect of *memcmp* only.
	10	11	Yes	Rule 19.2	
	11	12	Yes		
	12	13	Yes		
	13	14	Yes		Compliance with Rule 10.1 avoids generation of negative zeros when operating on expressions that have a signed type before promotion.
	14	15	Yes	Rule 7.4	
7, 8	15	16	Yes	Rule 13.2	
9	16	17	Yes	Rule 13.2	
	17	18	Yes	Rule 13.1	
7	18	19	Yes	Rule 13.2	
10	19	20	No		
	20	21	Yes	Rule 8.10	
	21	22	Yes	Rule 13.6, Rule 18.8	
7	22	23	Yes	Rule 13.1	
11	23	24	No		
*	24	25	Yes		
12	25	26	Yes	Rule 20.10, Rule 20.11	
13	26		No		
		*	Yes		Added by C18 – `#line __LINE__` *`new-line`*
	27	27	Yes	Rule 21.12	
	28	28	Yes	Rule 21.12	
	29	29	No		
	30	30	Yes	Dir 4.11, Dir 4.15	
		31	Yes		

Id			Critical	Guidelines	Notes
C90	C99	C11			
	31	32	Yes	Dir 4.11	
		33	Yes		
		34	No		
14	32	35	No	Rule 21.4	
15	33	36	No	Rule 17.1	
	34	37	Yes	Rule 21.6	
16	35	38	Yes	Rule 21.6	
17	36	39	Yes	Rule 21.6	
18	37	40	Yes	Rule 21.6	
	38	41	No		
19	39	42	No	Rule 18.1, Rule 18.2, Rule 18.3, Rule 21.3	Compliance with either Rule 21.3 or all of Rule 18.1, Rule 18.2 and Rule 18.3 will avoid this unspecified behaviour.
	40	43	Yes	Rule 21.3	
		44	Yes		
		45	Yes		
20	41	46	Yes	Rule 21.9	C11 incorrectly omitted *align_alloc*, which was corrected in C18.
21	42	47	Yes	Rule 21.9	C11 incorrectly omitted *align_alloc*, which was corrected in C18.
22	43	48	Yes	Rule 21.10	
	44	49	Yes	Rule 21.10	
		50	Yes		
		*	Yes		Added by C18 – *thrd_exit* destructor invocation ordering
		*	Yes		Added by C18 – *tss_delete* destructor invocations with multiple threads
	45	51	Yes		
	46	52	Yes	Dir 4.15	
	47	53	Yes	Dir 4.15	
	TC3	54	Yes	Dir 4.11, Dir 4.15	Added to C99 by TC3.
	TC3	55	Yes	Dir 4.11, Dir 4.15	Added to C99 by TC3.
	48	56	Yes	Dir 4.11	
	49	57	Yes	Dir 4.11	
	50	58	Yes	Dir 4.11	

4.6 Appendix J — Glossary

Amendment

Insert the following new definitions, in the appropriate (alphabetical) order:

AMD4.68 : Insert new *uniform resource identifier* definition:

Uniform resource identifier (URI)
A *uniform resource identifier* (URI) is a compact sequence of characters that identifies an abstract or physical resource, as defined by RFC 3986 [44].

5 Supporting documents

5.1 Addendum 3 — Coverage against CERT C

Update MISRA C:2012 Addendum 3 [10] to reflect the changes in this Amendment

5.1.1 Guideline by guideline

AMD4.69 : Replace the appropriate rows as follows:

CERT C Rule	MISRA C:2012 Guidelines			Comments
	Guidelines	Coverage		
DCL39-C		None	None	Recategorized from *Out of Scope*
FIO45-C	D.5.1	Implicit	Weak	
CON30-C	D.4.12, R.22.13, R.22.1	Explicit	Strong	
CON31-C	R.22.15, R.22.16	Explicit	Strong	
CON32-C	D.5.1	Implicit	Weak	
CON33-C	D.5.1, R.9.7	Implicit	Weak	
CON34-C	D.4.12, R.18.6, R.22.13	Explicit	Strong	
CON35-C	D.5.2	Explicit	Weak	
CON36-C		None	None	Recategorized from *Out of Scope*
CON38-C		None	None	Recategorized from *Out of Scope*
CON39-C	R.22.11	Explicit	Strong	
CON40-C	R.13.2	Explicit	Strong	
CON41-C		None	None	Recategorized from *Out of Scope*

Note: CON37-C coverage is already included in Addendum 3

5.1.2 Coverage summary

AMD4.70 : Replace the summary table as follows:

Classification	Strength	Number
Explicit	Strong	46
	Weak	6
Implicit	Strong	1
	Weak	16
Restrictive	Strong	24
	Weak	0
Out of Scope	None	0
None	None	6
Total		99

6 References

The following documents are referenced from within this amendment:

6.1 MISRA C

[1] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition)
ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF),
MIRA Limited, Nuneaton, March 2013

[2] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition, 1st Revision),
ISBN 978-1-906400-21-7 (paperback), ISBN 978-1-906400-22-4 (PDF),
HORIBA MIRA Limited, Nuneaton, February 2019

[3] MISRA C:2012 Technical Corrigendum 1, *Technical clarification of MISRA C:2012*,
ISBN 978-1-906400-17-0 (PDF),
HORIBA MIRA Limited, Nuneaton, June 2017

[4] MISRA C:2012 Technical Corrigendum 2, *Technical clarification of MISRA C:2012*,
ISBN 978-1-911700-00-5 (PDF),
The MISRA Consortium Limited, Norwich, February 2022

[5] MISRA C:2012 Amendment 1, *Additional security guidelines for MISRA C:2012*,
ISBN 978-1-906400-16-3 (PDF),
HORIBA MIRA Limited, Nuneaton, April 2016

[6] MISRA C:2012 Amendment 2, *Updates for ISO/IEC 9899:2011 Core Functionality*,
ISBN 978-1-906400-25-5 (PDF),
HORIBA MIRA Limited, Nuneaton, February 2020

[7] MISRA C:2012 Amendment 3, *Updates for ISO/IEC 9899:2011 Phase 2 — New C11/C18 features*,
ISBN 978-1-911700-02-9 (PDF),
The MISRA Consortium Limited, Norwich, October 2022

[8] MISRA C:2012 Addendum 1, *Rule mappings*,
ISBN 978-1-906400-12-5 (PDF),
MIRA Limited, Nuneaton, March 2013

[9] MISRA C:2012 Addendum 2 (2nd Edition), *Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure"*,
ISBN 978-1-906400-18-7 (PDF),
HORIBA MIRA Limited, Nuneaton, January 2018

[10] MISRA C:2012 Addendum 3, *Coverage of MISRA C:2012 against against CERT C 2016 Edition*,
ISBN 978-1-906400-19-4 (PDF),
HORIBA MIRA Limited, Nuneaton, January 2018

6.2 The C Standard

[11] ISO/IEC 9899:1999, *Programming languages — C*,
International Organization for Standardization, 1999

[12] ISO/IEC 9899:2011, *Programming languages — C*,
International Organization for Standardization, 2011

[13] ISO/IEC 9899:2018, *Programming languages — C*,
International Organization for Standardization, 2018

6.3 Other Standards

[14] ARINC 653, *Avionics Application Software Standard Interface*,
Aeronautical Radio Inc., https://aviation-ia.sae-itc.com/standards/

6.4 Other References

[15] *AUTomotive Open System ARchitecture* (AUTOSAR), https://www.autosar.org

[16] *OSEK/VDX Operating System*,
Version 2.2.3., 2005