The Status of Zephyr with respect to MISRA Compliance

Roberto Bagnara

University of Parma

bugseng

http://bugseng.com

Zephyr Developer Summit 2021
Safety Mini-Conference
June 8th, 2021

Outline I

Disclaimer

I am a member of the *MISRA C Working Group* and of ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*

however

the views expressed in this presentation and the accompanying paper are mine and my coauthors' and should not be taken to represent the views of either working group

Do You Want to Reason in Assembly?

```
f:
.LFB0:
  .cfi_startproc
  pushq   %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq    %rsp, %rbp
  .cfi_def_cfa_register 6
  movl    %edi, -20(%rbp)
  movq    $0, -8(%rbp)
  movl    $0, -12(%rbp)
  jmp     .L2
.L3:
  movl    -12(%rbp), %eax
  andl    -20(%rbp), %eax
  movl    %eax, %eax
  addq    %rax, -8(%rbp)
  addl    $1, -12(%rbp)
.L2:
  movl    -12(%rbp), %eax
  cmpl    -20(%rbp), %eax
  jb      .L3
  movq    -8(%rbp), %rax
  popq    %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

Or in C?

```
#include <stdint.h>

uint64_t f(uint32_t n) {
  uint64_t total = 0;
  for (uint32_t i = 0; i < n; ++i) {
    total += i & n;
  }
  return total;
}
```

Advantages of C

- C compilers exist for almost any processor
- C compiled code is very efficient and without hidden costs
- C allows writing compact code (many built-in operators, limited verbosity, . . .)
- C is defined by an ISO standard
- C, possibly with extensions, allows easy access to the hardware
- C has a long history of usage in critical systems
- C is widely supported by tools

Disadvantages of C

ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*, has always been faithful to the original spirit of the language

I Trust the programmer

II Let the programmer do anything

III Keep it fast, even if not portable

IV

V

Bad for safety and security!

What is “Behavior”

ISO/IEC 9899:1999 TC3 (N2156) 3.4

behavior

external appearance or action

As-if rule

The compiler is allowed to do any transformation that ensures that the “observable behavior” of the program is the one described by the standard

True in C, but also in C++, Rust, Go, OCaml, . . .

What is “Undefined Behavior”

ISO/IEC 9899:1999 TC3 (N2156) 3.4.3

undefined behavior
behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

No requirements means absolutely no requirements: crashing, erratic behavior of any kind, formatting the hard disk!

Normally it means the compiler assumes undefined behavior does not happen

If it does happen, the programmer has violated the contract: warranty void!

Undefined Behavior: Example

- The program attempts to modify a string literal (6.4.5)

```
#include <stdio.h>

int main() {
  char *str = "Hello!!!";
  str[6] = '\0'; /* Be less emphatic. */
  printf("%s\n", str);

  /* How does the program behave? */
}
```

What is “Unspecified Behavior”

ISO/IEC 9899:1999 TC3 (N2156) 3.4.4

unspecified behavior
use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Unspecified Behavior: Example

- The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?:`, and comma operators (6.5)

```
#include <stdio.h>

int hello(void) {
  return printf("Hello ");
}

int world(void) {
  return printf("world!");
}

int main() {
  return hello() + world();
}
/* Might print: ‘Hello world!’ or
                ‘world!Hello ’
```

What is “Implementation-Defined Behavior”

ISO/IEC 9899:1999 TC3 (N2156) 3.4.1

implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made

Why?

We described:

- Undefined behavior
- Unspecified behavior
- Implementation-defined behavior
- (and we glossed over locale-specific behavior)

Why is the standardized language not fully defined?

- Because implementing compilers is easier
- Because compilers can generate faster code

UB: Modifying String Literals

The behavior is undefined when:

The program attempts to modify a string literal (6.4.5)

Example: in a program there are literals `"Tail"` and `"HeadTail"`

The compiled program can store in memory only `"HeadTail"` and return the pointer to the fifth character as `"Tail"`

Changing one string may also change the other, but the compiler can assume this will never happen

UB: Shifting Too Much

The behavior is undefined when:

> *An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7)*

```
uint32_t i = 1;
i = i << 32;  /* Undefined behavior. */
```

Strange: if I push 32 or more zeros from the right the result should be zero, right?

UB: Shifting Too Much Example

From Intel64 and IA-32 Architectures Manual, page 1706 section "IA-32 Architecture Compatibility":

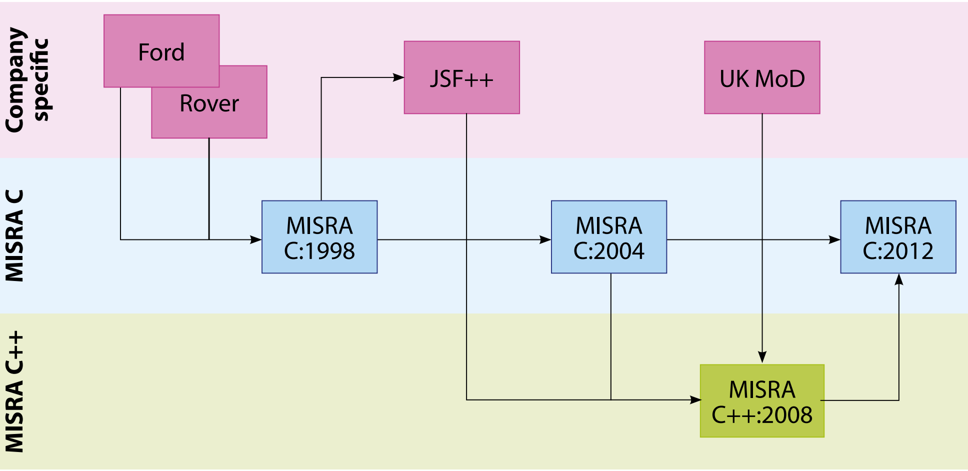
> *The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.*

Basically, this means that in those machines

```
i = i << 32;          /* This is equivalent to... */
i = i << (32 & 0x1F); /* ... this, i.e., ...      */
i = i << 0;           /* this, which is a no-op.  */
```

C leaves the behavior undefined for speed and ease of implementation

History of MISRA C/C++ Guidelines



Strength and Weakness of C

The weakness of the C language comes from its strength:

- Ease of writing efficient compilers for almost any architecture $\Longrightarrow$ non-definite behavior
- Efficient code with no hidden costs $\Longrightarrow$ no run-time error checking
- Many compilers, defined by an ISO standard (must standardize existing practice, many vendors, backward compatibility) $\Longrightarrow$ non-definite behavior
- Easy access to the hardware $\Longrightarrow$ easy to shoot your own foot
- Compact code $\Longrightarrow$ the language can be easily misunderstood and misused

Language Subsetting

Several features of C do conflict with both safety and security

For safety-related applications, language subsetting is crucial

Mandated or recommended by all safety- and security-related industrial standards:

- ISO 26262
- RTCA DO-178C
- CENELEC EN 50128
- IEC 61508

The most authoritative language subset for the C programming language is MISRA C

MISRA C:2012 Guidelines and their Classification

MISRA C:2012 Revision 1 with Amendment 2 contains a total of 175 guidelines

Every guideline is classified as either being a rule (157) or a directive (18)

- **Rule:** a guideline such that information concerning compliance is fully contained in the source code and in the implementation-defined aspects of the toolchain
- **Directive:** a guideline such that information concerning compliance is not fully contained in the source code and i.-d.b.'s: requirements, specifications, designs need to be taken into account

MISRA C:2012 Guideline Category

Every MISRA C:2012 guideline is given a single category: mandatory, required or advisory

Mandatory: C code that complies to MISRA C must comply with every mandatory guideline: deviation is not permitted

Required: C code that complies to MISRA C shall comply with every required guideline: a formal deviation is required where this is not the case

Advisory: these are recommendations that should be followed as far as is reasonably practical: formal deviation is not required, but non-compliances should be documented

MISRA C: Error Prevention, Not Bug Finding

MISRA C cannot be separated from the process of documented software development it is part of

The use of MISRA C in its proper context is part of an error prevention strategy which has little in common with bug finding

The deviation process is an essential part of MISRA C

The point of a guideline is not “You should not do that”

The point is: “This is dangerous: you may only do that if

1. it is needed
2. it is safe
3. a peer can easily and quickly be convinced of both 1 and 2”

MISRA C: Error Prevention, Not Bug Finding (cont'd)

One useful way to think about MISRA C and the processes around it is to consider them as an effective way of conducting a guided peer review to rule out most C language traps and pitfalls

Another aspect that places MISRA C in a different camp from bug finding has to do with the importance MISRA C assigns to reviews:

- code reviews
- reviews of the code against design documents
- reviews of the design documents against requirements

Many MISRA rules have to do with code readability and understandability

"Achieving compliance with MISRA Coding Guidelines"

MISRA Compliance:2020 defines what must be covered when making a claim of MISRA compliance

It replaces its predecessor, MISRA Compliance:2016, by including what must be covered in the software development process

It is applicable to MISRA C:2012 (all revisions) and to MISRA C++:2008: MISRA Guidelines in this presentation

Its adoption is now optional, but will become mandatory for the next editions of MISRA C and MISRA C++

There are concrete advantages in adopting it now

Justifying a Deviation

Deviations must not be permitted:

- simply to satisfy the convenience of the developer
- when a reasonable alternative coding strategy would make the need for a violation unnecessary
- without considering the wider consequences of a particular violation on other guidelines
- without the support of a suitable process
- without the consent of a designated technical authority

Justifying a Deviation (cont'd)

A proposed deviation should only be approved if it can be justified on the basis of one or more of:

1. code quality
2. access to hardware
3. adopted code integration
4. non-compliant adopted code

A Very Preliminary Analysis

We analyzed the build of the `hello_world` sample project for the UP Squared board with the Zephyr SDK 0.12.4 toolkit

The subset of the MISRA C:2012 guidelines considered is the one indicated in
`https://docs.zephyrproject.org/.../coding_guidelines/`
(145 out of 175)

For that build we identified around 32k violations for around half of the selected guidelines: around 20k for required and 12k for advisory guidelines

The Usual Suspects: Implicit Conversions

Rule 10.1	Operands shall not be of an inappropriate essential type: 3740
Rule 10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category: 7730
Rule 10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category: 2454
Rule 10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type: 175

Analysis of Disapplied Rules

Of the 30 rules that are (de facto) disapplied in
`https://docs.zephyrproject.org/.../coding_guidelines/`

17 are advisory

11 are required (Dir 4.3 and Rules 1.1, 1.4, 11.1, 11.3, 18.7, 20.6 (!), 21.5, 21.8, 21.10, and 21.21)

2 are mandatory (Rules 21.19 and 22.2)

Required and mandatory rules cannot be disapplied

Summary

The advantages of C come with corresponding disadvantages that severely impact safety and security: language subsetting is crucial!

MISRA C is the most authoritative subset of C for the development of high-integrity embedded systems

MISRA C is integral part of a software development process, and its adoption is radically different from bug finding

Zephyr is definitely not in bad shape regarding MISRA compliance, but a coordinated effort is required in order do address all the requirements

The End

Questions?

roberto.bagnara@bugseng.com
roberto.bagnara@unipr.it