Software Safety Standards and Guidelines

(Herrmann, §3.2)
MISRA Development Guidelines

- MISRA: Motor Industry Software Reliability Association
- Guidelines for activities across the entire software process:
  - Project planning
  - Integrity
  - Requirements specification
  - Design
  - Programming
  - Testing
  - Product support

- Built on top of **ISO 9001** (international standard for quality management), with extra rules for ensuring safety – beyond ordinary “quality”
Software vs. other automotive components

- Software is primarily a design, with no manufacturing variation, wear, corrosion or ageing aspects.
- It has a much greater capacity to contain complexity.
- It is perceived to be easy to change.
- Software errors are systematic, not random.
- It is intangible.
Automotive applications vs. applications in other industrial sectors

- High production volumes – Components subject to wear
  – Difficult to ensure regular maintenance

Therefore automotive software has an emphasis on

- *data driven* algorithms
  - The code processes data in continuous streams
  - Code should be flexible, able to handle a variety of data
  - Avoid specialized code for each data type

- *parameter optimization*
  - Find the “right” combination of inputs to maximize/minimize output

- *adaptive control*
  - Control laws are not fixed; rather, they adapt to changes over time (e.g. fuel levels) and sudden events (e.g. change in road surface)

- *on-board diagnostics*
Automotive applications vs. applications in other industrial sectors

- Passenger car drivers receive *little or no training* compared with other users of computer-based products and services.
- Therefore, automotive software requires an emphasis on failure management techniques based on the *controllability* of the vehicle.
Automotive applications vs. applications in other industrial sectors

- Traditional automotive test environments use *real* vehicles and components, as well as simulations.
- These are available to *test systems and software extensively and safely* before they reach the customer.
Integrity

- Software Integrity Level (SIL) assigned to each software (sub)component
  - Based on controllability: the ability of the vehicle occupants (not necessarily the driver) to control the safety of the situation following a failure
    - e.g. failure of power window and door lock controllers may prevent passenger from exiting car after an accident

<table>
<thead>
<tr>
<th>Controllability</th>
<th>Acceptable failure rate</th>
<th>SIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncontrollable</td>
<td>Extremely improbable</td>
<td>4</td>
</tr>
<tr>
<td>Difficult to control</td>
<td>Very remote</td>
<td>3</td>
</tr>
<tr>
<td>Debilitating</td>
<td>Remote</td>
<td>2</td>
</tr>
<tr>
<td>Distracting</td>
<td>Unlikely</td>
<td>1</td>
</tr>
<tr>
<td>Nuisance only</td>
<td>Reasonably possible</td>
<td>0</td>
</tr>
</tbody>
</table>
SIL drives choices throughout process

- **Languages & compilers**
  - SIL 0: (default to ISO 9001 standard)
  - SIL 1: “standardized structured language”
  - SIL 2/3: “restricted subset of a standardized structured language; Validated or tested compilers (if available)”
  - SIL 4: “independently certified compilers with proven formal syntax & semantics (when available)”
SIL drives choices throughout process

- Testing
  - SIL 0: (default to ISO 9001 standard)
  - SIL 1: “Show fitness for purpose; Test all safety requirements; Repeatable test plan”
  - SIL 2: “Black box testing”
  - SIL 3: “White box module testing – defined coverage; Stress testing against livelock & deadlock; Syntactic static analysis”
  - SIL 4: “100% white box module testing; 100% requirements testing; 100% integration testing; Semantic static analysis”
MISRA guidelines for C programming

- C’s syntax and type checking is weak, allowing many obscure, error-prone programs to compile. Combine this with almost nonexistent runtime error handling, and you have serious trouble!

- The MISRA guidelines place additional rules for well-formed C programs. Compliance with most of these constraints can be checked automatically and statically (at compile time).

- Two levels of importance:
  - required (R): Special “deviation” needed to override
  - advisory (A): Should not be ignored, but no deviation necessary
1. Environment

1.2 (R) No reliance shall be placed on undefined or unspecified behavior.

- unspecified: must compile correctly, but compiler writer has some freedom – e.g. order of evaluation
- undefined: programming errors for which compiler not obliged to provide error messages – e.g. overflow when adding int values
- Relying on such behavior limits portability
2. Language extensions

2.4 (A) Sections of code should not be “commented out”.

- Use `#if` or `#ifdef` instead:

```
#ifndef FLAG
/* These lines will be "commented out" if FLAG is undefined */
#endif
```

- Still risky – leaving `#ifdefs` in code after they’ve outlived their usefulness
3. Documentation

3.1 (R) All usage of implementation-defined behavior shall be documented.

- implementation-defined: like “unspecified”, but compiler writer is obliged to document implementation choices
- e.g. behavior of integer division & remainder operations when one operand is positive and the other negative
  - e.g. \(-5\) divided by \(3\): Is it \(-1\mod -2\), or \(-2\mod 1\)?
5. Identifiers

- 5.7 (A) No identifier name should be reused.

```c
struct air_speed {
    uint16_t speed;  /* knots */
} *x;

struct gnd_speed {
    uint16_t speed;  /* mph */
    /* Not compliant – speed is in different units */
} *y;

x->speed = y->speed;
```
6. Types

- **6.1 (R)** The plain `char` type shall be used only for the storage and use of characters.
  - Only `=`, `==`, `!=` operators may be used

- **6.2 (R)** `signed` and `unsigned char` types shall be used only for the storage and use of numeric values.
6. Types

- 6.3 (A) typedefs that indicate size and signedness should be used in place of the basic types.

```c
typedef signed char int8_t;
typedef signed short int16_t;
typedef signed int int32_t;
typedef signed long int64_t;
```
7. Constants

- 7.1 (R) Octal constants (other than zero) and octal escape sequences shall not be used.
  - Any integer constant beginning with a 0 is treated as octal. So, for instance, 052 signifies decimal 42.
8. Declarations and definitions

8.5 (R) There shall be no definitions of objects or functions in a header file.

- Header files contain *declarations* of functions – i.e. prototypes (“there will be a function with such-and-such name, return type, parameters”). But they shall not contain the *definitions* of any functions – i.e. the statements that determine what the functions do.

- Clear distinction: .c files contain executable source code; .h files only contain declarations.
8. Declarations and definitions

- 8.8 (R) An external object or function shall be declared in one and only one file.
  - Normally this will mean declaring an external identifier in a header file, that will be included in any file where the identifier is defined or used.

```
featureX.h: extern int16_t a;

useFeatureX.c: #include <featureX.h>
              int16_t a = 0;
```
8.12 (R) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

extern int array1[10];
extern int array2[]; /* Not compliant */
extern int array3[] = {0, 10, 15};
9. Initialization

- 9.1 (R) All automatic variables shall have been assigned a value before being used.
  - Intent: all variables shall have been written to before they are read.
  - This doesn’t imply that all variables must be initialized at declaration. In fact, this is impossible in some cases:

```c
struct LINK {
    int data; struct LINK *next;
} sa = {17}, sb = {42, &sa};
sa.next = &sb;
```
9. Initialization

- 9.2 (R) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

```c
int16_t y[3][2] = {1, 2, 3, 4, 5, 6}; /* not compliant */

int16_t y[3][2] = {{1, 2}, {3, 4}, {5, 6}}; /* compliant */
```
10. Arithmetic type conversions

- Integral promotion
  - Arithmetic operations are always conducted on integer values of type int or long.
  - Operands of any other integer type (char, short, bitfield, enum) are always converted to type int before an arithmetic operation.
10. Arithmetic type conversions

- **Dangerous type conversions**
  - Loss of value: Conversion to a type where the magnitude of the value cannot be represented
  - Loss of sign: Conversion from a signed type to an unsigned type resulting in loss of sign
  - Loss of precision: Conversion from a floating point type to an integer type with consequent loss of precision

- **Guaranteed safe conversions**
  - Conversion of an integral value to a wider type of the same signedness
  - Conversion of a floating type to a wider floating type
10. Arithmetic type conversions

- Always possible to multiply two 8-bit values and access a 16-bit result (if the magnitude requires it)
- But – When multiplying two 16-bit values, it is not always possible to access a 32-bit result

```c
uint16_t u16a = 40000;
uint16_t u16b = 30000;
uint32_t u32x = u16a + u16b; /* result? */
```
- If `int` is implemented as 32 bits, result will be 70000.
- But if `int` is implemented as 16 bits, result will be (70000%65536), or 4464.
- Note: The type of `u32x` has no effect here – the addition is performed *first* (with possible loss of value), *then* conversion to a 32-bit value happens before assignment.
10. Arithmetic type conversions

- 10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
  - it is not a conversion to a wider integer type of the same signedness, or
  - the expression is complex, or
  - the expression is not constant and is a function argument, or
  - the expression is not constant and is a return expression.

- A *complex expression* is anything that is not one of the following: a constant expression; an lvalue; or the return value of a function.
10. Arithmetic type conversions

- Consequences of 10.1
  - No implicit conversions:
    - between signed and unsigned types
    - between integer and floating types
    - from wider to narrower types
    - of function arguments
    - of function return expressions
    - of complex expressions
10. Arithmetic type conversions

```c
int32_t foo(void) {
    int16_t s16a;
    int8_t s8a, s8b;

    return s8a + s8b + s16a;

    return s16a + s8a + s8b;
}
```

+ associates left-to-right
  first addition performed in `int8_t`
  second addition performed in `int16_t`
  type conversion needed, so non-compliant

first addition performed in `int16_t`
second addition performed in `int16_t`
So, addition expression is compliant...

... except that it is a return expression!
Implicit conversion to `int32_t` makes it non-compliant
11. Pointer type conversions

- Pointers can be classified as:
  - pointer to object
  - pointer to function
  - pointer to `void`
  - the null pointer

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.
- e.g. Don’t convert a function pointer to a pointer to a different type of function.

11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to `void`.
11. Pointer type conversions

- 11.3 (A) A cast should not be performed between a pointer type and an integral type.
  - Compare with 11.1, 11.2 – note that this rule is *advisory*

- 11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

- 11.5 (R) A cast shall not be performed that removes any *const* or *volatile* qualification from the type addressed by a pointer.
12. Expressions

12.1 (A) Limited dependence should be placed on C’s operator precedence rules in expressions.

- Suggestion: Parentheses may be omitted
  - on right-hand side of assignment expression (unless the right-hand side itself contains an assignment)
  - with unary operators
  - in sequences of operations where the operators are all the same
12. Expressions

- 12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits.
  - Don’t rely on side effects happening in a particular order
13. Control statement expressions

- 13.3 (R) Floating-point expressions shall not be tested for equality or inequality.

Instead, write library functions that implement the comparison operations – taking into account the floating-point granularity and the magnitude of the numbers being compared.
13. Control statement expressions

- **13.5 (R)** The three expressions of a `for` statement shall be concerned only with loop control.
  - First expression: Initializing the loop counter
  - Second expression: testing the loop counter and optionally other control variables
  - Third expression: Increment/decrement of the loop counter

- **13.6 (R)** Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.
14. Control flow

- 14.1 (R) There shall be no unreachable code.
  - This refers to code *that can be identified at compile time* as unreachable.
  - In general, the reachability problem is undecidable!
    But many cases of unreachable code are easy to detect.

```c
switch(event) {
    case A: do_a();
    break;
    do_more();   /* unreachable */
...```

14. Control flow

- 14.7 (R) A function shall have a single point of exit at the end of the function.
14. Control flow

- 14.8 (R) The statement forming the body of a switch, while, do...while or for statement shall be a compound statement.

- 14.9 (R) An if construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

- 14.10 (R) All if...else constructs shall be terminated with an else clause.
14. Control flow

while (new_data_available)
    process_data(); /* Incorrectly not enclosed in braces */
    service_watchdog(); /* Added later –
                     despite indentation, this is not
                     part of the while body */

while (new_data_available) {
    process_data(); /* Problem avoided */
}
15. **switch** statements

- Many potential problems with this construct... Supplement C’s loose syntax with rules for safety
- 15.1 (R) A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.
- 15.2 (R) An unconditional `break` statement shall terminate every non-empty `switch` clause.
- 15.3 (R) The final clause of a `switch` statement shall be the `default` clause.
16. Functions

- 16.2 (R) Functions shall not call themselves, either directly or indirectly.
  - No recursion – danger of exceeding available stack space
17. Pointers and arrays

- 17.4 (R) Array indexing shall be the only allowed form of pointer arithmetic.
  - Clearer, less error prone than pointer manipulation.
18. Structures and unions

- 18.4 (R) Unions shall not be used.
  - Risk of misinterpreting the data within the union
19. Preprocessing directives

19.1 (A)  

#include statements in a file should only be preceded by other preprocessor directives or comments.

In principle, header files may be included anywhere, but including them at the beginning is in keeping with the “declaration/definition” distinction between .h and .c files.
19. Preprocessing directives

- 19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

- A do-while-zero construct is the only way to have complete statements within a macro:

```c
#define READ_TIME_32() \
   do { \
       DISABLE_INTERRUPTS(); \
       time_now = (uint32_t)TIMER_HI << 16; \
       time_now = time_now | (uint32_t)TIMER_LO; \
       ENABLE_INTERRUPTS(); \
   } while (0)
```
19. Preprocessing directives

- A clearly non-compliant example!

```
#define Obfuscation main
#define using ()
#define the {
#define preprocessor int
#define is i;
#define too for
#define easy (i=97;i<123;++i)
#define but putchar
#define here (i);
#define you but
#define have (0x0a)
#define it ;}
Obfuscation using the preprocessor is too easy but here
you have it
```
19. Preprocessing directives

- 19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.
  - This situation can easily arise with even a moderately complex hierarchy of nested header files.
  - One solution:

    ahdr.h:  
    ifndef AHDR_H
    define AHDR_H
    /* These lines will be excluded if the file has already been included */
    endif
20. Standard libraries

- Many standard functions and macros cannot be used:
  - Dynamic heap memory allocation: 20.4 (R)
  - Standard input library `<stdio.h>`: 20.9 (R)
21. Run-time failures

21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:

- static analysis tools/techniques;
- dynamic analysis tools/techniques;
- explicit coding of checks to handle run-time faults.
References

