CT2 Coding Guidelines

Jil Zerndt
FS 2025

Coding Guidelines

- reduce the number of bugs
- robustness
- correctness
- maintainability
- facilitate code reading within a team
- takes less time to understand another team member's code
- improve portability
- reuse of code on other HW platforms

Enforce by

- automated scans (part of static code checking)
- peer reviews

Appearance

- Indentation: 4 spaces (no tabs)
- Line length: 80 characters
- Only one statement per line (readability)
- No trailing whitespace (no spaces at the end of a line)
- Parentheses to aid clarity (do not rely on operator precedence)
- No magic numbers (use constants instead)

Braces

Non-function statement blocks

`if, else, while, for, do, do-while, switch`

```
if (x == y) {
    p = q;
}
```

~~if (x == y)~~
~~p = q;~~

```
do {
    body of do-loop
} while (condition);
```

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
    ...
}
```

Function statement blocks

`function, class, namespace`

```
int32_t function(int32_t x)
{
    body of function
}
```

`struct, enum:`

```
typedef enum {
    RED,
    GREEN
} colors;
```

```
struct entry {
    uint32_t index,
    uint32_t value
};
```

Spaces

- Mostly function-versus-keyword usage Use space after keywords `if, switch, case, for, do, while`
- No space with sizeof, typeof, alignof, or attribute (as they look somewhat like functions)
  `s = sizeof(struct file);`
- Pointer declaration: * adjacent to data name or function name
  ```
  uint8_t *ptr;
  uint32_t parse(uint8_t *ptr, uint8_t
  **retptr);
  uint8_t *match(uint8_t *s);
  ```

Operators

- No space before or after unary operators: ++, −, +, -, !, , `(type), (cast), sizeof, typeof, alignof, _attribute_, defined`
- One space before or after binary/ternary operators: *, /, %, +, -, «, », <, <=, >, >=, ==, !=, ?, :
- No space around struct member operator: . and ->
- No trailing whitespaces!

Functions

- short and sweet (less than 50 lines)
- do just one thing
- no more than 5-10 local variables
- no more than 3 parameters
- function prototypes shall include parameter names with their data types
- no more than 3 levels of indentation (if, for, while, do-while)
- no more than 2 nested loops (for, while, do-while)

Function Parameters Use const to define call-by-reference function parameters that should not be modified - int32_t strlen (const int8_t s[]); strlen() does not modify any character of character array s - void display(mystruct const *param) ;

Function Definition Just one exit point and it shall be at the bottom of the function - keyword return shall appear only once - All 'private' functions shall be defined static - 'private' → Functions that are only used within the module itself. The function is an implementation detail and not accessible from other modules A prototype shall be defined for each 'public' function in the module header file module.h - 'public' → Functions that are called by other modules. The function prototypes are part of the module interface.

Return Values Shall be checked by the caller If the name of a function is an action or an imperative command - Function should return an error-code integer i.e. 0 for success and -Exxx for failure. - If possible error codes shall be based on the Posix Errorcode - If self-defined error codes are being used they shall be properly documented. In the header file for public functions or in the .c file for private functions - For example, äddworkïs a command, and the add_work () function returns 0 for success or -EBUSY for failure.
If the name of a function is a predicate - Function should return a ßucceeded"boolean. - "PCI device presentïs a predicate, and the pci_dev_present () function returns 1 if it succeeds in finding a matching device or 0 if it doesn't. - Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. - Generally they indicate failure by returning some out-of-range result. - Typical examples would be functions that return pointers; they use NULL or the ERR_PTR mechanism to report failure.

General Rules

Naming No macro name (#define) shall contain any lowercase letters - Function and variable names shall not contain uppercase letters - Use descriptive names for functions, global variables and important local variables - Underscores shall be used to separate words in names e.g. count_active_users() Use short names e.g. i for auxiliary local variables like loop counters - Do not encode types in names. Let the compiler do the type checking

Comments - All comments shall be in English - C99 comments // are allowed - Explain WHAT your code does not HOW - Don't repeat what the statement says in a comment. - Assume that the reader is familiar with C - Comments shall never be nested - All assumptions shall be spelled out in comments - or even better in a set of design-by-contract tests or assertions - The interface of a public function shall be commented next to the function prototype in the header file. - The comment shall not be repeated next to the function definition in the .c file

Types - Use fixed width C99 data types from stdint. h - e.g. uint8_t or int32_t rather than unsigned char or int - Type char shall be restricted to declarations and operations on strings - Bit-fields shall not be defined within signed integer types - None of the bit-wise operators shall be used to manipulate signed integer data
Signed integers shall not be combined with unsigned integers in comparisons or expressions - Decimal constants meant to be unsigned should be declared with an ' U ' at the end Casts shall be done explicitly and accompanied by a comment - Use just one data declaration or one data definition per line - Allows a comment for each item.

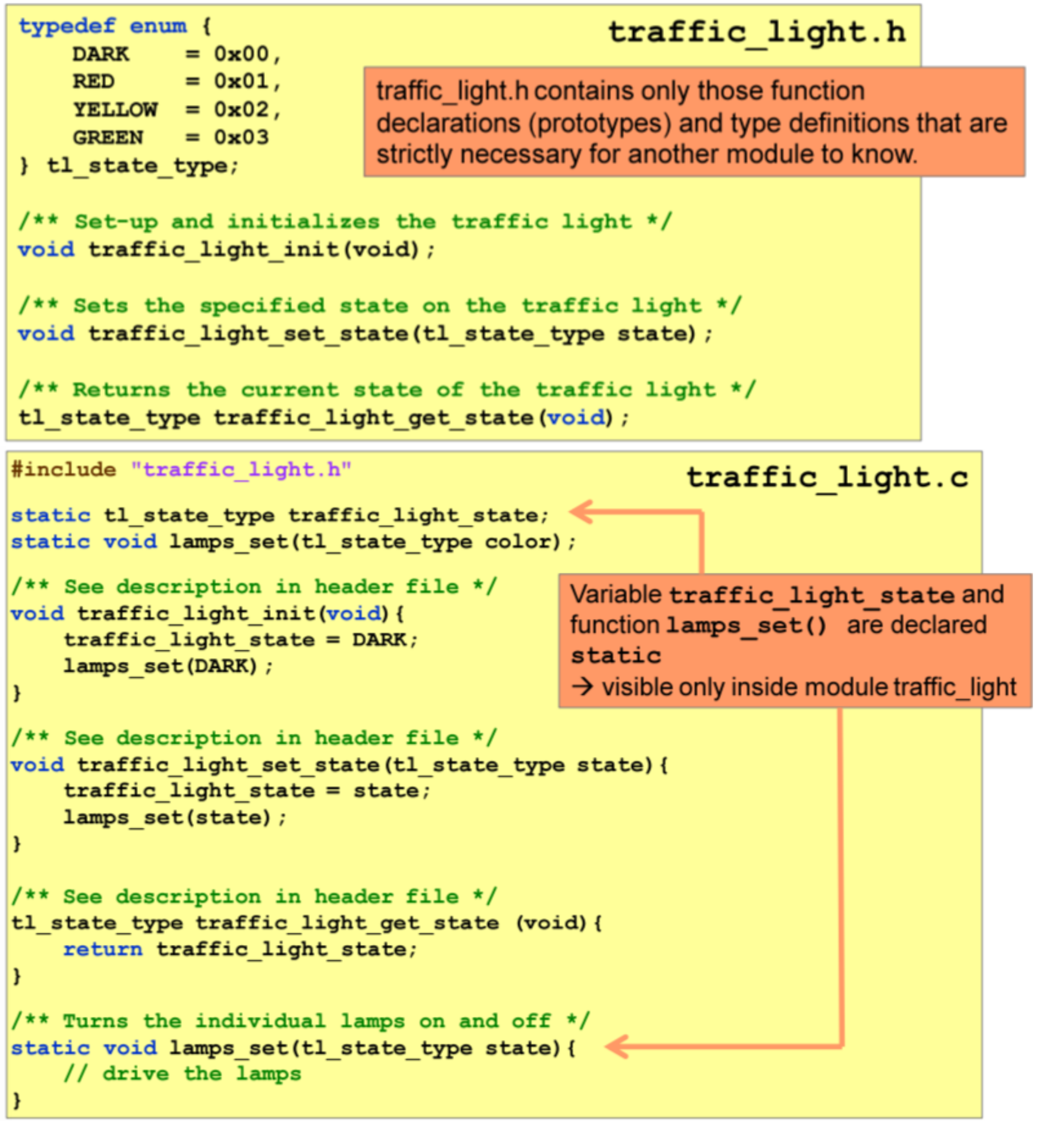
Header Files There shall be precisely one header file for each module Each header file shall contain a preprocessor guard against multiple inclusion

```
#ifndef _ADC_H
#define _ADC_H
#endif /* _ADC_H */
```

Only add #includes that are immediately needed for this header file; do not add #includes for convenience of others Do not define or declare variables - i.e. uint32_t count / extern uint32_t count

Coding Techniques

Example: Module Traffic Light



traffic_light.h

```c
typedef enum {
    DARK    = 0x00,
    RED     = 0x01,
    YELLOW  = 0x02,
    GREEN   = 0x03
} tl_state_type;

/** Set-up and initializes the traffic light */
void traffic_light_init(void);

/** Sets the specified state on the traffic light */
void traffic_light_set_state(tl_state_type state);

/** Returns the current state of the traffic light */
tl_state_type traffic_light_get_state(void);
```

traffic_light.h contains only those function declarations (prototypes) and type definitions that are strictly necessary for another module to know.

traffic_light.c

```c
#include "traffic_light.h"

static tl_state_type traffic_light_state;
static void lamps_set(tl_state_type color);

/** See description in header file */
void traffic_light_init(void){
    traffic_light_state = DARK;
    lamps_set(DARK);
}

/** See description in header file */
void traffic_light_set_state(tl_state_type state){
    traffic_light_state = state;
    lamps_set(state);
}

/** See description in header file */
tl_state_type traffic_light_get_state (void){
    return traffic_light_state;
}

/** Turns the individual lamps on and off */
static void lamps_set(tl_state_type state){
    // drive the lamps
}
```

Variable `traffic_light_state` and function `lamps_set()` are declared `static`
→ visible only inside module traffic_light

Encapsulation:
- Interface: .h file
- Implementation: .c file

maybe finish but not rn