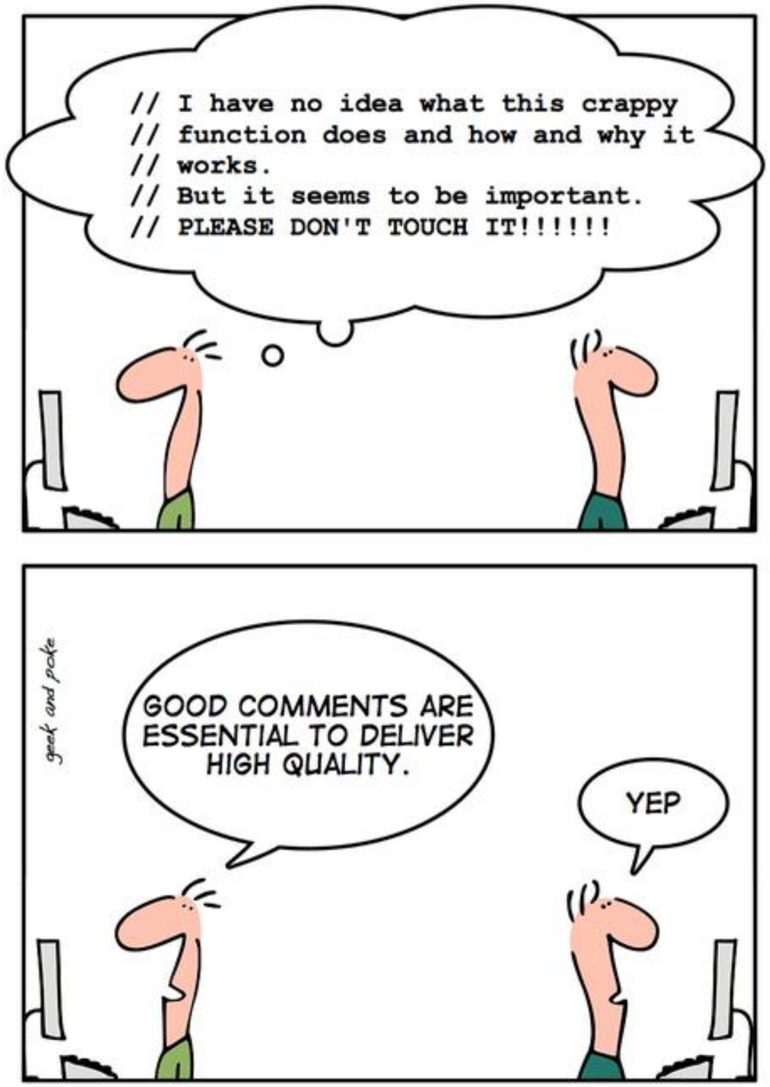
School of Engineering
InES Institute of Embedded Systems

Coding Guidelines

CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal, A. Rüst, J. Scheier, M. Thaler

Motivation


// I have no idea what this crappy
// function does and how and why it
// works.
// But it seems to be important.
// PLEASE DON'T TOUCH IT!!!!!!
GOOD COMMENTS ARE ESSENTIAL TO DELIVER HIGH QUALITY.
YEP
geek and poke

Why 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

Coding Guidelines

- **Rules are subjective**
  - Different organizations have different guidelines
- **"When in Rome do as the Romans do"**

Appearance

- **Indentation**
  - 4 Spaces, no Tabs
- **Maximum of 80 characters per line**
  - Print-outs
  - On-screen code diff
- **No more than one statement per line**
  - Readability and clarity
- **Use parentheses to aid clarity**
  - Do not rely on C's operator precedence rules
    - They may not be obvious to the maintainer

Braces

- **Non-function statement blocks**
  - `if, else, switch, for, while, do`
  - opening last on line
  - closing first on line
  - always use braces also for single statements and empty statements
    - reduces risk during code changes

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

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

Braces

- **Functions**
  - opening beginning of next line
  - closing first on line

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

Braces

- **The closing brace is empty on a line of its own,**
  - except in cases where it is followed by a continuation of the same statement
    - e.g. a "while" in a do-statement or an "else" in an if-statement

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

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

Braces

- **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);
```

Spaces

- **Use one space on each side of binary and ternary operators**

```
=    +    -    <    >    *    /    %    |
&    ^    <=   >=   ==   !=   ?    :
```

- **No space after unary operators**

```
&  *  +  -  ~  !  sizeof  typeof
alignof  __attribute__  defined
```

Spaces

School of Engineering
InES Institute of Embedded Systems

- No space before postfix unary operators

```
i++;
p--;
```

- No space after prefix unary operators

```
++i;
--p;
```

- No space around the '.' and "->" structure member operators
- Do not leave trailing whitespaces

Functions

- **Short and sweet**
  - i.e. no more than about 50 lines of code
- **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**

Functions

- **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);` [1]

1) Same as `void display(const mystruct *param);`

Functions

- **Just one exit point and it shall be at the bottom of the function**
  - keyword **`return`** shall appear only once
- **All <u>'private' functions</u> 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 <u>'public' function</u> 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, "add work" is a command, and the `add_work()` function returns 0 for success or -EBUSY for failure.

Return Values

- **If the name of a function is a predicate**
  - Function should return a "succeeded" boolean.
  - "PCI device present" is 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.

Naming

School of Engineering
InES Institute of Embedded Systems

- 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**
  - i.e. do not use `&`, `|`, `~`, `^`, `<<` and `>>` on signed integers

Types

- **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`

School of Engineering
InES Institute of Embedded Systems

Coding Techniques

CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal, A. Rüst, J. Scheier, M. Thaler

Module Traffic Light

Encapsulation

- Interface → .h
- Implementation → .c

.h contains the module interface

traffic_light.h

```
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.

Module TL

.c contains the implementation

School of Engineering
InES Institute of Embedded Systems

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

Module Traffic Light

- **Caveat**
  - Example module *'traffic_light'* can only be used for a single instance of a traffic light
  - Reason: **`traffic_light_state`** is a **`static`** variable
  - In many embedded use cases having a single instance is fine
  - But what if I have more than one traffic light?

Module Traffic Light

School of Engineering
InES Institute of Embedded Systems

Possible approach

- Include a static variable for each traffic light

```
static tl_state_type tl_state_pedestrian;
static tl_state_type tl_state_cars;
```

- Requires an additional parameter in many of the functions

```
typedef enum {
    PEDESTRIAN,
    CARS
} tl_instance_type;

void traffic_light_set_state(tl_state_type state,
                             tl_instance_type instance){
```

- Alternatively an array of traffic lights could be used

```
static tl_state_type tl_state[5];
```

Module Traffic Light

- **Now more than one traffic light is possible**
  - But each time we add an instance of a light we need to change the module *traffic_light*
- **Possible approach**
  - Extract the traffic light state from module *traffic_light* and let the module using *traffic_light* allocate the memory

module using traffic_light

```
#include traffic_light.h

int32_t main(void) {
    tl_state_type ped_light;

    void traffic_light_init(&ped_light);
    void traffic_light_set_state(&ped_light, RED);
    ...
}
```