C coding standard

Søren Bøgeskov Nørgaard

Last updated: September 10, 2013

This documents a coding style, that may be used for programming in C. If no other coding styles are requested, this can be followed to make code in a project look alike. Using a coding style in a project makes the code easier to read because everything is found in the same place.

This coding style is based on the Kerninghan & Richie (K&R) coding style, and is also inspired by the coding style of the Linux kernel. On the last pages there is a coding example, there most parts of coding is examplified (the code does absolutely nothing, but may make it easier to understand what is meant by the text).

Happy coding!

Contents

1 Visual organization

Visual setup of the code, including indention and parentheses (curly brackets).

1.1 Indention

- Indention is 4 spaces.
- Labels, regarding `goto`'s, are not indented, and are put all the way to the left.
- Labels/`case`'s regarding `switch`es are indented on the same level as "`switch`".

1.2 Curly brackets

- Curly brackets are put on the same line as the expression it belongs to (`if`, `while`, etc) *except* for functions!
- By functions the curly brackes are put on the line below the function expression, on a line of it's own.
- Don't use curly brackets after `if`, `else`, `for`, `while`, if only a single statement follows. The statement is put on a line of it's own, and is indented.
- In `if .. else if .. else` structures, the `else if` and `else` are put on the same line as the previous statement's ending curly bracket.
- After an ending curly bracket in a `do-while` structure, the `while` is put on the same line.

1.3 Whitespace

- Whitespace is used consequently to split parts of the code, that don't belong together – like you would do with paragraphs in a book.
- Generally, no more than 1 line of whitespace is used.

1.4 Line length

- Lines should be kept shorter that 80 characters. This is old pratice, but is generally smart if you, for example, are printing the code in a report.
- If a functions parameters are more than 80 characters, the line can be split after a comma, and aligning the next parameter below the beginning parenthesis of the function (below the first argument).
- If other structures like `if`, `for` and so on become to long, these can be split over several lines, by indenting under the parenthesis, like for functions. If the split is by an operater, this should be put on the next line.
- Long constant strings can become longer than 80 characters, to make it easier to search for them.

1.5 Editor setup

Much of the above can be done automatically by your editor, if it is set up properly. Below is a guide for setting up some popular editors.

Vim

In Vim the above effect is given, by putting the following in the init file: `~/.vimrc`:

```
set encoding=utf-8
set tabstop=4
set shiftwidth=4
set softtabstop=4
set expandtab
set autoindent
set cinoptions=:0,l1,t0,g0,(0
syntax enable
```

Emacs

In Emacs the following is put in the init file: `~/.emacs` or `~/.emacs.d/init.el`:

```
(setq-default tab-width 4)
(setq-default c-basic-offset 4)
(setq-default indent-tabs-mode nil)

(prefer-coding-system       'utf-8)
(set-default-coding-systems 'utf-8)
(set-terminal-coding-system 'utf-8)
(set-keyboard-coding-system 'utf-8)
(setq default-buffer-file-coding-system 'utf-8)
(setq x-select-request-type '(UTF8_STRING
    COMPOUND_TEXT TEXT STRING))
```

2 Naming conventions

- All names are in English.
- Global fuctions (those that are not `static`), should have very descriptive and unique names, like `spi_send_byte()`.
- `static` functions (only seen from the source file that contains them), should have short names, that are descriptive in the file context they are in, like `startclk()`.
- Same goes for variables: Global variables should have long/descriptive names, while local variables should be kept short, and descriptive inside the given function.
- Function and variable names are written only in lower case, with words separated by _.
- Often used (local) variable names, are:
  - `i, j, k` for indexes.
  - `p, q` for pointers.
  - `c` for characters.
  - `s` for strings
  - `x, y, z` for floats/doubles.
  - `l` for longs.
  - `n` for ints.
- By "opposite" functions, opposite names are used. Example: `setsck()` and `getsck()`, `readbyte()` and `writebyte()`, and so on.
- Defines are written in `UPPERCASE`, and so are macros. Example: `#define MAX(A,B) a>b?a:b`.

  If macros act like functions, they are written like functions: `#define udelay(us) _delay_us(us)`.
- Error messages are defines as *negative* numbers, that are descriptive of context, and contains and `E` for "error". Example: `#define SPI_ENACK` for an erro on an SPI module that gets NACK'ed.

3 Comments and code documentation

3.1 Documenting comments

- Before every global function there is a block comment, that starts with `/**` and has * indented 1 space for each line. The block is terminated by and empty line of `*/`
- The block is started by a short description of what the function does.

- For every parameter, there is a line starting with `@param x Description of x.`
- For return values is a line like: `@return 0 = success, SPI_ENACK = No acknowledge.`, there all return values are descriped.
- Documenting comments for fuctions are located at the functions prototype, ie. in the `.h`-files.
- `static` functions (in `.c` files) kan containg a short comment before the function, explaining it's return values. Apart from this, the function should be self explanatory.
- Documentation is in English.

3.2 Regular comments

- *Try* to make your code as understandable as possible without comments, and to *only* use them where it's necessary (this is a good exercise).
- Don't write unnecessary comments ("`i++; /* i is incremented by 1. */`")
- If possible, write comments at the end of the line, to make reading of the code easy. If a large comment must be made, write it as a block comment, like this:

  ```
  /*
   * FIXME:
   * Some work needs to be done, to let
   * the user know an ACK has occured.
   */
  ```

  starting with `/*` and ending in `*/`, each on their own line.
- Comments are in English, and are written as if you were writing to the code. Example: `/* Find the student with the best grades. */`.

4 Files

If, for example, a main program is written, the file `main.c`, with a library consisting of `{spi.h, spi.c}`, files are share like the following.

By the way, files are *always* encoded in UTF-8 format.

4.1 main.c

- Includes `spi.h`.
- Code in here is using functions from `spi.h`.

4.2 spi.h

- Contains prototypes for global functions.
- Contains documentation comments for global functions.
- Contains `typedef`s, `struct`s, and `#define`s belonging to the library. Generally `struct`s are used without `typedef`.

4.3 spi.c

- Includes `spi.h`.
- Contains, in the top, `static` functions that are later used in the global functions.[1]
- Contains, in the bottom, the code for the global functions, that are defined in `spi.h`.

[1] `static` functions are used to split the functions into simple functions, that each do 1 thing, and does it well. Just think of $\sin(x)$.

5 Code example

5.1 spi.h

```
#include <stdio.h>

#ifndef SPI_H
#define SPI_H
#define RED 0x00FF0000
#define BLUE 0x0000FF00
#define GREEN 0x000000FF

#define MAX(A,B) a>b?a:b
#define udelay(us) _delay_us(us)

typedef unsigned char u8;

struct spi_adapter {
    unsigned char mosi;
    unsigned char miso;
    unsigned char sck;
};

/**
 * Sends one byte via SPI.
 * @param byte Data byte to be sent.
 * @param divide Number to divide the clock by.
 * @param slave ID of the slave to send to.
 *
 * @return 0 = success, SPI_ENACK = No acknowledge.
 */
int spi_send_byte(unsigned char byte, unsigned byte divide,
                  unsigned char slave);

#endif
```


5.2 spi.c

```
#include "spi.h"

static void startclk()
{
    SPICTL |= 0x01;
}

int spi_send_byte(unsigned char byte, unsigned byte divide,
                  unsigned char slave)
{
    printf("Så er den sendt...\n");

    if (SPICTL & 0x01)
        return 0;
    else
        return SPI_ENACK;
}
```

5.3 main.c

```
#include "spi.h"

int main(void)
{
    unsigned char input, c;

    input = getchar();
    switch (input) {
    case 0x01:
        spi_send_byte(0x01, 1, 0x01);
        break;
    case 0x02:
        c = spi_receive_byte();
        break;
    default:
        printf("Command unknown: %x\n", input);
        goto bailout;
    }

    /*
     * Underneath are some different
     * ways to use if-else statements
     * for this standard
     */
    if (c > 100)
        printf("Det var et ordentligt tal!\n");
    else
        printf("Ja det fint...");

    if (c == 0x01) {  /* Switch colors. */
        change_color(BLUE);
        printf("Skiftet til blå");
    } else if (c == 0x02) {
        change_color(RED);
        printf("Skiftet til rød");
    } else {
        change_color(green);
        printf("Skiftet til grøn");
    }

    if (c < 10 || input == 0x03 || input == 0x04
        || input == 0x05) {
        print_help();
        delay(1000);
    }

    return 0;

bailout:
    return -1;
}
```