ENEL429 Project Coding Standards: DRAFT

Benjamin Washington-Yule

April 3, 2011

1 Introduction

This document goes over a few high level coding standards to make sure we can all work with each others code. It is *by no means* declaring that this is the way we have do things, but I hope it will serve as a starting point so we can each have our say on what our opinions are on coding style.

Please tell me what your feelings are on these matters. Especially anything you'd like to change/add that you feel is important.

2 Revision Control

An svn repository has been set up already, but I know that the computer engineers of the group would rather use git. I have no experience with git, but I'm not too fussed about which we use, although I do wonder about the benefits we'd gain from using git over svn for such a small project.

It looks to me like we have three options:

1. Ditch svn (if we are allowed) and use git. The onus would be on the computer engineers to teach the rest of us how to use the new system.
2. Stick with svn. I have a feeling Mike has given himself access to our repo for a reason: he wants to keep an eye on what we are doing. Will he be able to do this if we move to git?
3. Employ something like git-svn so that those who want to use git can do so, while the others can use svn. See `http://ecewiki.elec.canterbury.ac.nz/mediawiki/index.php/Git`. Of note is the passage: *"Git-svn allows Git to interface with a subversion repository. This is useful if you already have a project that uses subversion."*

3 Coding Style

With six people, it is more than likely that there will be six individual coding styles. Probably the easiest rule is simply to follow the style of whoever created or owns the module. That said, we can lay down some basic style rules from the outset. Personally, I tend to default to whatever Mike Hayes uses[1] as it is clear and makes sense to me. Also, he is the one marking our code,

The norm for (embedded) C seems to be `underscore_separators` rather than `camelCase`. If anyone does want to use camelCase (and they own the module), it is important to call variable/function names the same as they would be had they been written with underscores. E.g. `pio_config_set(...)` becomes `pioConfigSet(...)` et cetera.

Other things like the position of the opening brace for loops and finicky white-space issues, for example: "`my_way()`" versus "`their_way ()`" are probably best dealt with by following the golden rule of sticking with the existing style, decided on by whoever owns the module.

[1] Except for operating system choice.

3.1 Environments and Layouts

We will all be using very different development environments and we must be able to easily browse each others code. To make life easier on everyone we can adopt a maximum line length of 80 characters, anything longer should be manually broken. Luckily C is not sensitive to whitespace so this is relatively easy using the backslash character.

```
/* A function broken over two lines. */
static void rblack_rating (int annoyance_level, int talent_level \
                           int attractiveness)
{
    printf("Rebecca Black is the new queen of \\b\\\n");
    ...
}
```

3.2 Naming Conventions

It is important that we get these correct so that we know if we are dealing with a structure, function or variable and which module it belongs to.

3.2.1 Functions

Mike Hayes uses the following style: `<module_name>-<verb>`, or `<module_name>-<property>-<verb>`. Examples of these are:

- `pio_config_set(...)`
- `spi_eeprom_write(...)`
- `spi_mode_set(...)`
- `adc_read(...)`

At the very minimum I suggest we prefix functions with the module name.

3.2.2 Structures

There's three naming options here: `NamingExample`, `Naming_Example` and `naming_example_t`. Personally I like the latter but I think the last two are clear enough to be recognised immediately (the first may clash with anyone using camelCase). It may also may life easier if the structure is typedef'd to a pointer to that structure.

```
/* Naming method 1: */
struct example_struct {
        uint8_t something;
        bool something_else;
};

typedef example_struct *example_t;

/* Usage: */

example_t instance = example_init(); // Returns a pointer to a struct.
```

```
/* Naming method 2: */
struct Example_Struct {
        uint8_t something;
        bool something_else;
```

```
};

typedef Example_Struct *Example;

/* Usage: */

Example instance = example_init(); // Returns a pointer to a struct.
```

3.2.3 Enumerations

To avoid clashing with each others enumerations, it may help to prefix the module name to the enumeration, especially if an element of the enum is a common word. Example:

```
enum motor_speed_enum {MOTOR_SLOW = 0, MOTOR_NORMAL, MOTOR_FAST};

typedef enum motor_speed_enum motor_speed_t;
```

3.3 Documentation

The use of Doxygen throughout modules is an excellent idea. At the very least, functions should have a one-line comment about what they do.

```
/* Good: */

/* Set the current motor speed out of 255 */
bool motor_speed_set(motor_t m, uint8_t speed);
```

```
/* Better: */

/** Set the current motor speed.
        @param m The motor object.
        @param speed A number from 0 to 255. 0 = stopped, 255 = full speed.
        @return 1 on success, 0 on failure. */
bool motor_speed_set(motor_t m, uint8_t speed);
```

4 Code Structure

Since we will be working in C, it will probably make life easier if we "pretend" it is an OO language. We can do this by:

- Prefixing all public functions in a module with the module name, e.g., *pwm*_init(), *bluetooth*_wait(),
- Where possible, declare a structure and a set of functions that act on it (i.e, an ADT) for each module.
- Declaring that the first argument of any function in a module to be the "object" we are acting on.
- Write a pseudo-constructor for all modules named *module*_init, which returns a pointer to an instance of that particular structure.

Here's an example function prototype from Mike's code that shows some of these ideas, note the first argument in red, which is a pointer to the "instance" of the structure we are working with.

```
spi_eeprom_size_t
spi_eeprom_write (spi_eeprom_t eeprom, spi_eeprom_addr_t addr, const void *buffer,
                                    spi_eeprom_size_t size);
```

And here's an absolutely contrived example of a motor ADT showing these ideas (based of what I've seen in Mike's code):

```
struct motor_struct {
        pio_t control_line;
        uint8_t speed;
        motor_state_t state;
        ...
};

typedef motor_struct *motor_t;

/** Constructor. Returns a pointer to a filled-out motor stucture. */
motor_t motor_init(pio_t control_line);

/** Set motor speed. */
void motor_speed_set(motor_t motor, uint8_t speed);

/** Get motor state. */
motor_state_t motor_state_get(motor_t motor);
```

And another equally contrived usage situation:

```
int main(void)
{
        motor_t leftmotor = motor_init(LMOTOR_PIO);

        if (motor_state_get(leftmotor) == MOTOR_ON) {
                motor_speed_set(leftmotor, 100);
        }
}
```

5 Final Words

What say you? I'd like to know what you agree with and disagree with. Especially around coding style. It's almost guaranteed I've left something important out too.