Adrien Forest
fjxokt@gmail.com
Christophe Carasco
carasco.christophe@gmail.com
Matthieu Maury
mayeu.tik@gmail.com

UPPSALA
UNIVERSITET
Department of Information Technology
Master Student

Operating System Project 10hp

SSIK : Simply & Stupidly Implemented Kernel
Project Convention

May 19, 2010

University year 2009-2010

Written with LaTeX

Contents

1. Project

The project name, **SSIK**, standing for: "Simply & Stupidly Implemented Kernel", is the reversed acronym of **KISS**. **KISS** is an acronym for the design principle "keep it simple and stupid".

With this project we will try to follow this as a main rule.

1.1 Folder hierarchie

```
root
|- bin/
|- build/
|- doc/
|  |- kernel
|  |- user
|- doc_conf_kernel
|- doc_conf_user
|- include/
|- Makefile
|- README
|- report/
|- scripts/
|- src/
|  |- kernel/
|  |  |- include/
|  |- user/
```

Description of all the folder:

bin/: contain the final binarie

build/: contain the object file

doc/kernel/: contain the kernel documentation

doc/user/: contain the user documentation

doc_conf_kernel: doxygene configuration file for the kernel doc

doc_conf_user: doxygene configuration file for the user doc

include/: contain the API and system library header file

Makefile: the project Makefile. Support many option, see further

README: captain obvious prevent me to explain this

report/: contain project report, such as this document

scripts/: script folder, mainly for simisc

src/kernel/: contain the code of the kernel

src/kernel/include: contains general header for the system (like mips related header)

src/user/: contain the code of all the user software

1.2 Makefile

The make file propose this option:

all: build everything

clean: clean every build

doc: generate user and kernel doc

indent: indent all the source with the indent software

kernel: only compile the kernel part. But don't link anything

kerneldoc: generate kernel doc

user: only compile the user part without the system library. But don't link anything

userdoc: generate user doc, which correspond to the user API

run: build everything and run the system in simisc

2. Coding convention

2.1 Code formating

We follow the GNU coding conventions (`http://www.gnu.org/prep/standards/standards.html#Writing-C`) with this modification:

- the block bracket inside a functions should not be indented (`-bli0` in indent)
- no space between a function name and the parenthesis (`-npcs` in indent)
- comment at the right of the code should be aligned (`-c33` in indent)
- name of variable in declaration should be aligned on a column (`-di16` in indent)
- use space instead of space (`-nut` in indent)

The makefile provide an indent option to easily indent everything.

2.2 Comment formating

2.2.1 General information

- comment starting by `/**` will be indexed, not thos started by `/*`
- use at least a brief description with `@brief`. To add a long comment, add a white line after the brief comment
- this tag can be used to comment any structures, variables, etc.

2.2.2 File comment

```
/**
 * @file file name
 * @brief Brief description
 *
 * Long description
 */

/* end of file filename.x */
```

2.2.3 Function comment

```
/**
 * @Brief description which ends at this dot.
 *
 * Details follow here.
 *
 * @param a param
 * @return the return value
 */
```

3. Good use of git

3.1 General information

- Master is the principal branch
- Master should always compile, and should be exempt of obvious bug or malfunction
- other may not follow the previous rule
- always work in an other branch than master. When you want to merge your work with master ensure that:
  - your work compile
  - your work is well commented
  - your master branch is updated with the other master branch
- never merge the master branch IN a devellopment branch (i.e. never type `git merge master` even if it can save your life !)
- if you want to access the last shiny feature of the master branch in your dev branch, use the rebase function (`git rebase master`)

3.2 Writing good commit messages

Here I will massively quote the really good ProGit book (`http://progit.org/book/ch5-2.html`)

> [..] try to make each commit a logically separate changeset. If you can, try to make your changes digestible — don't code for a whole weekend on five different issues and then submit them all as one massive commit on Monday.
>
> The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior — this is a good guideline to follow. It's also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of "I added tests for" or "Adding tests for," use "Add tests for." Here is a template originally written by Tim Pope at tpope.net:

```
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
```

```
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a
single space, with blank lines in between, but conventions vary here
```

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages — I encourage you to run git log –no-merges there to see what a nicely formatted project-commit history looks like.