Fiasco style guide

Michael Hohmuth

February 11, 2004

Contents

1 Introduction

1.1 Motivation and intended audience

This document is meant as a quick reference to a number of guidelines meant for existing and especially new Fiasco kernel developers.

1.2 How to follow these rules: Metarules

This document does not differentiate between strict rules and "soft" guidelines. Also, when browsing the source code, you will notice a large number of deviations from these rules (because most rules are younger than the current kernel code). In this sense, all rules are "soft."

However, as the master guideline, when writing new code, please ensure that the number of rule violations does not grow. (In the future, we might even enforce this property automatically when someone is checking in code.)

2 Programming language, dialect, and subset

Fiasco has been written in C++. However, it uses C++ in a special dialect supported by the Preprocess tool [Hoh04]. In effect, programmers do not need to write C++ header files, and do not need to declare (member) functions – Preprocess automates these tasks.

We use some features of Preprocess for configuration management—please refer to Section 4.3.

Preprocess lacks support for the following C++ features, which therefore cannot be used in Fiasco source code (but can be used in third-party source code used, i. e., can be included, by Fiasco):

- Name spaces — use static class interfaces or global or static free functions instead (Section 6.3.1)
- Nested classes — use forward-declared "private classes" instead
- `#ifdef` and `#if` on file top level, except for `#if 0` — use Preprocess' configuration features instead (Section 4.3)

Some features of C++ are explicitly disallowed because Fiasco contains no run-time support for them:

- Exceptions
- Run-time type identification and dynamic_cast

These features are always disabled on the compiler command line.

On the other hand, templates are allowed. However, please keep in mind that Fiasco's source code needs to be interpreted not only by the latest version of GCC, but also by at least the two preceding stable versions of GCC, *and* —more significantly— by the VFiasco project's semantics compiler, which is a custom compiler *we* (in a broader sense of we) have to maintain. Therefore, using advanced tricks such as expression templates or partial specialization is strongly discouraged.

3 Source-code organization and directory structure

3.1 Subsystems

Fiasco consists of a number of subsystems; among them:

KERNEL The kernel proper. This is the part that implements the L4 specification.

ABI ABI-specific definitions, mostly type definitions and accessor functions.

JDB The built-in kernel debugger.

BOOT The bootstrapper. This part sets up virtual memory, copies the kernel to its standard virtual-address-space location (`0xf0001000`), and runs the kernel.

Subsystems are defined in the Modules file, which is the main configuration file for Fiasco's build system. Please refer to the build-system documentation [Hoh02] for more information on this build system.

3.2 Directory structure

Subsystems usually reside in their own directory under src (the exception being those subsystems which do not contain any source code, but which exist only for maintenance purposes).

Inside each (source-code) subsystem directory such as kern, the directory layout is as follows:

kern/ Source files shared for all Fiasco-supported architectures

kern/shared/ Source files shared by more than one, but not all architectures.

kern/ia32, kern/ia64, kern/ux, and so on Source files that are specific for only one architecture.

Currently, hardware architectures is the only configuration dimension that motivates moving a source file into a subdirectory. In other words, source files pertaining to a particular configuration option (other than hardware architecture), but not another, are located in one of the mentioned directories.

3.3 Source-file naming

Usually, source files belong to exactly one module, consisting of one main C++ class plus, optionally, public utility functions, small interface classes for exchanging data with the main class, and private auxiliary classes and functions. A module can be comprised of multiple source files. These source files all start with the name of the module's main class, in all lower case (e. g., for class Thread_state, file names start with the string "thread_state"). When multiple source files implement one module, each file name (except for the main file's) add a submodule-specific suffix, separated with a dash (-). For example:

- kern/thread.cpp
- kern/thread-ipc.cpp

Fiasco's build system mandates that all source files (of all subsystems) have different names, even if they reside in different directories. To make file names different, our naming convention is to add architecture-configuration strings to the file names, separated by dashes (-). For example:

- kern/thread.cpp
- kern/shared/thread-ia32-ux.cpp
- kern/ia64/thread-ia64.cpp

Occasionally, it is useful to separate configuration-specific code into a source file of its own (see Section 4.3). In this case, the file name contains the configuration string as a suffix, separated by dashes (-). For example:

- kern/thread-v4.cpp
- kern/thread-v2x0.cpp
- kern/ia32/thread-ia32-smas.cpp

3.4 Header files and the C++ preprocessor

As Preprocess assumes the task of writing header files, programmers should not add new header files. The exception to this rule is that header files are required when defining constants that are needed by both assembly code and C++ code.

When using header files, these files must be protected from multiple inclusions using include-file guards, as in the following example for the file config_gdt.h:

```
#ifndef CONFIG_GDT_H
#define CONFIG_GDT_H
// File contents
#endif
```

4 Configuration management

4.1 The configuration tool

The interactive configuration tool is started using “make config”, and can be left using the command “x”. The tool creates two files, intended for inclusion in C++ code and in makefiles:

globalconfig.h This file defines a preprocessor symbol for each *enabled* configuration option.

globalconfig.out This file defines a Make variable for *each* configuration option. Variables for enables options are set to “y”, those for disabled options are set to “n”.

ADDING NEW CONFIGURATION OPTIONS. Help texts for configuration options in the rules file (rules.cml) are sorted by order in which the options appear in the menu defined at the bottom of the file. New configuration options must add such a help text describing what the config option does. After a new config option has been added, define under which conditions the config option should be suppressed, if any. It is generally a good idea to suppress config options for architectures and configurations where the option is meaningless. Options which can be suppressed for certain configurations usually require consistency rules to ensure they are not left in an undefined state from a previous selection. An example suppression rule is:

```
when UX suppress SERIAL
```

The corresponding consistency rule is:

```
require UX implies (SERIAL == n)
```

This ensures that `SERIAL` is set to “n” when someone had previously chosen `IA32` and `SERIAL` set to “y” and then changes the architecture to `UX`.

Do not forget to update the configuration templates in the directory src/templates after modifying configuration options.

4.2 Class Config

This class defines boot-time and constant configuration variables for Fiasco.

The constant variables can be derived from options the configuration tool has written to globalconfig.h. As a special exception, `#ifdef` is allowed here.

Boot-time configuration variables can be derived from the kernel’s command line (class Cmdline).

4.3 Configuration-specific source code

4.3.1 Single or multiple source files

Source code that is specific to one or a single combination of configuration options can reside in a separate file or in the same file as code for other configuration options. The decision of which works better is in the developer's discretion. Developers should try to combine logically cohesive code fragments in one source file, even if the fragments are mutually exclusive.

4.3.2 Conditional compilation

Using preprocessor constants and `#ifdef` for conditional compilation is only allowed for source code passed to the assembler (assembly files and header files meant to be included in assembly code). In C++ source code, this style is discouraged, and Preprocess does not even support it (for top-level conditional compilation).

Configuration-specific C++ code blocks are labeled using Preprocess' configuration-tag feature with `IMPLEMENTATION` and `INTERFACE` directives.

This feature is available only on file top level. Conditional compilation inside function or class blocks is not allowed. Instead, programmers have the following options:

For class definitions Use Preprocess' `EXTENSION` feature to extend class data with configuration-specific contents.

For function definitions

1. Create a Boolean constant in class Config that depends on the configuration option (`#ifdef` is allowed there; see Section 4.2), and use the normal C++ `if` statement to conditionalize the code. We rely on the C++ compiler's optimizer to remove dead code.
2. Factor out configuration-specific functions with a common interface.

4.4 Compile-time polymorphism

Many configurable aspects of Fiasco are implemented as a set of (member) functions that implement a given interface in a specific way. However, unlike typical C++ programs, Fiasco usually does not define these interfaces as abstract base classes and then derives configuration-specific subclass implementations from them. We have deemed the overhead of virtual-function calls as too high in many cases.

Instead, in Fiasco these interfaces are not declared as virtual functions, but as nonvirtual functions. These functions are then defined in configuration-specific `IMPLEMENTATION` sections, which can be combined in one file or spread across multiple files (see Section 4.3). Preprocess assists in this kind of compile-time polymorphism by allowing configuration-specific inline functions and private member functions.

These interfaces do not need to be split out into separate classes. It is often useful to have a part of a class interface that is implemented in a configuration-specific way.

Usually, Preprocess assumes the task of copying (member-) function declarations into a module's public header file, and this is our preferred usage. However, for interfaces that are implemented in a configuration-specific source-code block, we make an exception: The "public" interface (i. e., the interface used by generic client code, which not necessarily needs consist of only public member functions) can be declared and documented in the `INTERFACE` section of the corresponding module. To prevent Preprocess from adding another declaration, the definition needs to add the `IMPLEMENT` directive. The prerequisite is that the interface is implemented in an `IMPLEMENTATION` section that depends on *more* configuration options than the `INTERFACE` section containing the declaration. For example:

```
INTERFACE:

class Pic
{
public:
  /**
   * @brief Static initalization of the interrupt controller
   */
  static void init();
  // ...
};

IMPLEMENTATION[i8259]:

IMPLEMENT
void
Pic::init()
{
  pic_init(0x20,0x28);
}
```

4.5 Maintainer-mode configuration

Fiasco's build process can be instrumented with a number of checks that ensure some of the rules defined in this document. Fiasco developers should enable this option in the interactive configuration tool; the option is called `MAINTAINER_MODE` ("Do additional checks at build time"). The checks enabled by this option include:

- Checking for mutual or circular dependencies between modules (see Section 5)
- Checking for use of deallocated initialization data after initialization time.

5 Dependency management

Fiasco has been designed to be configurable and robust. A precondition for achieving these properties is that modules and subsystems can be removed from the kernel and tested in isolation, which in turn depends on the absence of mutual or circular dependencies between modules and subsystems.

Therefore, as a rule, these dependencies must be avoided. Please consult [Lak96] for standard methods to resolve circular dependencies.

The current dependency graph can be generated in text or graphics form using "make DEPS" and "make DEPS.ps".

6 Source-code style and indentation

6.1 Naming conventions

In general, Fiasco developers despise the ugly MixedCapsNamingConvention made popular by Java. If you really must use such names, please go hack some Java project, not Fiasco. In Fiasco, words in multi-word identifier names are generally separated with underscores, with the sequencing words starting with a lowercase letter or a digit (the only exception being preprocessor symbols). Examples:

```
Funky_type
thread_id
Thread_ready
_current_sched
_thread_0
```

In particular, the conventions are as follows:

- Type names (class names, typedef names, enum names) all start with a capital letter. Examples: `Thread`, `Jdb`, `Boot_info`
- Function names (both member functions and free functions) start with a lower-case letter. Examples: `fpage_map()`, `ipc_send_regs()`
- Nonstatic member variables start with an underscore (_). Examples: `_mode`, `_ready_time`
- Other variables (including static member variables, global and local variables, function-argument names) start with a lower-case letter. Examples: `preempter`, `cpu_lock`
- Enumeration constants start with a capital letter. Examples: `Thread_ready`, `Page_writable`

  The Fiasco architecture board has declared that having the same naming convention for types and constants is not confusing.

- Preprocessor-symbol identifiers are all-uppercase and start with a letter. As with all other identifiers, multiple words are separated using underscores. Examples: `THREAD_BLOCK_SIZE`, `CONFIG_IA32_486`

  Please note that preprocessor constants are deprecated and allowed only in assembly files and header files meant to be included in assembly code (see Section 6.3.2).

(For file-naming conventions refer to Section 3.3.)

6.2 Comments

All comments must be in English. American / Aussie / Kiwi English are OK, too, but Pidgin English or Denglisch are not.

6.2.1 Doxygen comments

Please document at least all interfaces of all classes that are meant for client-code use. These interfaces usually include all public and protected member functions and all nonstatic free functions, but possibly more if there is a private interface implemented by configuration-specific code.

The interface documentation belongs to the function definition, except if configuration-specific functions (with the `IMPLEMENT` directive) are declared in an `INTERFACE` section. In the that case, the documentation belongs to the declaration.

Please use Doxygen's Javadoc-like style (the style using `@` instead of backslashes) to document your interfaces. Fiasco's documentation is generated using Doxygen's auto-brief feature, so `@brief` directives are unnecessary [vH04].

6.2.2 Comment style

The style of multi-line comments is not prescribed, but please be consistent within one source file. All of the following forms are OK:

```
/** The foo function.
 *  This function frobnifies its arguments.
 */

/**
 *  The foo function.
 *  This function frobnifies its arguments.
 */

/** The foo function.
    This function frobnifies its arguments.
*/
```

6.2.3 Marking incomplete code

Please use the token `XXX` inside a comment to mark broken or incomplete code.

6.3 Module-level rules

6.3.1 Singletons

Singleton objects (classes that will be instantiated only once) should be implemented as static class interfaces instead of a normal instantiable class in order to save kernel-stack space (the this pointer does not need to be passed to these classes). However, developers should use normal classes when it is foreseeable that the class needs to be instantiated multiple times in the future, for instance to support SMP machines.

6.3.2 Constant definitions

Constants should generally only be defined in enumerations. Preprocessor constants are discouraged except for source code passed to the assembler (assembly files and header files meant to be included in assembly code).

In C++ code, preprocessor constants must not be used for conditional compilation using `#ifdef` and friends (see Section 4.3.2).

6.3.3 Bit-field types

When implementing a binary interface that has been specified in terms of bits of machine words (such as the L4 ABI or a device interface), it is a bad idea to implement the interface using bit-field types, unless the interface is architecture-specific. The reason is that the assignment of bit-field members to bit offsets is both compiler-dependent and architecture-dependent. If bit fiddling is required, please define a class that wraps an integral type (such as `unsigned`) and manipulate the bits using bit-and and bit-or operators.

Bit-field structures are OK when the exact order of bits in the type's memory representation does not matter.

6.4 Block-level rules

6.4.1 Assertions

Use assertions generously.

Fiasco supports two kind of runtime assertions: `assert` and `check`. Both cause a kernel panic when their argument evaluates to false.

The first, `assert`, works just like `assert` in standard C. It can be removed from the build by defining the preprocessor symbol `NDEBUG` and therefore must not include code that has side effects.

When side effects are desired, use `check` instead of assert. The contents of this macro are not optimized away with `NDEBUG` – only the error-checking code is.

6.4.2 Common idioms

- Endless loops are programmed like this:

  ```
  for (;;)
    // do stuff
  ```

- Sometimes a private inline function is desired, for example if it needs to be wrapped by a stub that is callable from assembly code. In this case, use `inline NOEXPORT` to avoid having to specify a lengthy `NEEDS[]` directive for the inline function:

  ```
  extern "C"
  void
  asm_callable_stub (Thread* t)
  {
    t->do_stuff();
  }

  PRIVATE inline NOEXPORT
  Thread::do_stuff ()
  {
    // ...
  }
  ```

- Macros are discouraged. Use inline functions instead.

- If you really, absolutely have to define a macro, please make sure it can be used as single-statement blocks after `if`, `else`, and the like, by wrapping it like this:

  ```
  #define foo(x)  do { /* your stuff */ } while (0)
  ```

6.4.3 Rules for spacing, bracing, and indentation

If not specified more precisely in this standard, the GNU Coding Conventions for spacing and bracing apply ([GNU04], Section 5.1).

- The tabulator size is 8. A tabulator can always be replaced with spaces that lead to the next tab stop.

- The maximum line length is 80. If a code line becomes longer than 80 characters, it must be broken into shorter lines. If line length becomes excessive, developers should factor out utility functions.

- The indentation increment is 2 spaces.

- In function definitions, put both the return type and the function signature on a line by themselves. Example:

```
PUBLIC inline
void
Preemption::set_preempter (Receiver *preempter)
{
  // ...
}
```

- Put whitespace before each opening parenthesis, except if it is preceded by another opening parenthesis. Put whitespace after each closing parenthesis, except if it is followed by another closing parenthesis or a comma or semicolon. As a special exception, you do not need to put a space between a function name and the function-call operator.

```
a = f (b(), sin ((x + y) * z));
```

- Do not wrap the argument to the `return` statement into parentheses.

- Opening and closing braces ({, }, including the form `};`) always reside on a line of their own. The braces are indented for nested code blocks, but not for type and function definitions. The braced content is always indented with respect to the braces. Goto labels are back-indented 1 space, class-access specifiers (public, protected, private) are back-indented 2 spaces.

```
class Thread
{
public:
  enum Thread_state
  {
    Thread_invalid, Thread_running
  };

  int _prio;
};


PUBLIC
void
Thread::stuff ()
{
  if (receiver()->ipc_try_lock (this) == 0)
    {
```

```
        do_stuff();
    }
}
```

As an exception, if a function definition fits into a single line, the whole function can be defined in one line. This style can aid readability when defining many small functions in a row.

- Curly braces around single-statement code blocks for `if`, `while`, `do`, and `for` are optional, except if the control expression(s) of these statement spans more than one line, the statement following it must be braced.
- Spacing inside expressions is not prescribed. Please do something sensible to save us from adding more rules to this document.

References

[GNU04] The Free Software Foundation (FSF), Boston, MA. *GNU Coding Standards*, February 2004. Web site `http://www.stack.nl/prep/standards.html`.

[Hoh02] Michael Hohmuth. Fiasco's build system. README file in Fiasco's source tree, June 2002. See `l4/kernel/fiasco/src/README`.

[Hoh04] Michael Hohmuth. *Preprocess - A preprocessor for C and C++ modules*. TU Dresden, January 2004. Web site `http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess/`.

[Lak96] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., Reading, MA, 1996. ISBN 0-201-63362-0.

[vH04] Dimitri van Heesch. *Doxygen Manual*, February 2004. Web site `http://www.stack.nl/~dimitri/doxygen/manual.html`.