Micronics Coding Style Guidelines

# Code layout

## Indentation

Use 4 spaces per indentation level. Do not use tabs (most editors allow for tabs to be replaced by multiple spaces).

## Maximum line length

Limit all lines to a maximum of 79 characters.

## Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

## Function definition style

Function name in column 1, outermost curly braces in column 1, blank line after local variable declarations.

static int function(int \*type, char \*base)

{

int t\_size = 1;

int b\_size = 10;

assert(t\_size >= b\_size); /\* type smaller than base! \*/

...

return 1;

}

## Code structure

Filenames should be concise and descriptive and can contain ‘-‘ or ‘\_’ word separators (‘\_’ is preferred). Each file should contain logically complete functionality. This will normally take the form of a single class definition or small set of functions. The public API should be declared in the header file, as the class declaration or function prototypes. Each file should also contain a summary with appropriate details as shown in the example below.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Filename: filename.h

\* Author: xyz

\* Date: 02/01/14

\* Description: Useful description of the class / functions

contained by this file

\* Copyright: Copyright Micronics Ltd 2014

\* Changelog:

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

One space between keywords like if, for and the following left paren; no spaces inside the paren; braces may be omitted where C permits but when present, they should be formatted as shown:

if (mro != NULL)

{

...

}

else

{

...

}

The return statement should not get redundant parentheses:

return 0; /\* correct \*/

return(0); /\* incorrect \*/

Function and macro call style: foo(a, b, c) -- no space before the open parenthesis, no spaces inside the parenthesis, no spaces before commas, one space after each comma.

Always put spaces around assignment, Boolean and comparison operators. In expressions using a lot of operators, add spaces around the outermost (lowest-priority) operators.

If you can, break after commas in the outermost argument list. Always indent continuation lines appropriately, e.g.:

format(TypeError,

"cannot create '%.100s' instances",

type->tp\_name);

When you break a long expression at a binary operator, the operator goes at the end of the previous line, e.g.:

if (type->tp\_dictoffset != 0 && base->tp\_dictoffset == 0 &&

type->tp\_dictoffset == b\_size &&

(size\_t)t\_size == b\_size + sizeof(Object \*))

return 0; /\* "Forgive" adding a \_\_dict\_\_ only \*/

Put blank lines around functions, structure definitions, and major sections inside functions.

All functions and global variables should be declared static unless they are to be part of a published interface

For external functions and variables, always have a declaration in an appropriate header file in the "Include" directory.

## Whitespace

Avoid extraneous whitespace. Use whitespace after operators and braces. No line should end in whitespace. If you think you need significant trailing whitespace, think again -- somebody's editor might delete it as a matter of routine.

## Comments

Comments go before the code they describe. Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

Comments should ***never*** be nested.

## Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with /\* and a single space and ends with \*/. Each new line within the comment block should start with \*.

Paragraphs inside a block comment are separated by a single line starting with \*.

## Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a // and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

x = x + 1; // Increment x

But sometimes, this is useful:

x = x + 1; // Compensate for border

# Naming Conventions

New modules and packages should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

## Overriding Principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

## Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

* b (single lowercase letter)
* B (single uppercase letter)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (or CapWords, or CamelCase -- so named because of the bumpy look of its letters [[3]](http://www.python.org/dev/peps/pep-0008/#id6)). This is also sometimes known as StudlyCaps.

Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

* mixedCase (differs from CapitalizedWords by initial lowercase character!)
* Capitalized\_Words\_With\_Underscores

There's also the style of using a short unique prefix to group related names together. For example, prefix\_this\_function, prefix\_that\_function and so on. (This is done to emphasize correspondence.)

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

* \_single\_leading\_underscore: weak "internal use" indicator.
* single\_trailing\_underscore\_: used by convention to avoid conflicts with keywords, e.g.
* \_\_double\_leading\_underscore: when naming a class attribute
* \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces.

## Prescriptive: Naming Conventions

## Names to Avoid

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

## Namespace Names

Namespaces should have short, all-lowercase names. Underscores can be used in the namespace name if it improves readability.

## Class Names

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

## Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

## Global Variable Names

(Let's hope that these variables are meant for use inside one module only and are therefore declared static.) The conventions are about the same as those for functions.

## Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

mixedCase is allowed only in contexts where that's already the prevailing style to retain backwards compatibility.

## Function and method arguments

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

## Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

## Structs

Structs should be defined using typedef and should have ‘\_t’ appended to the name using CapWords convention . E.g.

typedef struct

{

…

} MyStruct\_t;

## Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX\_OVERFLOW and TOTAL.

## #define Guard

Header files should be guarded against multiple inclusion using the flowing format \_\_FILENAME\_H\_\_

## Preprocessor

Preprocessor #endif statements should always be closed with a comment e.g.

#ifdef something

int var = 0;

#endif // something

## Designing for inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Private attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

Another category of attributes are those that are part of the "subclass API" ("protected"). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behaviour. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the guidelines:

* Public attributes should have no leading underscores.
* If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

* For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behaviour side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

## Public and internal interfaces

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

An interface is also considered internal if any containing namespace or class is considered internal.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module's API.

# Programming Recommendations

General programming principles always apply e.g. [DRY](http://en.wikipedia.org/wiki/Don%27t_repeat_yourself), [KISS](http://en.wikipedia.org/wiki/KISS_principle), SOLID ([SRP](http://en.wikipedia.org/wiki/Single_responsibility_principle), [OCP](http://en.wikipedia.org/wiki/Open/closed_principle), [LSP](http://en.wikipedia.org/wiki/Liskov_substitution_principle), [ISP](http://en.wikipedia.org/wiki/Interface_segregation_principle), [DIP](http://en.wikipedia.org/wiki/Dependency_inversion_principle)), [SoC](http://en.wikipedia.org/wiki/Separation_of_concerns)

Avoid global variables wherever possible

All global functions and variables should be declared static unless explicitly used as part of the public API. (static used in this context limits scope to the specific compilation unit.

Use const wherever possible for variables, arguments and members. (const member functions do not have side effects i.e. they do not modify the class state)

Never nest comments.

Never comment out code blocks – they run the risk of accidentally being reintroduced.

Prefer the use of types defined by stdint.h i.e. int16\_t instead of short, uint32\_t instead of unsigned int etc.

All rules are made to be broken. Use common sense. If in doubt, ask!

# Example

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Filename: filename.cpp

\* Author: xyz

\* Date: 02/01/14

\* Description: Useful description of the class / functions

\* contained by this file

\* Copyright: Copyright Micronics Ltd 2014

\* Changelog:

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include “stdio.h”

#define MACROS\_USE\_UPPERCASE(x) (x)

static int global\_variable = 0;

class SomeClass

{

public:

void public\_function(const int x, char \*y);

int another\_public\_function();

private:

void \_private\_function() const;

int \_private\_variable;

};

/\* Multi-line comment detailing the purpose of

\* the following public function. Should include

\* details about arguments and return type. May

\* contain formatting specifications for Doxygen

\* processing

\*/

void SomeClass::public\_function(const int x, char \*y)

{

/\* Multi-line descriptive comment about

\* the following code block

\*/

for (int i = 0; i < x; i++)

{

\*y++; // Inline comment.

}

}