**Names**

**Make Names Fit**

Names are the heart of programming.

Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

**Function Names**

* Usually every function performs an action, so the name should make clear what it does: check\_for\_errors() instead of error\_check(), dump\_data\_to\_file() instead of data\_file(). This will also make functions and data objects more distinguishable.

Structs are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

* Suffixes are sometimes useful:
  + *max* - to mean the maximum value something can have.
  + *cnt* - the current count of a running count variable.
  + *key* - key value.

For example: retry\_max to mean the maximum number of retries, retry\_cnt to mean the current retry count.

* Prefixes are sometimes useful:
  + *is* - to ask a question about something. Whenever someone sees *Is* they will know it's a question.
  + *get* - get a value.
  + *set* - set a value.

For example: is\_hit\_retry\_limit.

**Include Units in Names**

If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:

uint32 timeout\_msecs;

uint32 my\_weight\_lbs;

**Structure Names**

* Use underbars ('\_') to separate name components
* When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastage because of compiler alignment issues, then by size, and then by alphabetical order. E.g, don't use ``int a; char \*b; int c; char \*d''; use ``int a; int b; char \*c; char \*d''. Each variable gets its own type and line, although an exception can be made when declaring bitfields (to clarify that it's part of the one bitfield). Note that the use of bitfields in general is discouraged. Major structures should be declared at the top of the file in which they are used, or in separate header files, if they are used in multiple source files. Use of the structures should be by separate declarations and should be "extern" if they are declared in a header file. It may be useful to use a meaningful prefix for each member name. E.g, for ``struct softc'' the prefix could be ``sc\_''.

**Example**

struct foo {

struct foo \*next; /\* List of active foo \*/

struct mumble amumble; /\* Comment for mumble \*/

int bar;

unsigned int baz:1, /\* Bitfield; line up entries if desired \*/

fuz:5,

zap:2;

uint8\_t flag;

};

struct foo \*foohead; /\* Head of global foo list \*/

**Variable Names on the Stack**

* use all lower case letters
* use '\_' as the word separator.

**Justification**

* With this approach the scope of the variable is clear in the code.
* Now all variables look different and are identifiable in the code.

**Example**

int handle\_error (int error\_number) {

int error= OsErr();

Time time\_of\_error;

ErrorProcessor error\_processor;

}

**Pointer Variables**

* place the *\** close to the variable name not pointer type

**Example**

char \*name= NULL;

char \*name, address;

**Global Variables**

* Global variables should be prepended with a 'g\_'.
* Global variables should be avoided whenever possible.

**Justification**

* It's important to know the scope of a variable.

**Example**

Logger g\_log;

Logger\* g\_plog;

**Global Constants**

* Global constants should be all caps with '\_' separators.

**Justification**

It's tradition for global constants to named this way. You must be careful to not conflict with other global *#define*s and enum labels.

**Example**

const int A\_GLOBAL\_CONSTANT= 5;

**#define and Macro Names**

* Put #defines and macros in all upper using '\_' separators. Macros are capitalized, parenthesized, and should avoid side-effects. Spacing before and after the macro name may be any whitespace, though use of TABs should be consistent through a file. If they are an inline expansion of a function, the function is defined all in lowercase, the macro has the same name all in uppercase. If the macro is an expression, wrap the expression in parenthesis. If the macro is more than a single statement, use ``do { ... } while (0)'', so that a trailing semicolon works. Right-justify the backslashes; it makes it easier to read.

**Justification**

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

Some subtle errors can occur when macro names and enum labels use the same name.

**Example**

#define MAX(a,b) blah

#define IS\_ERR(err) blah

#define MACRO(v, w, x, y) \

do { \

v = (x) + (y); \

w = (y) + 2; \

} while (0)

**Enum Names**

**Labels All Upper Case with '\_' Word Separators**

This is the standard rule for enum labels. No comma on the last element.

#### Example

enum PinStateType {

PIN\_OFF,

PIN\_ON

};

### Make a Label for an Error State

It's often useful to be able to say an enum is not in any of its *valid* states. Make a label for an uninitialized or error state. Make it the first label if possible.

#### Example

enum { STATE\_ERR, STATE\_OPEN, STATE\_RUNNING, STATE\_DYING};

# Formatting

## Brace Placement

Of the three major brace placement strategies one is recommended:

if (condition) { while (condition) {

... ...

} }

## When Braces are Needed

All if, while and do statements must either have braces or be on a single line.

### Always Uses Braces Form

All if, while and do statements require braces even if there is only a single statement within the braces. For example:

if (1 == somevalue) {

somevalue = 2;

}

#### Justification

It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. This doesn't affect execution speed. It's easy to do.

### One Line Form

if (1 == somevalue) somevalue = 2;

#### Justification

It provides safety when adding new lines while maintainng a compact readable form.

## Add Comments to Closing Braces

Adding a comment to closing braces can help when you are reading code because you don't have to find the begin brace to know what is going on.

while(1) {

if (valid) {

} /\* if valid \*/

else {

} /\* not valid \*/

} /\* end forever \*/

## Consider Screen Size Limits

Some people like blocks to fit within a common screen size so scrolling is not necessary when reading code.

## Parens *()* with Key Words and Functions Policy

* Do not put parens next to keywords. Put a space between.
* Do put parens next to function names.
* Do not use parens in return statements when it's not necessary.

### Justification

* Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

### Example

if (condition) {

}

while (condition) {

}

strcpy(s, s1);

return 1;

## A Line Should Not Exceed 78 Characters

* Lines should not exceed 78 characters.

## Justification

* Even though with big monitors we stretch windows wide our printers can only print so wide. And we still need to print code.
* The wider the window the fewer windows we can have on a screen. More windows is better than wider windows.
* We even view and print diff output correctly on all terminals and printers.

***If Then Else* Formatting**

**Layout**

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

if (condition) {

} else if (condition) {

} else {

}

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

**Condition Format**

Always put the constant on the left hand side of an equality/inequality comparison. For example:

if ( 6 == errorNum ) ...

One reason is that if you leave out one of the = signs, the compiler will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

***switch* Formatting**

* Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
* The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
* If you need to create variables put all the code in a block.

**Example**

switch (...)

{

case 1:

...

/\* comments \*/

case 2:

{

int v;

...

}

break;

default:

}

**Use of *goto,continue,break* and *?:***

**Goto**

Goto statements should be used sparingly, as in any well-structured code. The goto debates are boring so we won't go into them here. The main place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

for (...) {

while (...) {

...

if (disaster) {

goto error;

}

}

}

...

error:

clean up the mess

When a goto is necessary the accompanying label should be alone on a line and to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose.

**Continue and Break**

Continue and break are really disguised gotos so they are covered here.

Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

* It may bypass the test condition
* It may bypass the increment/decrement expression

Consider the following example where both problems occur:

while (TRUE) {

...

/\* A lot of code \*/

...

if (/\* some condition \*/) {

continue;

}

...

/\* A lot of code \*/

...

if ( i++ > STOP\_VALUE) break;

}

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

**?:**

The trouble is people usually try and stuff too much code in between the *?* and *:*. Here are a couple of clarity rules to follow:

* Put the condition in parens so as to set it off from other code
* If possible, the actions for the test should be simple functions.
* Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

**Example**

(condition) ? funct1() : func2();

or

(condition)

? long statement

: another long statement;

**One Statement Per Line**

There should be only one statement per line unless the statements are very closely related.

The reasons are:

1. The code is easier to read. Use some white space too. Nothing better than to read code that is one line after another with no white space or comments.

**One Variable Per Line**

Related to this is always define one variable per line:

**Not:**

char \*\*a, \*x;

**Do**:

char \*\*a = 0; /\* add doc \*/

char \*x = 0; /\* add doc \*/

The reasons are:

1. Documentation can be added for the variable on the line.
2. It's clear that the variables are initialized.
3. Declarations are clear which reduces the probablity of declaring a pointer when you meant to declare just a char.