The chaos Coding Standards

Per Lundberg

Henrik Hallin

Anders Öhrt

The chaos Coding Standards

by Per Lundberg, Henrik Hallin, and Anders Öhrt

Published $Id: coding-standards.sgml,v 1.6 2000/07/08 09:17:38 plundis Exp $

Table of Contents

List of Examples

Introduction

This is The chaos Coding Standards. It is a guide in programming for the chaos operating system. If you intend to submit your code, please adhere to these standards. This will save both you and us some unnecessary work reformatting and rewriting the code. It is very important to keep the same quality on all code.

The code formatting chapters in this publication is mostly C-related, but some of them can be used for other languages too.

Chapter 1. Language choice

Choosing the right language for a project is very important. If you make a bad choice, it may render the program unusable for some, or all, of its users. For example, if you choose a language where a free interpreter/compiler is not available, few people will be able to use it. So please, choose a language which is usable. Another important aspect is to choose a language as suitable for the project as possible. For example, storm, the chaos kernel, is written in a C and assembly combination. This is because those were the most suitable languages - C is quite low-level, which is good for a kernel, and assembly is almost necessary to use. But those considerations may vary depending on your project type. For example, the cAPS (chaos Advanced Packaging System) system is written in a combination of C and Perl, since we found those languages to be most suitable.

If you do not think you will be able to do the project all by yourself, try using a language that is widely accepted (C, C++ and Perl are three good examples). Avoid using "smaller" languages like Objective C or Haskell if possible.

Chapter 2. License choice

chaos is a free operating system, as defined by the FSF, and as such, all parts of it must be free. However, they do not have to be GPL, even though we encourage people to release their work under this license. This is the list of acceptable licenses for chaos:

- *GNU GPL (General public license)* is our preferred license for all code we write *except* for the system libraries. This is necessary to make licensing of chaos programs under other free licenses possible. Use this license if possible.
- *GNU LGPL (Lesser General Public License)* permits linkage between your work and proprietary (closed-source, non-free) programs. We only use this license for the system libraries. It is deprecated for everything else.
- *BSD* is the Berkeley Software Distribution license. We do not use it, but you may use it *if*, and only if, you remove the advertising clause. We do not recommend using this license anyway, since it is not a copyleft and thus does not protect your freedom in a way we prefer, but we will nevertheless accept if you decide to do so.
- *The Artistic license* is the license used by Perl. We do not encourage its usage except for Perl-related things; in such cases, please follow the Perl example and dual-license your code under both the Artistic license and the GNU GPL.
- The GNU FDL (Free Documentation License) is a pretty new license from the Free Software Foundation. It is intended for documentation, and should be used for that.

You do not need to give over the rights to your code to us. Licensing it under one of those licenses is probably enough, but be careful if you have signed a contract with your employer stating they own everything you write. If so, you can not legally license the code under GPL yourself. You must get permission from your employer to do so. Please be careful with these issues so neither you or we get into legal trouble.

Chapter 3. Compiler flags

The chaos build system, autochaos, automatically uses our standard C flags, so there is little need to duplicate them here. What our flags do is try to eliminate the pitfalls of the C programming language (there are quite a few). The only really important flag is -Werror, which turns all warnings into errors. We believe this should have been the default in gcc. A warning is nothing to take lightly. And really, if you do not care about the warning, why don’t you turn it off?

Chapter 4. Symbol names

All symbol names should be in lower-case only. The only exception to this is acronyms like VGA, GDT, IDT etc, where you could put the acronym in uppercase if you would like. Separate logic words with underscores, like this: screen_base_address.

It is very important to be consequent when naming your functions, variables and such. Do not abbreviate names. It only makes it harder to figure out what a function or variable is used for. Symbol names must *always* be descriptive. One-name variables are neat for the lazy, but is a no-no. Try to name your counter variables index (if you are operating on a one-dimensional list), x and y (if a two-dimensional structure), x, y and z, and so on. In short, try to be as descriptive as possible. It is tempting at times to call your variables i, j, k and such, but this just makes the program more difficult for other people to get into.

All global symbols should have their file name prepended to them. Thus, all the symbols exported from the file `port.c` should be named port_*. When doing a library, it is okay to split it up into smaller files and still have the functions named foo_bar. (if the library name is foo)

Chapter 5. Functions

The usage of functions is standard in all structured programming. You should try to keep your functions as long as suitable. We do not give a specific limit, but try to split your code in natural functions. For example, if your program is reading and processing lines from a file, and the processing is more than a few statements of code, put it in a separate function. This is not really difficult when you get used to it.

Always put a comment before the function declaration where you state what it does. It is also desirable that you document the input and return variables if it is not implicit.

A typical function declaration can look like this:

```
/* Initialise the page allocation system. */

void page_init (void)
{
  [...]
}
```

As you see, we do not put the function name in the leftmost column. There is no reason to do so. It just makes the code look ugly.

Also, if the function types and names do not fit on one line, write it like this:

```
extern inline return_type system_call_thread_control
  (process_id_type process_id, u32 class, u32 parameter)
```

Chapter 6. Comments

Code without comments is like an operating system without applications -- it is not very usable. Try to comment your code as much as suitable to understand it for a reasonably experienced programmer. A good idea is to let someone else look at your code and see where she gets stuck.

Always put comments on their own line, with a blank line above and below. Do not put comments at the end of the line, after the code; it just makes everything messy.

When commenting a statement, put the comment above the statement, like this:

```
/* Check if we are ready yet. */

if (finished ())
{
  return;
}
```

Some people like C++ comments. We prefer to only use them for temporary commenting away code, so that they can not get mistaken for being real comments.

Chapter 7. Indentation

The indentation level should be two spaces. Do not use tabs. When starting a new block, the braces should be on their own line at the same level of indentation as the previous line, like this:

```
if (age > LEGAL_AGE)
{
  print_age (age);
}
```

Chapter 8. Inline assembly

Inline assembly is a twin-edged sword. It can make computing-intensive programs a lot speedier, at the cost of making the source more difficult to get into. Generally, only use it if the speed gain is significant, or in the kernel and other places where things might be impossible to do without it. In general user programs there is little need for it, but occasionally, it might be okay. The thumb rule is to start by optimising the algorithms. If that is not enough, try to profile your code and see where the most time is spent, and rewrite those parts in assembly. If the gain is more than a few percent, inline assembly is okay.

If you for one reason or another have to use inline assembly, do it like this:

```
asm volatile
("\
  pushl %2
  pushl %3
  pushl %4
  lcall %5, $0
"
: "=a" (return_value), "=g" (*buffer)
: "g" (buffer), "g" (parameter), "g" (class),
  "n" (SYSTEM_CALL_KERNELFS_ENTRY_READ << 3));
```

Do not write long blocks of uncommented inline assembly, but keep it into functional blocks separated with blank lines and comments, so it is easier to get into the code.

Chapter 9. Miscellaneous

Here you will find stuff that did not fit into one of the previous chapters:

- When writing if-statements, always use braces to indicate a block, even for one-line statements where it is not required. Like this:

```
if (age > LEGAL_AGE)
{
  print_age (age);
}
else
{
  return;
}
```

- When writing switch statements, always use braces in the case blocks. Like this:

```
switch (key)
{
  case ESCAPE:
  {
    return;
  }
}
```

- When writing conditional statements, never use the fact that non-zero is TRUE. You can only do this on booleans. If something is not a boolean, you must compare it with a number. Like this:

```
if (get_age () != 0)
{
  do_something ();
}
```

  This makes things much clearer. The lack of real booleans in C is showing, but using this fact makes your programs very ugly.

- When declaring external functions (in library header files and elsewhere), always write like this:

```
extern my_function (void);
```

The extern attribute is not mandatory in gcc, but use it anyway since it makes it very clear what we are talking about.

Chapter 10. Whitespace

A lot of programmers tend to forget how important whitespace is to write good code. A good example of this can be found in the Linux source code, but unfortunately, the practice is rather common (especially in the Unix world). In chaos, we use as much whitespace as is practical. Always use whitespace between function names and parentheses, after commas, semicolons and other places. Separate blocks of code with blank lines and comments.

Chapter 11. Documentation

Documentation is essential for a program to succeed. All documentation should be written in English (but may also be translated to other languages if desirable). The standard format for documentation in chaos is SGML using the DocBook DTD. You can find more information about this DTD on the DocBook Web Site (http://www.docbook.org).

Appendix A. Examples

Example A-1. A C example

```
/* $Id: coding-standards.sgml,v 1.6 2000/07/08 09:17:38 plundis Exp $ */
/* Abstract: Semaphores for the kernel. */

/* Copyright 1999-2000 chaos development. */

/* This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
   USA */

#include <storm/i386/error_numbers.h>
#include <storm/i386/semaphore.h>
#include <storm/i386/thread.h>
#include <storm/i386/types.h>
#include <storm/state.h>

/* Semaphore lock variable. 1 if none is accessing the semaphores, 0
   if someone is. */

static kernel_mutex_type semaphore_mutex = 1;

error_type semaphore_wait (kernel_semaphore_type *semaphore)
{
  spin_lock (semaphore_mutex);

  if (*semaphore <= 0)
  {
    spin_unlock (semaphore_mutex);
    thread_block_semaphore (STATE_SEMAPHORE, semaphore);
```

```
    return ERROR_UNKNOWN;
  }
  else
  {
    (*semaphore)--;
    spin_unlock (semaphore_mutex);
  }

  return ERROR_UNKNOWN;
}

/* Release the lock around the given semaphore. */

error_type semaphore_signal (kernel_semaphore_type *semaphore)
{
  spin_lock (semaphore_mutex);

  if (thread_unblock_semaphore (semaphore) == ERROR_NO_THREAD_UNBLOCKED)
  {
    (*semaphore)++;
  }

  spin_unlock (semaphore_mutex);

  return ERROR_UNKNOWN;
}
```

Example A-2. A Perl example

```
#!/usr/bin/perl -w

# $Id: coding-standards.sgml,v 1.6 2000/07/08 09:17:38 plundis Exp $

# Abstract: Program for building chaos packages.
# Author: Per Lundberg <plundis@chaosdev.org>

use chaos::cAPS;
use strict;

my $CONFIGFILE;

# Abstract: Clean things up.
```

```
sub cleanup
{
  close (CONFIGFILE);
}

my $cAPS = new chaos::cAPS (’/mnt/chaos/system/caps’);
my %config_keyword = $cAPS->config_keyword_get ();
my %config;

# All mandatory keywords are stored in this list.

my @mandatory_keyword = (’name’, ’type’, ’version’, ’description’ );

open (CONFIGFILE, ’<chaos/config’) or die "Couldn’t open chaos/config.\n";

# This is the main config file parser loop. Neat, isn’t it? :) This
# used to be written in flex/bison, and was rather complex and
# difficult to get into. But now...

while (<CONFIGFILE>)
{
  chop;
  ($_, my $dummy) = split (’#’);
  (my $key, my $value) = /^([\w\-]+)\s*=\s*\"([^\"]*)\"/;

  if (!$key || !$value)
  {
    print ("Syntax error in config file: $_.\n");
    exit;
  }

  my $flag = 1;

  foreach my $config_key (keys %config_keyword)
  {
    if ($config_keyword{$config_key} eq $key)
    {
      $config{$key} = $value;
      $flag = 0;
    }
  }
  if ($flag == 1)
  {
    print ("Bad keyword ’$key’ in config file. Aborting.\n");
    exit 1;
```

```
  }
}

# It’s time to start some serious business. But first, we check that
# all mandatory keywords have been specified.

foreach my $keyword (@mandatory_keyword)
{
  if (!$config{$keyword})
  {
    print ("Mandatory keyword $keyword missing from config file.\n");
    exit 1;
  }
}

# All things clear? Great! Now we check if the package directory
# exists.

# ..chaos/$config{name}

# Clean things up.

cleanup ();
```

Appendix B. Emacs settings for chaos indentation

This is our .emacs settings for chaos indentation style for C code. It makes writing compliant code much easier.

```
;; chaos indentation style

(defconst chaos-c-style
 '(
   (c-basic-offset . 2)
   (c-comment-only-line-offset . (0 . 0))
   (c-comment-continuation-stars . "")
   (c-hanging-comment-ender-p . t)
   (c-offsets-alist . (
     (statement-block-intro . +)
     (knr-argdecl-intro . +)
     (substatement-open . 0)
     (label . 0)
     (statement-cont . +)
     (case-label . 2)
   )
  )
 )
 "chaos"
)

(defun chaos-c-mode-common-hook ()

;; add my personal style and set it for the current buffer

(c-add-style "chaos" chaos-c-style t)

;; this will make sure spaces are used instead of tabs

(setq indent-tabs-mode nil))

(add-hook 'c-mode-common-hook 'chaos-c-mode-common-hook)
```