YOUR KERNEL NEEDS YOU

Lord Kernel **Greg Kroah-Hartman** wants you for his Linux army. Now here's your basic training...

You don't need a PhD in computer science and years of experience to hack the kernel. Sure, they help, but the nature of Linux development means that it's open to all by default. All you have to do is get stuck in. You use the Linux kernel in whatever shape or form every day; wouldn't you feel just the tiniest swell of pride if you'd helped work on it, no matter in how small a way?

But what if everything works just fine for you, and there's nothing that you'd like to fix? Well, don't despair, the Linux kernel developers need all the help they can get, and have plenty of code in the tree that's just waiting to get cleaned up. One example is the code in the **drivers/staging/** tree, which consists of a lot of drivers that do not meet the normal Linux kernel coding guidelines. The code is in that location so that other developers can help on cleaning it up before it and gets merged into the main portion of the Linux kernel tree.

Every driver in the **drivers/staging** directory contains a **TODO** listing the things that need to be done on it in order for the code to be moved to the proper location in the kernel tree. Most of the drivers all have the following line in their **TODO** file:

```
- fix checkpatch.pl issues
```

Let's look into what this means and what you can do.

Every large body of code needs to have a set of coding style rules in order for it to be a viable project that a large number of developers can work on. The goal of any Linux kernel developer is to have other developers help find problems in their code, and by keeping all of the code in the same format it makes it much easier for anyone else to pick it up, modify it, or notice bugs in it. As every line of kernel code is reviewed by at least two developers before it is accepted, it's important to have a common style guideline.

The Linux kernel coding style can be found in the file **Documentation/CodingStyle** in the kernel source tree. The important thing to remember when reading it is not that this style is somehow better than any other style, just that it is consistent. In order to help developers easily find coding style issues, the script **scripts/checkpatch.pl** in the kernel source tree has been developed. This script can point out problems easily, and should always be run by a developer on their changes, instead of having a reviewer waste their time by pointing out problems later on.

The drivers in the **drivers/staging/** directory all usually have coding style issues, as they were developed by people not familiar with the Linux kernel guidelines. One of the first things that need to be done to the code is to fix it up to follow the correct rules. And this is where you come in: by running the **checkpatch.pl** tool, you can find a large number of problems that need to be fixed.

Specific rules

Let's have a look at some of the common rules that are part of the kernel guidelines.

Whitespace

The first rule that everyone needs to follow is to use the Tab character rather than spaces, to indent code. Also, the Tab character should represent eight spaces. Following along with the eight-character Tab indentation, the code should not flow past the 80 character line limit on the right of the screen.

Numerous developers have complained about the 80 character limit recently, and there are some places where it is acceptable to go beyond that limit. If you find that you're being forced to do strange line-wrapping formatting just to fit

From tiny acorns...

One of the best tutorials for running Git comes within *Git* itself. You can read it and can be read by running:

```
$ man gittutorial
```

after you have installed Git on your box.

So run off and install *Git* on your Linux system using the package manager you are comfortable with, then start by cloning the main Linux kernel repository:

```
$ mkdir ~/linux
$ cd ~/linux
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

This will create the directory **linux-2.6** within the **linux/** directory.

Everything we do from here out will be within that directory, so go into it to start with:

```
$ cd ~/linux/linux-2.6
```

Now that you have the raw source code, how do you build it and install it on your system? That is a much larger task, one that is beyond this article. Luckily a whole book has been written on this topic: *Linux Kernel in a Nutshell*, and can be found free online at: **www.kroah.com/lkn/**.

into the 80-character limit with all of your code on the right-hand side of the screen, it is better to refactor the logic to prevent this from happening in the first place.

Forcing an 80-character limit also forces developers to break their logic up into smaller, easier to understand chunks, which makes it easier to review and follow as well. So yes, there is a method to the madness of the 80 character limit.

checkpatch.pl

With these simple whitespace and brace rules now understood, let's run the **checkpatch.pl** script on some code and see what it tells us:

```
$ ./scripts/checkpatch.pl --help
Usage: checkpatch.pl [OPTION]... [FILE]...
Version: 0.30
Options:
-q, --quietquiet
--no-tree             run without a kernel tree
--no-signoff          do not check for 'Signed-off-by' line
--patch               treat FILE as patchfile (default)
--emacs               emacs compile window format
--terse               one line per report
-f, --file            treat FILE as regular source file
--subjective, --strict enable more subjective tests
--root=PATH           PATH to the kernel tree root
--no-summary          suppress the per-file summary
--mailback            only produce a report in case of
warnings/errors
--summary-file        include the filename in summary
--debug KEY=[0|1]     turn on/off debugging of KEY, where
KEY is one of  'values', 'possible', 'type', and 'attr' (default is
all off)
--test-only=WORD      report only warnings/errors containing
WORD literally
-h, --help, --version display this help and exit When FILE is -
read standard input.
```

Two common options that we will be using are the **--terse** and **--file** options, as those enable us to see the problems in a much simpler report, and they work on an entire file, not just a single patch.

So, let's pick a file in the kernel and see what running **checkpatch.pl** tells us about it:

```
$ ./scripts/checkpatch.pl --file --terse drivers/staging/comedi/
drivers/ni_labpc.c drivers/staging/comedi/drivers/ni_
labpc.c:4: WARNING: line over 80 characters
...
drivers/staging/comedi/drivers/ni_labpc.c:486: WARNING:
braces {} are not necessary for single statement blocks
...
drivers/staging/comedi/drivers/ni_labpc.c:489: WARNING:
braces {} are not necessary for single statement blocks
...
drivers/staging/comedi/drivers/ni_labpc.c:587: WARNING:
suspect code indent for conditional statements (8, 0)
...
drivers/staging/comedi/drivers/ni_labpc.c:743: WARNING:
printk() should include KERN_ facility level

drivers/staging/comedi/drivers/ni_labpc.c:750: WARNING:
kfree(NULL) is safe this check is probably not required
...
drivers/staging/comedi/drivers/ni_labpc.c:2028: WARNING:
EXPORT_SYMBOL(foo); should immediately follow its
function/variable
total: 0 errors, 76 warnings, 2028 lines checked
```

About Git

Probably the most important thing to remember when you're working with Git is never to do your work on the same branch that Linus pushes to, called 'master'. You should create your own branch, and use that instead. This ensures that you will be able to update any changes that are committed to Linus's branch upstream without any problems. To create a new branch called 'tutorial' and check it out, do the following:

```
$ git branch tutorial
$ git checkout tutorial
```

That's it. You are now in the 'tutorial' branch of your kernel repository, as can be seen by the following command:

```
$ git branch
  master
* tutorial
```

The * in front of the 'tutorial' name shows that you are on the correct branch. Now, let's go and make some changes to the kernel code!

I've removed a lot of the warnings from the above output, as there was a total of 76 of them and they were all variants of the ones above.

As can be seen, the **checkpatch.pl** tool points out where the code has gone beyond the 80-character limit, and where braces were used that were not needed, as well as a few other things that should be cleaned up in the file.

Now that we know what needs to be done, fire up your favourite editor and let's fix something. How about the brace warning, (see box on page 58) – that should be simple to resolve. Looking at the original code, lines 486–490 look like the following:

```
    if (irq) {
        printk(", irq %u", irq);
    }
    if (dma_chan) {
        printk(", dma %u", dma_chan);
    }
```

A simple removal of those extra braces results in:

```
    if (irq)
        printk(", irq %u", irq);
    if (dma_chan)
        printk(", dma %u", dma_chan);
```

Save the file and run the *checkpatch* tool again to verify that the warning is gone:

```
$ ./scripts/checkpatch.pl --file --terse drivers/staging/comedi/
drivers/ni_labpc.c | grep 486
$
```

And of course you should build the file to verify that you did not break anything:

```
$ make drivers/staging/comedi/drivers/ni_labpc.o
  CHK     include/linux/version.h
  CHK     include/generated/utsrelease.h
  CALL    scripts/checksyscalls.sh
  CC [M]  drivers/staging/comedi/drivers/ni_labpc.o
```

Great, you've now made your first kernel code fix! But how do you take this change and get it to the kernel developers in the format that they can apply?

More git fun

As you edited this file within a *Git* repository, your change to it is caught by Git. This can be seen by running **git status**:

```
$ git status
# On branch tutorial
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
```

»

Braces

The rules regarding brace usage in the kernel are slightly fiddly. Opening braces should be placed on the same line of the statement they are being used for, with one exception, as shown below. Closing braces should be placed back at the original indentation. This can be shown with the following example:

```
if (error != -ENODEV) {
        foo();
        bar();
}
```

If you need to add an **else** statement to an **if** statement you should put it on the same line as the closing brace, as shown here:

```
if (error != -ENODEV) {
        foo();
        bar();
} else {
report_error();
goto exit;
}
```

If braces are not needed for a statement, do not put them in, as they are unnecessary:

```
if (error != -ENODEV)
        foo();
else
goto exit;
```

The one exception for opening braces is for function declarations, those go on a new line, like so:

```
int function(int *baz)
{
        do_something(baz);
        return 0;
}
```

```
# modified:   drivers/staging/comedi/drivers/ni_labpc.c
#
```

no changes added to commit (use **git add** and/or **git commit -a**).

This output shows that we are on the branch called 'tutorial', and that we have one file modified at the moment, the **ni_labpc.c file**. If we ask *Git* to show what we changed, we will see the actual lines:

```
$ git diff
diff --git a/drivers/staging/comedi/drivers/ni_labpc.c b/
drivers/staging/comedi/drivers/ni_labpc.c
index dc3f398..a01e35d 100644
--- a/drivers/staging/comedi/drivers/ni_labpc.c
+++ b/drivers/staging/comedi/drivers/ni_labpc.c
@@ -483,12 +483,10 @@ int labpc_common_attach(struct
comedi_device *dev, unsigned long iobase,
        printk("comedi%d: ni_labpc: %s, io 0x%lx", dev->minor,
thisboard->name,
              iobase);
-       if (irq) {
+        if (irq)
              printk(", irq %u", irq);
-       }
-       if (dma_chan) {
+        if (dma_chan)
              printk(", dma %u", dma_chan);
-       }
        printk("\n");
        if (iobase == 0) {
```

This output is in the format that the *patch* tool can use to apply a change to a body of code. The leading **-** and **+** on some lines show what lines are removed, and what lines are added. Reading these diff outputs soon becomes natural, and is the format in which you need to send your changes to the kernel maintainer to get the change accepted.

Description, description, description

The raw *diff* output shows what code has been changed, but for every kernel patch, more information needs to be provided in order for it to be accepted. This metadata is as important as the code changes, as it is used to show who made the change, why the change was made, and who reviewed the change.

Here's a sample change that was accepted into the Linux kernel tree a while ago:

```
USB: otg: Fix bug on remove path without transceiver
In the case where a gadget driver is removed while no
transceiver was found at probe time, a bug in otg_put_
transceiver() will trigger.
Signed-off-by: Robert Jarzmik <robert.jarzmik@free.fr>
Acked-by: David Brownell <dbrownell@users.sourceforge.
net>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
--- a/drivers/usb/otg/otg.c
+++ b/drivers/usb/otg/otg.c
@@ -43,7 +43,8 @@ EXPORT_SYMBOL(otg_get_
transceiver);
 void otg_put_transceiver(struct otg_transceiver *x)
 {
-       put_device(x->dev);
+        if (x)
+               put_device(x->dev);
 }
```

The first line of the change is a one-line summary of what part of the kernel the change is for, and very briefly, explains what it does:

```
USB: otg: Fix bug on remove path without tranceiver
```

Then comes a more descriptive paragraph that explains why the change is needed:

```
In the case where a gadget driver is removed while no
transceiver was found at probe time, a bug in otg_put_
transceiver() will trigger.
```

After that, come a few lines that show who made and reviewed the patch:

```
Signed-off-by: Robert Jarzmik <robert.jarzmik@free.fr>
Acked-by: David Brownell <dbrownell@users.sourceforge.
net>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
```

The term 'Signed-off-by:' refers to the ability for the developer to properly claim that they are allowed to make this change, and offer it up under the acceptable licence to be able for it to be added to the Linux kernel source tree. This agreement is called the Developer's Certificate of Origin, and can be found in full in the file, **Documentation/SubmittingPatches** in the Linux kernel source tree.

In brief, the Developer's Certificate of Origin consists of the following:

1. I created this change; or
2. Based this on a previous work with a compatible licence; or
3. Provided to me by (1), (2), or (3) and not modified
4. This contribution is public.

It is a very simple to understand agreement, and ensures that everyone involved knows that the change is legally acceptable. Every developer who the patch goes through, adds their 'Signed-off-by:' to it as the patch flows through the developer and maintainer chain before it is accepted into the Linux kernel source tree. This ensures that every line of code in the Linux kernel can be tracked back to the developer who created it and the developers who reviewed it.

Now we know how a patch is structured, we can create ours. First, tell *Git* to check in the change that we made:

```
$ git commit drivers/staging/comedi/drivers/ni_labpc.c
```

Git will fire up your favorite editor and place you in it, with the following information already present:

```
# Please enter the commit message for your changes. Lines
starting with '#' will be ignored, and an empty message
aborts the commit.
```

```
# Explicit paths specified without -i nor -o; assuming --only
paths...
# On branch tutorial
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   drivers/staging/comedi/drivers/ni_labpc.c
```

Create a summary line for the patch:

```
Staging: comedi: fix brace coding style issue in ni_labpc.c
```

And then a more descriptive paragraph:

```
This is a patch to the ni_labpc.c file that fixes up a brace
warning found by the checkpatch.pl tool
```

Then add your Signed-off-by: line:

```
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
```

Then save the file and *Git* will make the commit, printing out the following:

```
[tutorial 60de825] Staging: comedi: fix brace coding style
issue in ni_labpc.c
1 files changed, 2 insertions(+), 4 deletions(-)
```

If you use the command **git show HEAD** to see the most recent change, it will show you the full commit you made:

```
$ git show HEAD
commit 60de825964d99dee56108ce4c985a7cfc984e402
Author: Greg Kroah-Hartman <gregkh@suse.de>
Date:   Sat Jan 9 12:07:40 2010 -0800
    Staging: comedi: fix brace coding style issue in ni_labpc.c
    This is a patch to the ni_labpc.c file that fixes up a brace
warning found by the checkpatch.pl tool
Signed-off-by: My Name <my_name@my_email_domain>
diff --git a/drivers/staging/comedi/drivers/ni_labpc.c b/
drivers/staging/comedi/drivers/ni_labpc.c
index dc3f398..a01e35d 100644
--- a/drivers/staging/comedi/drivers/ni_labpc.c
+++ b/drivers/staging/comedi/drivers/ni_labpc.c
@@ -483,12 +483,10 @@ int labpc_common_attach(struct
comedi_device *dev, unsigned long iobase,
printk("comedi%d: ni_labpc: %s, io 0x%lx", dev->minor,
thisboard->name,
       iobase);
-       if (irq) {
+       if (irq)
printk(", irq %u", irq);
-       }
-       if (dma_chan) {
+       if (dma_chan)
printk(", dma %u", dma_chan);
-       }
printk("\n");
if (iobase == 0) {
```

You have now created your first kernel patch!

Get your change into the kernel tree

Now that you have created the patch, how do you get it into the kernel tree? Linux kernel development primarily still happens through email, with patches and review both happening that way.

First off, let's export our patch in a format that we can use to email it to the maintainer who will be responsible for accepting our patch. To do that, once again, *Git* has a command, **format-patch**, that you can use:

```
$ git format-patch master..tutorial
0001-Staging-comedi-fix-brace-coding-style-issue-in-ni_
la.patch
```

In this command we're creating all patches that exist in the difference from the 'master' branch (which is Linus's branch, remember way back at the beginning?) and our private branch, called 'tutorial'.

This consists of only one change, our patch. It is now in the file **0001-Staging-comedi-fix-brace-coding-style-issue-in-ni_la.patch** in our directory in a format that we can send off.

Before we attempt to send the patch off, we should verify that our patch is in the correct format, and does not add any errors to the kernel tree as far as coding style issues go. To do that, we use the **checkpatch.pl** script again:

```
$ ./scripts/checkpatch.pl 0001-Staging-comedi-fix-brace-
coding-style-issue-in-ni_la.patch
total: 0 errors, 0 warnings, 14 lines checked
0001-Staging-comedi-fix-brace-coding-style-issue-in-ni_
la.patch has no obvious style problems and is ready for
submission.
```

All's well...

... but who do we send it to? Once again, the kernel developers have made this very simple with a script that will tell you who needs to be notified. This script is called, **get_maintainer.pl**, and is also in the **scripts/** subdirectory in the kernel source tree. This script looks at the files you have modified in the patch and matches it up with the information in the **MAINTAINERS** file in the kernel source tree that describes who is responsible for what portion of the kernel, as well as looking at the past history of the files being modified. From this it magically generates a list of people who need to be notified about the patch, complete with email addresses.

So, we should just bring up our favourite email client and send the patch off to all addresses that **get_maintainer.pl** told us about, right? Not so fast! Almost all common email clients do nasty things with patch files, wrapping lines when they shouldn't be wrapped, changing tabs into spaces, eating spaces when they shouldn't and all sorts of other nasty things.

For details about all of these common problems, and how to properly configure a large number of email clients, take a look at the file, **Documentation/email-clients.txt** in the kernel source tree. It will help you out if you want to use your normal email client to send patches. But there's another way...

Git has a way to send patches created with **git format-patch** via email to the developers who need it. The **git send-email** command handles this all for us:

```
$ git send-email --to gregkh@suse.de --to wfp5p@virginia.
edu \
  --cc devel@driverdev.osuosl.org \
  --cc linux-kernel@vger.kernel.org \
  0001-Staging-comedi-fix-brace-coding-style-issue-in-ni_la.
patch
```

will send the patch we created to the proper developers and CC the proper mailing lists.

What next?

Now that you have successfully created a patch and sent it off, the developer who you sent it to should respond by email in a few days with either a nice, "thanks for the patch, I have applied it," or possibly some comments for changes that you should make in order to get it accepted. If you have not heard anything within a week, send it again. You shouldn't worry about being annoying; persistence is the key to attracting the attention of a busy kernel subsystem maintainer.

So there you have it, the simple steps involved in creating, committing, and sending off a Linux kernel patch. Hopefully this means that everyone reading this article will soon send in their own kernel patch, and after you've had fun doing that you'll continue to contribute to the largest software project in the history of computing. LXF