Intro

This tutorial will cover how to get your first patch submitted. You have three choices for how to complete this tutorial:

1. Run Linux in VMPlayer from Windows.
2. Run Linux natively on your own machine.
3. Run Linux within VMPlayer on Linux.

We recommend running Linux natively. Most Linux kernel developers run Linux natively, so you may as well get used to it. 🙂 If you want to run Linux in VMPlayer, follow these directions. Note, you will not be able to compile the Linux kernel on a Mac, because the filesystem defaults to case-insensitive.

This tutorial assumes you are running Ubuntu or Debian. If you are running Fedora, Suse, Arch, or Gentoo, the package installation commands or package names may be slightly different. Ask for help on #kernel-newbies on irc.oftc.net if you get stuck. If (and only if) you are an applicant for Outreachy, then you should ask for help on #kernel-outreachy on irc.oftc.net.

Additionally, we highly recommend that applicants have a stable internet connection, with no download caps. Communication over IRC can be difficult if your internet connection keeps dropping or has a big lag time, so you need a stable internet connection. Downloading the initial kernel will use over 5 GB of data, which will easily blow through a standard 3G capped plan. We recommend making sure you have cable internet, or an unlimited 3G plan.

Overview

This tutorial will show you how to:

- Setup your tools
- Explore the kernel tree
- Play with some git basics
- Update your kernel
- Make a driver change
- Start creating your first patch
- Understand patch best practices
- Find a driver to clean up
- Committing your changes
- Submit a patch
- Send your patch to your mentors
- Responding to emails
- Revising your patches
- Submitting a patchset

Setup your tools

You'll need to install, configure, and download some software to get started. You should follow the setup directions here.

Setup vim

(Note, if you're running a native Linux install and you're used to another editor like emacs or nano, you can still use that editor, and you can skip this step. You may not be able to use gvim.)

> Tip: Vim is a simple text editor that has a couple modes. It starts out in standard mode, and you can move the cursor down or up with the arrow keys (or the 'j' or 'k' keys), and move the cursor left or right with the arrow keys (or the 'h' and 'l' keys). You can go into "Insert mode" by typing 'i'. Now you can change text. To get back into standard mode, type <Escape>. To write a file, get into standard mode, and type :w<enter>. To quit vim, type :q<enter>. If you want to learn more about vim, the VIM adventures game is quite fun.

First, we need to make sure to enable the C indentation module in our default text editor (vim). Turning on this module will ensure that lines automatically get indented to the right level as you're editing. It saves you from hitting <tab> a lot. You can turn on automatic indentation based on the file type. First run:

```
vim ~/.vimrc
```

Then add this line:

```
filetype plugin indent on
```

You'll also want to add a couple more lines, to turn syntax highlighting on, and show the file name in the terminal title bar:

```
syntax on
set title
```

Most distributions compile vim so that 8 space tabs are the default. If you find they're not the default, you will need to add the following line to your .vimrc:

```
set tabstop=8
set softtabstop=8
set shiftwidth=8
set noexpandtab
```

That's equivalent to running this command in vim:

```
:set tabstop=8 softtabstop=8 shiftwidth=8 noexpandtab
```

Setup vim as your default editor

Next, we'll need to set up mutt to use vim as the default editor, instead of nano. Run:

```
sudo update-alternatives --config editor
```

and chose /usr/bin/vim.basic as the default editor.

Set up email

To be able to send Linux kernel patches, you'll need to be able to send email from the Linux VM image (or your computer that is natively running Linux). The VM image comes installed with esmtp, and if you were following the native Linux install directions you should have that installed on your computer as well. Esmtp is a mail transport agent. It routes email to your mail server, such as gmail. To know what information to give esmtp, you will need to look up your mail server settings.

You will also need to install an approved email client, and use it to respond to message on the outreachy-kernel mailing list. These instructions assume you're using mutt, but you may find a GUI mail client like Evolution to be easier to use.

Gmail set up

In gmail, go click the gear icon, go to "Settings", go to the tab "Forwarding POP/IMAP", and click "Enable IMAP", then click on "Save Changes".

Then click the "Configuration instructions" link at the very bottom of the page. Note the outgoing mail server information in "Step 2", and copy it into the .esmtprc file, as shown in the next section.

If your gmail account uses two-step verification, then you will need to generate and use App Password.

Configure esmtp

> **Note:** If you already have another mail transfer agent (MTA) installed, you do not need to install esmtp. Instead, change the .muttrc file "sendmail" line to be the path to your MTA. Mutt uses ssmtp by default, so if your MTA is ssmtp, you can leave that line out entirely.

First, create a .esmtprc file with the right permissions:

```
touch ~/.esmtprc
chmod g-rwx ~/.esmtprc
chmod o-rwx ~/.esmtprc
```

Edit the .esmtprc in your home directory, and add lines like this:

```
identity "my.email@gmail.com"
hostname smtp.gmail.com:587
username "my.email@gmail.com"
password "ThisIsNotARealPassWord"
starttls required
```

Next, set up the mail client, mutt, with some defaults, by creating a .muttrc file in your homedirectory:

```
set sendmail="/usr/bin/esmtp"
set envelope_from=yes
set from="Your Name <my.email@gmail.com>"
set use_from=yes
set edit_headers=yes
```

Test your email setup

Next, let's send a test email message with mutt. Run this command:

```
mutt
```

Say "no" to creating an inbox for now. Type 'm' to create a new message. Specify your own email address (or a secondary email) to send the test message to. Set the Subject however you want to. Type a message in the body, and then save and quit. Hit 'y' to send the message, hit 'e' to edit the message again, or hit 'q' to abort sending the message.

Look in your email to double check you received a message. If you send the email to yourself, for some mail services like gmail, the message will not show up in your inbox, and you will have to look in your Sent Mail folder.

If mutt is not working, try

```
mutt -d 2
```

This will cause the creation of files with names beginning .muttdebug, followed by 0, 1, etc., that can help you find the problem.

Setup git

First, you need to tell git what your name and email address is, so that it can be used in the authorship information in the git commit. Create a file called `.gitconfig` and add lines like these to it:

```
[user]
   name = Your Name
   email = your.email@example.com
```

Make sure that the email you specify here is the same email you used to set up sending mail. The Linux kernel developers will not accept a patch where the "From" email differs from the "Signed-off-by" line, which is what will happen if these two emails do not match.

Make sure you store your full, legal name in the 'name' line. By adding your Signed-off-by line to a patch, you are certifying that you have read and understood the Developer's Certificate of Origin. Please read though that document before you send patches to the kernel.

Explore the kernel tree

First, open a terminal, by clicking the black screen icon with the " >_ " text in it.

> Tip: You can exit out of a terminal tab or window by pressing `<CTRL>d` at any time. This is the recommended way of closing the terminal, since it won't kill any processes you have running in the background. Get used to exiting the terminal this way by opening and closing the terminal a couple times.

Change directories to your git checkout you set up earlier:

```
cd git/kernels/staging/
```

This is the Linux kernel tree. You can explore it by using the `` `ls` `` and `` `cd` `` commands. If you run `` `ls` ``, you'll see several different folders:

```
intern@ubuntu:~/git/kernels/staging$ ls
COPYING  Documentation  Kconfig      Makefile  arch   certs   drivers   fs       init  kernel  mm   samples  security  tools  virt
CREDITS  Kbuild         MAINTAINERS  README    block  crypto  firmware  include  ipc   lib     net  scripts  sound     usr
```

There's more to this directory than meets the eye! If you run ls -A, you'll see there's a hidden directory called `.git`. This contains all the meta information that git uses to track branches, remote repositories, and changes to files in the local directory.

You can view the commit history by running

```
git log
```

If you want a more compact form, you can run a command to see just the "short description" for each commit, with an abbreviated git commit ID:

```
git log --pretty=oneline --abbrev-commit
```

Play with some git basics

Git is a distributed revision control system, which means you can hack on your version of the code without having to coordinate with other developers. Think of your git checkout as a separate copy of the kernel respository.

Git includes support for branches. Each branch can contain a completely different set of patches. Kernel developers typically use one branch per patchset. For example, you might have one branch that includes bug fixes, and another branch that contains commits for a new feature you're working on.

You can run `` `git branch` `` to see which branch you're on, and what other branches are available:

```
intern@ubuntu:~/git/kernels/staging$ git branch
 * staging-testing
```

In this case, there is only one branch, called staging-testing. The star indicates that the "staging-testing" branch is the one you are currently on. In git speak, we say that you currently have the master branch "checked out".

Create a new branch called 'first-patch', and checkout that branch by running:

```
git checkout -b first-patch
```

Now if you run git branch, you'll see that there are two branches, and you are currently on the "first-patch" branch:

```
intern@ubuntu:~/git/kernels/staging$ git branch
  staging-testing
* first-patch
```

You can also use the git branch command to show branches on the staging remote repository. Run the command:

```
intern@ubuntu:~/git/kernels/staging$ git branch -a
* first-patch
  staging-testing
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/staging-linus
  remotes/origin/staging-next
  remotes/origin/staging-testing
  remotes/origin/test
```

The first remote repository that is used to create the git checkout is called "origin". For now, just remember that "origin" means Greg Kroah-Hartman's staging git repository. Here, you can see the staging remote has five branches: master, staging-linus, staging-next, staging-testing, and test. The staging-linus branch contains bug fix patches for the current kernel release candidate, and the staging-next branch contains patches for the next kernel release. Your patches will all go into staging-testing (since they will be code clean up, not bug fixes), so you want to base all your branches on the staging-testing branch. Greg first applies patches to staging-testing. They are moved to staging-next shortly thereafter.

Update your kernel

When you create your first application clean-up patches, you want to create them on top of the latest commit from the staging-testing tree. If your patch is out-of-date and doesn't apply to the latest tree, it may be rejected. You'll need to use git to fetch the latest changes:

```
git fetch origin
```

The third word in that command is the name of the remote repository you are fetching from. In this case, it's origin, which is the remote repository we initially cloned from (the staging repository).

That command will fetch the changes from the remote, but it won't actually change in files in the working copy (i.e. the files in this directory). If you run:

```
git log
```

You will see that your current working directory still points to the original commit. So where are the staging tree current changes?

The answer is that git stores the changes in a special hidden directory called `.git`. You can view the history of the staging repository by giving git log the "staging-testing" branch of the "origin" remote repository:

```
git log origin/staging-testing
```

Next, we need to update our branch to include the changes in the staging tree. The safest way to do this is to "rebase" your branch. This means that if you have any commits on your branch, they will be placed on top of the staging tree commits. Sometimes you may have to edit your commits if there are conflicts, but you should ask your mentor for help with this. For now, run:

```
git rebase origin/staging-testing
```

If you run `` `git log` `` to show your staging branch history and then `` `git log origin/staging-testing` `` to show the staging-testing branch history, you should see that they have exactly the same commits.

Configure the kernel

The next step is to create a configuration file, compile the new kernel, and install it.

The first thing to know is that the Linux kernel is completely configurable. Each driver can be separately configured to be installed or not. There are three choices for driver installation:

- disable the driver completely,
- build the driver into the main kernel file (vmlinuz),
- or build it as a module.

If you build the driver into the main kernel file, it will be loaded at boot time. The downside is that the kernel will have to load more code at boot for drivers that may not even correspond to hardware on the system. To avoid this, kernel developers often compile drivers as "modules". A module is a stand-alone .ko driver file that is loaded when the kernel detects hardware that matches the driver. For example, you could configure your wifi driver as a module, and the kernel will load it when it detects the wifi card.

The Linux kernel make system uses a special file called `.config` that stores what drivers are compiled in, or compiled as modules. Most Linux distributions store the .config file they used to compile your distro kernel in the /boot/ directory:

```
intern@ubuntu:~/git/kernels/staging$ ls /boot/
System.map-4.8.0-2-amd64  System.map-4.9.0-rc8-amd64  config-4.9.0-1-amd64    grub                     initrd.img-4.9.0-1-amd64    lost+fo
und             vmlinuz-4.9.0-1-amd64
System.map-4.9.0-1-amd64  config-4.8.0-2-amd64        config-4.9.0-rc8-amd64  initrd.img-4.8.0-2-amd64  initrd.img-4.9.0-rc8-amd64  vmlinuz
-4.8.0-2-amd64  vmlinuz-4.9.0-rc8-amd64
```

You can duplicate the distro's configuration by copying one of the config-* files to a .config file in your git tree. This has already been done for you in the VM image.

You can read more about configuring a kernel here.

Compile the kernel

Next, you'll need to run `make` to compile your new kernel. Optionally, make can take a flag that indicates how many threads to spawn to start separate compilations. Usually you want to pick a number that is equal to the number of CPUs you have in your machine. For example, if you had a dual core system, you would run:

```
make -j2
```

That may take a while. I would suggest reading some of the Linux Device Drivers book while you're waiting.

Make a driver change

These next couple of steps will allow you to make a change to a driver, and test that you've correctly compiled and installed the modified driver.

Modifying a driver under the VM

If you are running Linux in a VM, follow these directions. Otherwise, if you are running Linux natively on your machine, go to the next section.

One driver that's included in all VM images is the e1000 driver, the Intel ethernet driver. If you're running Linux natively, you will need to find a different driver. See the next driver section for how to find an appropriate driver.

The e1000 driver is found in the networking portion of the kernel:

```
intern@ubuntu:~/git/kernels/staging$ ls drivers/net/ethernet/intel/e1000/
e1000_ethtool.c  e1000.h  e1000_hw.c  e1000_hw.h  e1000_main.c  e1000_osdep.h  e1000_param.c  Makefile
```

Let's make a small change to the probe function of the e1000 driver. A probe function is called when the driver is loaded. Let's edit e1000_main.c:

```
vim drivers/net/ethernet/intel/e1000/e1000_main.c
```

Next, find the probe function. You can search for text by typing '/' in standard mode. Once you've found the probe function, add a printk line to it:

```
static int e1000_probe(struct pci_dev *pdev, const struct pci_device_id *ent) {
         struct net_device *netdev;
         struct e1000_adapter *adapter;
         struct e1000_hw *hw;

         printk(KERN_DEBUG "I can modify the Linux kernel!\n");
         static int cards_found = 0;
```

Then type `:wq<enter>` to save the file and quit.

A printk function causes a message to be written to the kernel log buffer, which can then be viewed using the `dmesg` command.

Modifying a driver on native Linux

Your native Linux system will have many different drivers than the ones loaded on a Linux system running in VMPlayer. You will not necessarily have hardware that the e1000 driver can run on. Instead, you must find out which drivers are loaded on your system, and modify one of them.

First, use `lsmod` to see what drivers are loaded, and pick a name from that list to modify. If you have a variant of the e1000 driver, like e1000e, you may want to use that. Or you can find your wireless driver and modify that.

Once you have the name of a driver, it's time to find out where the .c and .h files for that driver are in the Linux kernel repository. You can do that by searching through the Makefiles to find out what C files are compiled into the driver binary. The best way to do that is with `` `git grep` ``. Unlike normal grep, git grep only searches through checked-in files in the repository. Normal grep will also search the binary files, and even the .git directory, which isn't useful and wastes your time.

For example, if you wanted to search for the files that are compiled into the xhci-hcd driver, you would run this command:

```
git grep xhci-hcd -- '*Makefile'
```

Once you've identified the files for your driver, you will need to make a change to the probe function as described in the previous section.

Compile your changes

Recompile your kernel, by running `make` (with an optional `-jN` flag):

```
make -j2
```

You may need to fix some compilation errors. Also fix any new warnings that are due to your changes. In the Linux kernel, we strive to make sure that drivers do not produce warnings on anyone's system (this includes 32-bit and 64-bit systems, as well as different architectures, such as PPC, ARM, or x86). New features or bug fix patches that add additional warnings may not get merged.

Install your changes

After you've compiled the driver, you need to install your changes by running:

```
sudo make modules_install install
```

Test your changes

Since you've compiled a completely new kernel, you need to reboot into that new kernel in order to test your module changes. Reboot your VM (or computer), and then run:

```
dmesg | less
```

Search for your printk in the log file by typing "/I can modify". You should be able to find this message within the driver output during boot. If you don't see this message, ask for help on the #kernel-outreachy IRC channel on irc.oftc.net, or on the outreachy-kernel mailing list

Revert your changes

Since that was just a simple test, and you probably don't want to commit that change, you can revert your changes. Exit out of your editor by typing `:q<enter>` and running this command:

```
git reset --hard HEAD
```

That will revert **ALL FILES** in your current working directory to the last known commit (the HEAD commit), wiping out all your uncommited changes. Read the git reset manual for more information on ways to reset the state for specific files.

Start creating your first patch

Next, you'll get to do some useful modifications to the kernel, create your first git commit, and send out your first patch. Before you make your first commit using git, you'll need to do some setup.

Git post-commit hooks

Git includes some "hooks" for scripts that can be run before or after specific git commands are executed. The "post-commit" hook is run after you make a git commit with the `` `git commit` `` command.

Linux kernel developers have very stringent guidelines for coding style. They're so picky, they created a script called checkpatch.pl that you can run over your patches to make sure the patch complies with the kernel coding style.

To ensure that you create good commits that comply with the coding style, you can run checkpatch.pl over your commit with the "post-commit" hook. Note that running checkpatch after the commit, checks the patch file you've created - not just the source code diff. Therefore it will catch more issues - spelling errors in log messages, spacing in log messages, warnings about adding/removing files, etc.

If you already have a `.git/hooks/post-commit` file, move it to `.git/hooks/post-commit.sample`. git will not execute files with the .sample extension.

Then edit the `.git/hooks/post-commit` file to contain only the following two lines:

```
exec git show --format=email HEAD | ./scripts/checkpatch.pl --strict --codespell
```

Make sure the file is executable:

```
chmod a+x .git/hooks/post-commit
```

If you don't already have /usr/share/codespell/dictionary.txt, get it:

```
apt-get install codespell
```

After you commit, this hook will output any checkpatch errors or warnings that your patch creates. If you see warnings or errors that you know you added, you can amend the commit by changing the file, using `git add` to add the changes, and then using `git commit --amend` to commit the changes.

Checking that the post-commit hook works

You can check that the post-commit hook is working by adding a deliberate change that will trigger checkpatch (such as adding a really long line or adding trailing whitespace to a line), and then attempting to run `git commit`. After you commit, you should see the additional checkpatch.pl warning or error.

Note that you will need to place this hook in any/every tree in which you build patches.

If your post-commit hook is not working, please ask for help on IRC.

Understand patch best practices

Before you create your patch, you need to understand how to create good patches. Otherwise your patches will be less likely to be accepted by maintainers, and you will have to go through more revisions than necessary.

First, read about PatchPhilosophy. That document will help you create patches that are easy to read, and have a better chance of being applied by maintainers. Please also read CodingStyle (which is also available in the kernel code repository under Documentation). This will help you understand how to write code that the Linux kernel community will accept, and the rules here are the basis for the script checkpatch.pl.

Note that checkpatch.pl is failable! Use your best judgement when deciding whether it makes sense to make the change checkpatch.pl suggests. The end goal is for the code to be more readable. If checkpatch.pl suggests a change and you think the end result is not more readable, don't make the change. For example, if a line is 81 characters long, but breaking it makes the resulting code look ugly, don't break that line. Please read the CheckpatchTips page for how to avoid common mistakes when cleaning up drivers.

Find a driver to clean up

The staging tree, in `drivers/staging/` is full of drivers that are not quite up to kernel coding style, or that use deprecated API. Drivers get placed here in order to get cleaned up. Some drivers have a TODO file in their parent directory, that lists things that need to be done to it:

```
find drivers/staging -name TODO
```

You can either tackle one of those TODO items, or you can do a simple coding style cleanup.

drivers/staging contains both drivers that are on their way into the kernel and those that are on their way out of the kernel. It would be better to avoid working on the latter. Some drivers that are on their way out of the kernel as of February 2015 are

```
i2o
line6
media/parport
media/tlg2300
media/vino
```

Running checkpatch.pl

If you pick a driver in staging, you can run the script that checks whether a file conforms to kernel coding style:

```
perl scripts/checkpatch.pl -f drivers/staging/android/* | less
```

Pick a warning, and try to fix it. For your first patch, only pick one warning. In the future you can group multiple changes into one patch, but only if you follow the PatchPhilosophy of breaking each patch into logical changes.

Note that there is a lot of variation in the warnings generated by checkpatch. Sometimes the code is clearly not ideal and the fix is obvious. Other cases might be a matter of personal preference, or might require specialized knowledge about the code to make the correct change. It may be helpful to look back at old messages on the mailing list, to see if a similar change has been rejected, and why.

In addition to checkpatch cleanups, you can also try other bug finding tools in the kernel. Here is some information on using Coccinelle

Recompiling the driver

You'll need to make sure the driver you're changing is configured as a module. Run:

```
make menuconfig
```

This opens up a text-based GUI that allows you to explore the configuration options.

Use the arrow keys to go to `Device Drivers ->` and hit <enter>. Then go down to `Staging drivers` . At any time, you can hit '?', which will show you the help text for that kernel configuration option. You can search for the driver you're modifying by '/', in order to get the driver's longer name. Make sure the driver you're working on is compiled as a module ('M'), instead of being built-in ('*'). You can change a driver to being compiled as a module by typing 'm' when the driver is selected in the menu. Hitting <enter> will change the driver to being built-in.

Once you've enabled the driver you're modifying, use <tab> or the right arrow key to move the cursor from 'Select' to 'Exit' and hit <enter>. Continue to do this until you get to the main menu. When it asks you to save your configuration, chose 'Yes'.

Then recompile the kernel with:

```
make -j2
```

You should reboot your kernel, load the driver with `modprobe` . You'll be able to see that the driver is loaded by running `lsmod` . Loading the driver at least makes sure that the driver probe function works.

> **Note:** Do not work on drivers that show that they depend on CONFIG_BROKEN. If you search for a driver after running `` `make menuconfig` `` , and you notice the "Depends on:" line includes BROKEN, do not work on this driver.

Compiling only part of the kernel

There are several ways to compile only part of the kernel:

- make path/file.o: This compiles only a single file. It may not be sufficient to check changes that affect interactions with other files. It is possible to build files that are disabled in .config this way, but that can fail in some cases, e.g. when the file includes architecture-specific headers.
- make path, e.g. `make drivers/staging` : This always succeeds. It does nothing at all because the directory exists.
- make path/, e.g. `make drivers/staging/` : This descends into that directory to build all the files in it, but does not try to link the modules.
- `make M=drivers/staging` : This seems to try to link the modules in a previously build vmlinux file. After changing options in .config or rebasing on a newer git tree, this can fail unless all other files are built once using `make -j2` .

Make path/ could be a reasonable choice for a localized change within a single driver, or for a change localized to drivers staging. In any case, as a minimum always check that compilation produces a .o file for every file that your patch changes. If there is no .o file, you have not compiled the file, and you need to look for other compiler options.

Driver dependencies

Sometimes it's hard to find your driver. Maybe you searched for the CONFIG option, but you can't find it in the menus where it should be. This may be because a driver or subsystem it depends on is not enabled, and so this driver is hidden from the menu. You need to enable all the dependencies in order to make the menu option visible, and then you can enable the driver you're modifying.

For example, say I was modifying drivers/usb/host/xhci-ring.c. If I look in the Kconfig file in the parent directory (drivers/usb/host/Kconfig), I can see an option for the xHCI driver:

```
config USB_XHCI_HCD
         tristate "xHCI HCD (USB 3.0) support"
         depends on USB && USB_ARCH_HAS_XHCI
         ---help---
           The eXtensible Host Controller Interface (xHCI) is standard for USB 3.0
           "SuperSpeed" host controller hardware.

           To compile this driver as a module, choose M here: the
           module will be called xhci-hcd.
```

Now, I run `` `make menuconfig` `` and type '/' to search for USB_XHCI_HCD. The search entry shows:

```
Symbol: USB_XHCI_HCD [=m]
 Type  : tristate
 Prompt: xHCI HCD (USB 3.0) support
   Location:
    -> Device Drivers
 (1)   -> USB support (USB_SUPPORT [=y]
   Defined at drivers/usb/host/Kconfig:20
   Depends on: USB_SUPPORT [=y] && USB [=n] && USB_ARCH_HAS_XHCI [=y])
```

Look at the "Depends on" line. This is a boolean equation that represents the driver dependencies that need to be enabled in order to even show the driver option in the menus. A 'y' means the driver or subsystem is built into the kernel and an 'm' means the driver is built as a module. Both these options are equivalent to '1' in boolean logic. A 'n' means the driver or subsystem is disabled. An 'n' is equivalent to a '0' in boolean logic.

> Tip: If you don't know boolean logic, you can take a look at these tutorials. If you just need a brush up on boolean logic, here's a crib sheet. If you're lazy, here's a boolean algebra calculator, or you can use the "Programming Mode" for the calculator application installed in Ubuntu by default.

In this case, the xHCI driver config option is not being shown in the menus because USB is set to 'n'. If I search for USB, I see lots of results, and finally find this relevant one:

```
Symbol: USB [=n]
 Type  : tristate
 Prompt: Support for Host-side USB
   Location:
     -> Device Drivers
 (1)   -> USB support (USB_SUPPORT [=n])
   Defined at drivers/usb/Kconfig:51
   Depends on: USB_SUPPORT [=n] && USB_ARCH_HAS_HCD [=n]
```

If I look under the Device Drivers menu, I can find "USB Support" and set it to 'm'. Once that's done, I can find CONFIG_XHCI_HCD under the "USB Support" menu.

It may take recursively enabling several different configuration options before you can even see your driver in the menu. Ask for help if you're stuck.

> **Note:** Do not work on drivers that show that they depend on CONFIG_BROKEN. If you search for a driver after running `make menuconfig` , and you notice the "Depends on:" line includes BROKEN, do not work on this driver.

Reloading modules

If you're running a kernel that has the same release version (`uname -r`) as the new code you're compiling, you can test your changes without rebooting. Simply install the module in /lib/modules, and unload and reload the driver:

```
make -j2 && sudo make modules_install
sudo modprobe -r <module_name>
sudo modprobe <module_name>
```

How do you know what the module name is? If you've compiled the driver as a module, there should be a .ko file in the parent directory. For example, after we configure the android driver to be compiled as a module, we can run this command:

```
intern@ubuntu:~/git/kernel/staging$ ls drivers/staging/android/*.ko
 drivers/staging/android/alarm-dev.ko  drivers/staging/android/logger.ko
```

So, there are two drivers that we need to load with modprobe. You can load those drivers one at a time by passing modprobe the base filename:

```
sudo modprobe alarm-dev
```

To ensure the driver got loaded, you can run:

```
lsmod | less
```

Committing your changes

In this example, assume we've addressed a warning in the android driver, modified the file, recompiled the driver, and tested our changes.

Viewing your changes

Git keeps track of changes in the working directory. Git can be told to ignore binary files (like .o or .ko files), so it won't track changes to those files. You can see which files have been modified by running:

```
git status
```

git can also show you a diff stat of what changed:

```
git diff
```

Commit your changes

Assuming we want to include all of our changes in one git commit, you can use git to add the changed file to the list of changes to be committed (the "staging area"):

```
git add <file>
```

If you run `git diff` again, you'll notice it doesn't list any changed files. That's because, by default, git diff only shows you the unstaged changes. If you run this command instead, you'll see the staged changes:

```
git diff --cached
```

That command will show you the changes to be committed.

Reverting your staged changes

If you don't want to commit those changes, you can remove those changes from the staging area by running:

```
git reset <file>
```

Committing parts of files

You can also add parts of files to the staging area by using the

flag:

```
git add -p
```

That will allow you to add hunks of the file to the staging area, or even edit hunks that you want to commit. This is useful, for instance, if you've made whitespace changes, and also made a camel-case variable name fix, but those changes are on the same line. You can edit the line to revert the camel-case name change, and just add the whitespace change to the staging area. Then when you commit, you will just be committing the whitespace change.

Committing changes

Finally, you can commit your staged changes:

```
git commit -s -v
```

That will add the Signed-off-by line that is needed at the end of your patch description. The -v flag will show you the diff that you're committing. This is very useful to decide whether you are committing the correct code.

Make sure that when you create your patch, you follow the PatchPhilosophy guidelines. Make sure to include a blank line between your short description (what will become the Subject line of your patch) and the body of your patch. Make sure there is a blank line between the body of your patch and your Signed-off-by line. **Again, make sure you read PatchPhilosophy before you make your first commit.**

Editing your commits

If you should need to edit your commits, please see the "Editing commits" and "Editing patchsets" sections.

Viewing your commit

Make sure your commit looks fine by running these commands:

```
git show HEAD
```

This will show the latest commit. If you want git to show a different commit, you can pass the commit ID (the long number that's shown in `` `git log` ``, or the short number that's shown in `` `git log --pretty=oneline --abbrev-commit` ``). Read the "Specifying Revisions section" of the `` `git rev-parse` `` manual page for more details on what you can in place of a commit ID.

You'll also want to make sure your commit looks fine when you run these two commands:

```
git log
```

```
git log --pretty=oneline --abbrev-commit
```

Submit a patch

The first step to sending a patch is to figure out who to send it to. For this, you need to find the maintainer of the code you're patching, and Cc the correct mailing list. If you simply send it off to LKML, it will get ignored. The perl script in the kernel source directory scripts/get_maintainer.pl will take either take a git commit or a file, and tell you who to send your patch to:

```
$ git show --pretty=oneline --abbrev-commit HEAD
cb9a537 staging: most: constify snd_pcm_ops structure
diff --git a/drivers/staging/most/aim-sound/sound.c b/drivers/staging/most/aim-s
index 9c64580..21fa0df 100644
--- a/drivers/staging/most/aim-sound/sound.c
+++ b/drivers/staging/most/aim-sound/sound.c
@@ -457,7 +457,7 @@ static snd_pcm_uframes_t pcm_pointer(struct snd_pcm_substrea
 /**
  * Initialization of struct snd_pcm_ops
  */
-static struct snd_pcm_ops pcm_ops = {
+static const struct snd_pcm_ops pcm_ops = {
        .open       = pcm_open,
        .close      = pcm_close,
        .ioctl      = snd_pcm_lib_ioctl,

$ git show HEAD | perl scripts/get_maintainer.pl --separator , --nokeywords --nogit --nogit-fallback --norolestats --nol
Greg Kroah-Hartman <gregkh@linuxfoundation.org>,devel@driverdev.osuosl.org,linux-kernel@vger.kernel.org
$ perl scripts/get_maintainer.pl --separator , --nokeywords --nogit --nogit-fallback --norolestats --nol -f drivers/staging/most/aim-sound/
sound.c
Greg Kroah-Hartman <gregkh@linuxfoundation.org>
```

The many arguments to get_maintainer.pl cause the output to be in the form of a comma-separated list and cause only the actual maintainers of the affected files to be included. However, for IIO drivers, please also include ✉linux-iio@vger.kernel.org . On the other hand, the maintainers of staging/vc04_services would prefer not to be CCd on Outreachy patches. Please CC Greg explicitly on these patches: Greg KH <✉gregkh@linuxfoundation.org >. More generally, for non-outreachy patches, drop the -nol argument, to include mailing lists.

The second step to submitting a patch is to create and send a patch as an email. You cannot send patches as attachments to the mailing list. Instead, you will have to craft a special email, and send the patch inline.

There are two ways to send a patch. Either way is correct, and we suggest you get comfortable with sending patches both ways. We suggest sending your first patch to yourself, in order to make sure it works.

Creating a patch to send with mutt

First, create a patch that describes the change, using `` `git format-patch` ``. That command takes a starting commit ID (and optionally) an ending commit ID, in order to create patches for the commit **after** the starting commit ID. We want to create a patch for the first commit in our history (the HEAD commit). To specify the commit before the HEAD commit, you can use either "HEAD^" or "HEAD~".

```
git format-patch -o /tmp/ HEAD^
```

The -o flag specifies where to put the patch. If you've run the command correctly, you should see the command output a filename in /tmp/. If it outputs nothing, you've forgotten the '^'.

Sending your patch with mutt

Next, tell mutt to use that patch as a draft email, with the -H flag:

```
mutt -H /tmp/0001-<whatever your filename is>
```

Send this first patch to yourself. Leave the Subject line intact. Now you can see how your patch translates to an inline email. The git short description becomes the email subject line, and the patch body and diff become the body of the email.

Write the mail, quit out of vim, and send the mail with 'y'.

Sending patches with git send-email

You can also send a patch with the command-line tool `` `git send-email` ``. You can either pass it the file that `` `git format-patch` `` generated, or you can give it the same commit ID range you gave `` `git format-patch` ``:

```
git send-email --annotate HEAD^
```

You will need to add the following information to your .gitconfig file:

```
[sendemail]
   smtpserver = /usr/bin/esmtp
```

git send email will prompt you with who to send the message to, and other odd questions:

```
Who should the emails be sent to (if any)?
Message-ID to be used as In-Reply-To for the first email (if any)?
```

Put in your mentor's address in the first line, and leave the second blank unless you want it to appear as a thread in an existing conversation.

At this point `` `git send-email` `` will look for Cc: lines in your commit message, and include them in the email headers. It will then show you the resulting email header, and ask you to confirm:

```
Send this email? ([y]es|[n]o|[q]uit|[a]ll):
```

More help with git send-email

More comprehensive documentation, with help for those who use gmail

Send with 'y' (or 'a': git send-email can be used to send multiple commits at once).

Tips and Tricks

Please read the patch tips and tricks page for an explanation of patch tags (e.g. Reviewed-by and Signed-off-by), and when to use [PATCH] vs. [RFC] in your patch subject prefix.

Send your patch to your mentors

Once you've send your patch to your own email, and ensured that it looks fine, it's time to send your patch off.

Send your patch to the maintainer and lists that the get_maintainer.pl script tells you to.

If you are an Outreachy applicant, please additionally send your patches to outreachy-kernel mailing list .

Responding to emails

When responding to emails on the mailing list, it's important to use a communication style consistent with what the open source community expects. Please make sure you use one of the standard email clients listed in Documentation/email-clients.txt . Do NOT use the gmail web interface to respond to patch feedback, as it line-wraps the mail (even when in plain text mode). Do NOT use outlook, as it mangles patches (turns tabs into spaces).

Don't include quotes in your signature.

Responding inline

When responding to email, make sure you respond inline, rather than top-posting. This is a good example of responding inline .

Make sure your email client appends '>' characters to inline mail when you respond to it.

When you reply inline to a message, the lines you type shouldn't have a '>' symbol at the beginning of the line. Think of the carrot symbols as being similar to code indentation. When you follow the last '>' symbol up to the first "wrote" line where the '>' was introduced, you can find out who wrote that block of text.

Take a look at this example email with three people who are responding inline:

```
From: Kludge Crufty <example@email.com>
Subject: Design decisions for next release

On Fri, Sep 12, 2014 at 03:00:56PM -0700, Baz Quux wrote:
> On Fri, 12 September 2014 at 02:30:17PM -0700, Foo Bar wrote:
> >
> > I think we should do X.
>
> I think we should do Y.

I think we should do Z.

Kludge
```

The email was sent by Kludge. Kludge is responding to an email sent by Baz at 3PM. Baz was responding to an email sent from Foo at 2:30PM.

From this snippet of mail, we can tell who said what by looking at the number of '>' symbols in front of each line:

- Kludge wants to do Z, because their dialog has no '>' in front of it.

- Baz wants to do Y, because their dialog has one '>' in front, and we follow that '>' level up to Baz's "wrote" line.
- Foo wants to do X, because their dialog has two '> >' in front, and we follow the last '>' up to Foo's "wrote" line.

Another point to notice is that each responding person has put a blank line before and after their response. This makes it easier to find where some new text has been added.

Note that following inline conversation responses can be very difficult if your mail client or terminal isn't configured to show text in fixed-width fonts. This is one of the many reasons that kernel developers prefer text-based mail clients like mutt.

Revising your patches

Editing your commits

If you should need to edit your commits, please see the "Editing commits" and "Editing patchsets" sections.

Versioning one patch revision

If you receive feedback on a patch, and were asked to update the patch, you need to version the patches that you re-send. A new version number lets reviewers know you made changes to the patch, and they should review it again.

An example of what this would look like is:

```
[PATCH] Foo: Fix these things
```

And the updated versioning for a second revision:

```
[PATCH v2] Foo: Fix these things better
```

It's fairly simple to accomplish this, and there's certainly a few ways to do this. If you generate your patches using `` `git format-patch` ``, then it's simple to do this. Just add the --subject-prefix option like this:

```
git format-patch --subject-prefix="PATCH v2"
```

or whatever version you are currently on (3, 4, etc.).

Make sure to include a summary of what changed from one version of the patch (or patchset) to the next. Do not include a version change log in the patch description, because it won't make sense when the final version of the patch makes it into the kernel. Instead, edit your patches before you send them to include a change log below the "---" line. Git will ignore this change log when the patch is applied. Here's a good example of a patch with a change log:

```
Subject: [PATCH v2 1/2] USB: at91: fix the number of endpoint parameter

In sama5d3 SoC, there are 16 endpoints, which is different with
earlier SoCs (only have 7 endpoints). The USBA_NR_ENDPOINTS micro
is not suitable for sama5d3. So, get the endpoints number through
the udc->num_ep, which get from platform data for non-dt kernel,
or parse from dt node.

Signed-off-by: Bo Shen <voice.shen@atmel.com>
---
Changes in v2:
  - Make the commit message more clearer.

 drivers/usb/gadget/atmel_usba_udc.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/drivers/usb/gadget/atmel_usba_udc.c b/drivers/usb/gadget/atmel_usba_udc.c
```

Update In case of more than 2 versions, make sure to include what has changed in each version below the -- so that there is a logical flow and the maintainers do not have to dig up previous versions. The most recently changed version should be described first followed by the subsequent changes. Have a look at this patch example with 3 versions to get a better idea.

Versioning patchsets

When you send out a new version of a patchset, you could either indicate on each patch what has changed or you could include the changes in a coverletter. If you plan to indicate the changes in each patch, you need to specify what has changed in each version like in this patch example and indicate **No change** explicitly for patch versions that are unaltered. In case you decide to indicate the changes in a coverletter, here is an Example cover letter. You can include a coverletter in your patchset by using the `--cover-letter` option to `git format-patch`, e.g.

```
git format-patch -n --subject-prefix="PATCH vY" --cover-letter
```

where Y is the version of the patch you are currently sending. The cover-letter option will create a "PATCH 0/Y" that you can add a change log to. Make sure to change the SUBJECT HERE on the coverletter subject line. Note that cover letters are discarded when applying patches, so any information that you want to preserve in the git log should be in the patch descriptions, not the cover letter. The cover letter is for introducing what problem you're trying to solve, and including version change logs.

Make sure to include all of the patches you sent in the patchset before (i.e. if you sent three patches and you had to revise the second patch, send all three again).

An example of what this would look like is:

```
[PATCH 0/3] comedi: Fix these things
[PATCH 1/3] comedi: Fix the first thing
[PATCH 2/3] comedi: Fix the second thing
[PATCH 3/3] comedi: Fix the third thing
```

And the updated versioning for a second revision:

```
[PATCH v2 0/3] comedi: Fix these things
[PATCH v2 1/3] comedi: Fix the first thing
[PATCH v2 2/3] comedi: Fix the second thing
[PATCH v2 3/3] comedi: Fix the third thing
```

Submitting a patchset

Sometimes you need to send multiple related patches. This is useful for grouping, say, to group driver clean up patches for one particular driver into a set, or grouping patches that are part of a new feature into one set.

For example, take a look at this patch set:

- cover letter
- PATCH 1/7
- PATCH 2/7
- PATCH 3/7
- PATCH 4/7
- PATCH 5/7
- PATCH 6/7
- PATCH 7/7

Typically, the subject prefix for patches in the patchset are [PATCH X/Y] or [RFC X/Y], where Y is the total number of patches, and X is the current patch number. Patchsets often have a "cover letter" that is [PATCH 0/Y] or [RFC 0/Y]. A cover letter is used to explain why the patchset is necessary, and provide an overview of the end result of the patches. Cover letters are also great places to ask for help in reviewing specific patches in the patchset.

In order to create patchsets like this, you will need to use either `` `git send-email``}} or {{{`git format-patch` ``. These tools will generate the right "In-Reply-To" Headers, so that in a text mail client, the patches will appear next to each other, rather than having unrelated email in between. Otherwise, patches may get jumbled, depending on when they were received.

Using git format-patch to send patchsets

First, use `` `git log --pretty=oneline --abbrev-commit` `` to figure out the first commit ID you want to send. For example, say that my tree had this git log history:

```
b7ca36a Docs: Move ref to Frohwalt Egerer to end of REPORTING-BUGS
bf6adaf Docs: Add a tips section to REPORTING-BUGS.
bc6bed4 Docs: Expectations for bug reporters and maintainers
2c97a63 Docs: Add info on supported kernels to REPORTING-BUGS.
7883a25 Docs: Add "Gather info" section to REPORTING-BUGS.
d60418b Docs: Step-by-step directions for reporting bugs.
3b12c21 Trivial: docs: Remove six-space indentation in REPORTING-BUGS.
bb33db7 Merge branches 'timers-urgent-for-linus', 'irq-urgent-for-linus' and 'core-urgent-for-linus' of git://git.kernel.org/pub/scm/linux/
kernel/git/tip/tip
41ef2d5 Linux 3.9-rc7
```

The first commit I want to send as part of the patchset has commit ID 3b12c21. The last patch I want to send has commit ID b7ca36a. So, I want to pass the commit range `3b12c21^..b7ca36a` to `git format-patch`. (Remember, the `git format-patch` range starting commit must be the commit *before* the first commit you want to send, so we use the '^' to specify the patch before commit 3b12c21.) The command will look something like this:

```
git format-patch -o /tmp/ --cover-letter -n --thread=shallow --cc="linux-usb@vger.kernel.org" 3b12c21^..b7ca36a
```

Again, the -o flag specifies where to put the email files. The -n flag says to add numbering to each patch (e.g. [PATCH 2/5]). The --thread=shallow flag specifies that all patches will be In-Reply-To your cover letter.

That will output files into /tmp, and you can edit them in mutt in multiple terminal tabs:

```
/tmp/0000-cover-letter.patch
/tmp/0001-Trivial-docs-Remove-six-space-indentation-in-REPORTI.patch
/tmp/0002-Docs-Step-by-step-directions-for-reporting-bugs.patch
/tmp/0003-Docs-Add-Gather-info-section-to-REPORTING-BUGS.patch
/tmp/0004-Docs-Add-info-on-supported-kernels-to-REPORTING-BUGS.patch
/tmp/0005-Docs-Expectations-for-bug-reporters-and-maintainers.patch
/tmp/0006-Docs-Add-a-tips-section-to-REPORTING-BUGS.patch
/tmp/0007-Docs-Move-ref-to-Frohwalt-Egerer-to-end-of-REPORTING.patch
```