Submitting patches: the essential guide to getting your code into the kernel

For a person or company who wishes to submit a change to the Linux kernel, the process can sometimes be daunting if you're not familiar with "the system." This text is a collection of suggestions which can greatly increase the chances of your change being accepted.

This document contains a large number of suggestions in a relatively terse format. For detailed information on how the kernel development process works, see Documentation/process/development-process.rst. Also, read Documentation/process/submit-checklist.rst for a list of items to check before submitting code. If you are submitting a driver, also read Documentation/process/submitting-drivers.rst; for device tree binding patches, read Documentation/devicetree/bindings/submitting-patches.rst.

This documentation assumes that you're using `git` to prepare your patches. If you're unfamiliar with `git`, you would be well-advised to learn how to use it, it will make your life as a kernel developer and in general much easier.

Some subsystems and maintainer trees have additional information about their workflow and expectations, see :ref:`Documentation/process/maintainer-handbooks.rst <maintainer_handbooks_main>`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]submitting-patches.rst`, line 25); *backlink***
>
> Unknown interpreted text role "ref".

Obtain a current source tree

If you do not have a repository with the current kernel source handy, use `git` to obtain one. You'll want to start with the mainline repository, which can be grabbed with:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Note, however, that you may not want to develop against the mainline tree directly. Most subsystem maintainers run their own trees and want to see patches prepared against those trees. See the **T:** entry for the subsystem in the MAINTAINERS file to find that tree, or simply ask the maintainer if the tree is not listed there.

Describe your changes

Describe your problem. Whether your patch is a one-line bug fix or 5000 lines of a new feature, there must be an underlying problem that motivated you to do this work. Convince the reviewer that there is a problem worth fixing and that it makes sense for them to read past the first paragraph.

Describe user-visible impact. Straight up crashes and lockups are pretty convincing, but not all bugs are that blatant. Even if the problem was spotted during code review, describe the impact you think it can have on users. Keep in mind that the majority of Linux installations run kernels from secondary stable trees or vendor/product-specific trees that cherry-pick only specific patches from upstream, so include anything that could help route your change downstream: provoking circumstances, excerpts from dmesg, crash descriptions, performance regressions, latency spikes, lockups, etc.

Quantify optimizations and trade-offs. If you claim improvements in performance, memory consumption, stack footprint, or binary size, include numbers that back them up. But also describe non-obvious costs. Optimizations usually aren't free but trade-offs between CPU, memory, and readability; or, when it comes to heuristics, between different workloads. Describe the expected downsides of your optimization so that the reviewer can weigh costs against benefits.

Once the problem is established, describe what you are actually doing about it in technical detail. It's important to describe the change in plain English for the reviewer to verify that the code is behaving as you intend it to.

The maintainer will thank you if you write your patch description in a form which can be easily pulled into Linux's source code management system, `git`, as a "commit log". See :ref:`explicit_in_reply_to`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]submitting-patches.rst`, line 78); *backlink***
>
> Unknown interpreted text role "ref".

Solve only one problem per patch. If your description starts to get long, that's a sign that you probably need to split up your patch. See :ref:`split_changes`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]submitting-patches.rst, line 82); *backlink*

Unknown interpreted text role "ref".

When you submit or resubmit a patch or patch series, include the complete patch description and justification for it. Don't just say that this is version N of the patch (series). Don't expect the subsystem maintainer to refer back to earlier patch versions or referenced URLs to find the patch description and put that into the patch. I.e., the patch (series) and its description should be self-contained. This benefits both the maintainers and reviewers. Some reviewers probably didn't even receive earlier versions of the patch.

Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behaviour.

If you want to refer to a specific commit, don't just refer to the SHA-1 ID of the commit. Please also include the oneline summary of the commit, to make it easier for reviewers to know what it is about. Example:

```
Commit e21d2170f36602ae2708 ("video: remove unnecessary
platform_set_drvdata()") removed the unnecessary
platform_set_drvdata(), but left the variable "dev" unused,
delete it.
```

You should also be sure to use at least the first twelve characters of the SHA-1 ID. The kernel repository holds a *lot* of objects, making collisions with shorter IDs a real possibility. Bear in mind that, even if there is no collision with your six-character ID now, that condition may change five years from now.

If related discussions or any other background information behind the change can be found on the web, add 'Link:' tags pointing to it. In case your patch fixes a bug, for example, add a tag with a URL referencing the report in the mailing list archives or a bug tracker; if the patch is a result of some earlier mailing list discussion or something documented on the web, point to it.

When linking to mailing list archives, preferably use the lore.kernel.org message archiver service. To create the link URL, use the contents of the `Message-Id` header of the message without the surrounding angle brackets. For example:

```
Link: https://lore.kernel.org/r/30th.anniversary.repost@klaava.Helsinki.FI/
```

Please check the link to make sure that it is actually working and points to the relevant message.

However, try to make your explanation understandable without external resources. In addition to giving a URL to a mailing list archive or bug, summarize the relevant points of the discussion that led to the patch as submitted.

If your patch fixes a bug in a specific commit, e.g. you found an issue using `git bisect`, please use the 'Fixes:' tag with the first 12 characters of the SHA-1 ID, and the one line summary. Do not split the tag across multiple lines, tags are exempt from the "wrap at 75 columns" rule in order to simplify parsing scripts. For example:

```
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the number of pages it actually freed")
```

The following `git config` settings can be used to add a pretty format for outputting the above style in the `git log` or `git show` commands:

```
[core]
        abbrev = 12
[pretty]
        fixes = Fixes: %h (\"%s\")
```

An example call:

```
$ git log -1 --pretty=fixes 54a4f0239f2e
Fixes: 54a4f0239f2e ("KVM: MMU: make kvm_mmu_zap_page() return the number of pages it actually freed")
```

Separate your changes

Separate each **logical change** into a separate patch.

For example, if your changes include both bug fixes and performance enhancements for a single driver, separate those changes into two or more patches. If your changes include an API update, and a new driver which uses that new API, separate those into two patches.

On the other hand, if you make a single change to numerous files, group those changes into a single patch. Thus a single logical change is contained within a single patch.

The point to remember is that each patch should make an easily understood change that can be verified by reviewers. Each patch should be justifiable on its own merits.

If one patch depends on another patch in order for a change to be complete, that is OK. Simply note **"this patch depends on patch X"** in your patch description.

When dividing your change into a series of patches, take special care to ensure that the kernel builds and runs properly after each

patch in the series. Developers using `git bisect` to track down a problem can end up splitting your patch series at any point; they will not thank you if you introduce bugs in the middle.

If you cannot condense your patch set into a smaller set of patches, then only post say 15 or so at a time and wait for review and integration.

Style-check your changes

Check your patch for basic style violations, details of which can be found in Documentation/process/coding-style.rst. Failure to do so simply wastes the reviewers time and will get your patch rejected, probably without even being read.

One significant exception is when moving code from one file to another -- in this case you should not modify the moved code at all in the same patch which moves it. This clearly delineates the act of moving the code and your changes. This greatly aids review of the actual differences and allows tools to better track the history of the code itself.

Check your patches with the patch style checker prior to submission (scripts/checkpatch.pl). Note, though, that the style checker should be viewed as a guide, not as a replacement for human judgment. If your code looks better with a violation then its probably best left alone.

The checker reports at three levels:

- ERROR: things that are very likely to be wrong
- WARNING: things requiring careful review
- CHECK: things requiring thought

You should be able to justify all violations that remain in your patch.

Select the recipients for your patch

You should always copy the appropriate subsystem maintainer(s) on any patch to code that they maintain; look through the MAINTAINERS file and the source code revision history to see who those maintainers are. The script scripts/get_maintainer.pl can be very useful at this step. If you cannot find a maintainer for the subsystem you are working on, Andrew Morton (akpm@linux-foundation.org) serves as a maintainer of last resort.

You should also normally choose at least one mailing list to receive a copy of your patch set. linux-kernel@vger.kernel.org should be used by default for all patches, but the volume on that list has caused a number of developers to tune it out. Look in the MAINTAINERS file for a subsystem-specific list; your patch will probably get more attention there. Please do not spam unrelated lists, though.

Many kernel-related lists are hosted on vger.kernel.org; you can find a list of them at http://vger.kernel.org/vger-lists.html. There are kernel-related lists hosted elsewhere as well, though.

Do not send more than 15 patches at once to the vger mailing lists!!!

Linus Torvalds is the final arbiter of all changes accepted into the Linux kernel. His e-mail address is <torvalds@linux-foundation.org>. He gets a lot of e-mail, and, at this point, very few patches go through Linus directly, so typically you should do your best to -avoid- sending him e-mail.

If you have a patch that fixes an exploitable security bug, send that patch to security@kernel.org. For severe bugs, a short embargo may be considered to allow distributors to get the patch out to users; in such cases, obviously, the patch should not be sent to any public lists. See also Documentation/admin-guide/security-bugs.rst.

Patches that fix a severe bug in a released kernel should be directed toward the stable maintainers by putting a line like this:

```
Cc: stable@vger.kernel.org
```

into the sign-off area of your patch (note, NOT an email recipient). You should also read Documentation/process/stable-kernel-rules.rst in addition to this document.

If changes affect userland-kernel interfaces, please send the MAN-PAGES maintainer (as listed in the MAINTAINERS file) a man-pages patch, or at least a notification of the change, so that some information makes its way into the manual pages. User-space API changes should also be copied to linux-api@vger.kernel.org.

No MIME, no links, no compression, no attachments. Just plain text

Linus and other kernel developers need to be able to read and comment on the changes you are submitting. It is important for a kernel developer to be able to "quote" your changes, using standard e-mail tools, so that they may comment on specific portions of your code.

For this reason, all patches should be submitted by e-mail "inline". The easiest way to do this is with `git send-email`, which is strongly recommended. An interactive tutorial for `git send-email` is available at https://git-send-email.io.

If you choose not to use `git send-email`:

> **Warning**

> Be wary of your editor's word-wrap corrupting your patch, if you choose to cut-n-paste your patch.

Do not attach the patch as a MIME attachment, compressed or not. Many popular e-mail applications will not always transmit a MIME attachment as plain text, making it impossible to comment on your code. A MIME attachment also takes Linus a bit more time to process, decreasing the likelihood of your MIME-attached change being accepted.

Exception: If your mailer is mangling patches then someone may ask you to re-send them using MIME.

See Documentation/process/email-clients.rst for hints about configuring your e-mail client so that it sends your patches untouched.

Respond to review comments

Your patch will almost certainly get comments from reviewers on ways in which the patch can be improved, in the form of a reply to your email. You must respond to those comments; ignoring reviewers is a good way to get ignored in return. You can simply reply to their emails to answer their comments. Review comments or questions that do not lead to a code change should almost certainly bring about a comment or changelog entry so that the next reviewer better understands what is going on.

Be sure to tell the reviewers what changes you are making and to thank them for their time. Code review is a tiring and time-consuming process, and reviewers sometimes get grumpy. Even in that case, though, respond politely and address the problems they have pointed out.

See Documentation/process/email-clients.rst for recommendations on email clients and mailing list etiquette.

Don't get discouraged - or impatient

After you have submitted your change, be patient and wait. Reviewers are busy people and may not get to your patch right away.

Once upon a time, patches used to disappear into the void without comment, but the development process works more smoothly than that now. You should receive comments within a week or so; if that does not happen, make sure that you have sent your patches to the right place. Wait for a minimum of one week before resubmitting or pinging reviewers - possibly longer during busy times like merge windows.

It's also ok to resend the patch or the patch series after a couple of weeks with the word "RESEND" added to the subject line:

```
[PATCH Vx RESEND] sub/sys: Condensed patch summary
```

Don't add "RESEND" when you are submitting a modified version of your patch or patch series - "RESEND" only applies to resubmission of a patch or patch series which have not been modified in any way from the previous submission.

Include PATCH in the subject

Due to high e-mail traffic to Linus, and to linux-kernel, it is common convention to prefix your subject line with [PATCH]. This lets Linus and other kernel developers more easily distinguish patches from other e-mail discussions.

`git send-email` will do this for you automatically.

Sign your work - the Developer's Certificate of Origin

To improve tracking of who did what, especially with patches that can percolate to their final resting place in the kernel through several layers of maintainers, we've introduced a "sign-off" procedure on patches that are being emailed around.

The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below:

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

a. The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
b. The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
c. The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
d. I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

then you just add a line saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

using your real name (sorry, no pseudonyms or anonymous contributions.) This will be done for you automatically if you use `git commit -s`. Reverts should also include "Signed-off-by". `git revert -s` does that for you.

Some people also put extra tags at the end. They'll just be ignored for now, but you can do this to mark internal company procedures or just point out some special detail about the sign-off.

Any further SoBs (Signed-off-by:'s) following the author's SoB are from people handling and transporting the patch, but were not involved in its development. SoB chains should reflect the **real** route a patch took as it was propagated to the maintainers and ultimately to Linus, with the first SoB entry signalling primary authorship of a single author.

When to use Acked-by:, Cc:, and Co-developed-by:

The Signed-off-by: tag indicates that the signer was involved in the development of the patch, or that he/she was in the patch's delivery path.

If a person was not directly involved in the preparation or handling of a patch but wishes to signify and record their approval of it then they can ask to have an Acked-by: line added to the patch's changelog.

Acked-by: is often used by the maintainer of the affected code when that maintainer neither contributed to nor forwarded the patch.

Acked-by: is not as formal as Signed-off-by:. It is a record that the acker has at least reviewed the patch and has indicated acceptance. Hence patch mergers will sometimes manually convert an acker's "yep, looks good to me" into an Acked-by: (but note that it is usually better to ask for an explicit ack).

Acked-by: does not necessarily indicate acknowledgement of the entire patch. For example, if a patch affects multiple subsystems and has an Acked-by: from one subsystem maintainer then this usually indicates acknowledgement of just the part which affects that maintainer's code. Judgement should be used here. When in doubt people should refer to the original discussion in the mailing list archives.

If a person has had the opportunity to comment on a patch, but has not provided such comments, you may optionally add a `Cc:` tag to the patch. This is the only tag which might be added without an explicit action by the person it names - but it should indicate that this person was copied on the patch. This tag documents that potentially interested parties have been included in the discussion.

Co-developed-by: states that the patch was co-created by multiple developers; it is used to give attribution to co-authors (in addition to the author attributed by the From: tag) when several people work on a single patch. Since Co-developed-by: denotes authorship, every Co-developed-by: must be immediately followed by a Signed-off-by: of the associated co-author. Standard sign-off procedure applies, i.e. the ordering of Signed-off-by: tags should reflect the chronological history of the patch insofar as possible, regardless of whether the author is attributed via From: or Co-developed-by:. Notably, the last Signed-off-by: must always be that of the developer submitting the patch.

Note, the From: tag is optional when the From: author is also the person (and email) listed in the From: line of the email header.

Example of a patch submitted by the From: author:

```
<changelog>

Co-developed-by: First Co-Author <first@coauthor.example.org>
Signed-off-by: First Co-Author <first@coauthor.example.org>
Co-developed-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: Second Co-Author <second@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
```

Example of a patch submitted by a Co-developed-by: author:

```
From: From Author <from@author.example.org>

<changelog>

Co-developed-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: Random Co-Author <random@coauthor.example.org>
Signed-off-by: From Author <from@author.example.org>
Co-developed-by: Submitting Co-Author <sub@coauthor.example.org>
Signed-off-by: Submitting Co-Author <sub@coauthor.example.org>
```

Using Reported-by:, Tested-by:, Reviewed-by:, Suggested-by: and Fixes:

The Reported-by tag gives credit to people who find bugs and report them and it hopefully inspires them to help us again in the future. Please note that if the bug was reported in private, then ask for permission first before using the Reported-by tag. The tag is intended for bugs; please do not use it to credit feature requests.

A Tested-by: tag indicates that the patch has been successfully tested (in some environment) by the person named. This tag informs maintainers that some testing has been performed, provides a means to locate testers for future patches, and ensures credit for the testers.

Reviewed-by:, instead, indicates that the patch has been reviewed and found acceptable according to the Reviewer's Statement:

Reviewer's statement of oversight

By offering my Reviewed-by: tag, I state that:

a. I have carried out a technical review of this patch to evaluate its appropriateness and readiness for inclusion into the mainline kernel.
b. Any problems, concerns, or questions relating to the patch have been communicated back to the submitter. I am satisfied with the submitter's response to my comments.
c. While there may be things that could be improved with this submission, I believe that it is, at this time, (1) a worthwhile modification to the kernel, and (2) free of known issues which would argue against its inclusion.
d. While I have reviewed the patch and believe it to be sound, I do not (unless explicitly stated elsewhere) make any warranties or guarantees that it will achieve its stated purpose or function properly in any given situation.

A Reviewed-by tag is a statement of opinion that the patch is an appropriate modification of the kernel without any remaining serious technical issues. Any interested reviewer (who has done the work) can offer a Reviewed-by tag for a patch. This tag serves to give credit to reviewers and to inform maintainers of the degree of review which has been done on the patch. Reviewed-by: tags, when supplied by reviewers known to understand the subject area and to perform thorough reviews, will normally increase the likelihood of your patch getting into the kernel.

Both Tested-by and Reviewed-by tags, once received on mailing list from tester or reviewer, should be added by author to the applicable patches when sending next versions. However if the patch has changed substantially in following version, these tags might not be applicable anymore and thus should be removed. Usually removal of someone's Tested-by or Reviewed-by tags should be mentioned in the patch changelog (after the '---' separator).

A Suggested-by: tag indicates that the patch idea is suggested by the person named and ensures credit to the person for the idea. Please note that this tag should not be added without the reporter's permission, especially if the idea was not posted in a public forum. That said, if we diligently credit our idea reporters, they will, hopefully, be inspired to help us again in the future.

A Fixes: tag indicates that the patch fixes an issue in a previous commit. It is used to make it easy to determine where a bug originated, which can help review a bug fix. This tag also assists the stable kernel team in determining which stable kernel versions should receive your fix. This is the preferred method for indicating a bug fixed by the patch. See :ref:`describe_changes` for more details.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\process\[linux-master][Documentation][process]submitting-patches.rst, line 555); *backlink***
>
> Unknown interpreted text role "ref".

Note: Attaching a Fixes: tag does not subvert the stable kernel rules process nor the requirement to Cc: stable@vger.kernel.org on all stable patch candidates. For more information, please read Documentation/process/stable-kernel-rules.rst.

The canonical patch format

This section describes how the patch itself should be formatted. Note that, if you have your patches stored in a `git` repository, proper patch formatting can be had with `git format-patch`. The tools cannot create the necessary text, though, so read the instructions below anyway.

The canonical patch subject line is:

```
Subject: [PATCH 001/123] subsystem: summary phrase
```

The canonical patch message body contains the following:

- A `from` line specifying the patch author, followed by an empty line (only needed if the person sending the patch is not the author).
- The body of the explanation, line wrapped at 75 columns, which will be copied to the permanent changelog to describe this patch.
- An empty line.
- The `Signed-off-by:` lines, described above, which will also go in the changelog.
- A marker line containing simply `---`.
- Any additional comments not suitable for the changelog.
- The actual patch (`diff` output).

The Subject line format makes it very easy to sort the emails alphabetically by subject line - pretty much any email reader will support that - since because the sequence number is zero-padded, the numerical and alphabetic sort is the same.

The `subsystem` in the email's Subject should identify which area or subsystem of the kernel is being patched.

The `summary phrase` in the email's Subject should concisely describe the patch which that email contains. The `summary phrase` should not be a filename. Do not use the same `summary phrase` for every patch in a whole patch series (where a `patch series` is

an ordered sequence of multiple, related patches).

Bear in mind that the `summary phrase` of your email becomes a globally-unique identifier for that patch. It propagates all the way into the `git` changelog. The `summary phrase` may later be used in developer discussions which refer to the patch. People will want to google for the `summary phrase` to read discussion regarding that patch. It will also be the only thing that people may quickly see when, two or three months later, they are going through perhaps thousands of patches using tools such as `gitk` or `git log --oneline`.

For these reasons, the `summary` must be no more than 70-75 characters, and it must describe both what the patch changes, as well as why the patch might be necessary. It is challenging to be both succinct and descriptive, but that is what a well-written summary should do.

The `summary phrase` may be prefixed by tags enclosed in square brackets: "Subject: [PATCH <tag>...] <summary phrase>". The tags are not considered part of the summary phrase, but describe how the patch should be treated. Common tags might include a version descriptor if the multiple versions of the patch have been sent out in response to comments (i.e., "v1, v2, v3"), or "RFC" to indicate a request for comments.

If there are four patches in a patch series the individual patches may be numbered like this: 1/4, 2/4, 3/4, 4/4. This assures that developers understand the order in which the patches should be applied and that they have reviewed or applied all of the patches in the patch series.

Here are some good example Subjects:

```
Subject: [PATCH 2/5] ext2: improve scalability of bitmap searching
Subject: [PATCH v2 01/27] x86: fix eflags tracking
Subject: [PATCH v2] sub/sys: Condensed patch summary
Subject: [PATCH v2 M/N] sub/sys: Condensed patch summary
```

The `from` line must be the very first line in the message body, and has the form:

> From: Patch Author <author@example.com>

The `from` line specifies who will be credited as the author of the patch in the permanent changelog. If the `from` line is missing, then the `From:` line from the email header will be used to determine the patch author in the changelog.

The explanation body will be committed to the permanent source changelog, so should make sense to a competent reader who has long since forgotten the immediate details of the discussion that might have led to this patch. Including symptoms of the failure which the patch addresses (kernel log messages, oops messages, etc.) are especially useful for people who might be searching the commit logs looking for the applicable patch. The text should be written in such detail so that when read weeks, months or even years later, it can give the reader the needed details to grasp the reasoning for **why** the patch was created.

If a patch fixes a compile failure, it may not be necessary to include _all_ of the compile failures; just enough that it is likely that someone searching for the patch can find it. As in the `summary phrase`, it is important to be both succinct as well as descriptive.

The `---` marker line serves the essential purpose of marking for patch handling tools where the changelog message ends.

One good use for the additional comments after the `---` marker is for a `diffstat`, to show what files have changed, and the number of inserted and deleted lines per file. A `diffstat` is especially useful on bigger patches. If you are going to include a `diffstat` after the `---` marker, please use `diffstat` options `-p 1 -w 70` so that filenames are listed from the top of the kernel source tree and don't use too much horizontal space (easily fit in 80 columns, maybe with some indentation). (`git` generates appropriate diffstats by default.)

Other comments relevant only to the moment or the maintainer, not suitable for the permanent changelog, should also go here. A good example of such comments might be `patch changelogs` which describe what has changed between the v1 and v2 version of the patch.

Please put this information **after** the `---` line which separates the changelog from the rest of the patch. The version information is not part of the changelog which gets committed to the git tree. It is additional information for the reviewers. If it's placed above the commit tags, it needs manual interaction to remove it. If it is below the separator line, it gets automatically stripped off when applying the patch:

```
<commit message>
...
Signed-off-by: Author <author@mail>
---
V2 -> V3: Removed redundant helper function
V1 -> V2: Cleaned up coding style and addressed review comments

path/to/file | 5+++--
...
```

See more details on the proper patch format in the following references.

Backtraces in commit mesages

Backtraces help document the call chain leading to a problem. However, not all backtraces are helpful. For example, early boot call

chains are unique and obvious. Copying the full dmesg output verbatim, however, adds distracting information like timestamps, module lists, register and stack dumps.

Therefore, the most useful backtraces should distill the relevant information from the dump, which makes it easier to focus on the real issue. Here is an example of a well-trimmed backtrace:

```
unchecked MSR access error: WRMSR to 0xd51 (tried to write 0x0000000000000064)
at rIP: 0xffffffffae059994 (native_write_msr+0x4/0x20)
Call Trace:
mba_wrmsr
update_domains
rdtgroup_mkdir
```

Explicit In-Reply-To headers

It can be helpful to manually add In-Reply-To: headers to a patch (e.g., when using `git send-email`) to associate the patch with previous relevant discussion, e.g. to link a bug fix to the email with the bug report. However, for a multi-patch series, it is generally best to avoid using In-Reply-To: to link to older versions of the series. This way multiple versions of the patch don't become an unmanageable forest of references in email clients. If a link is helpful, you can use the https://lore.kernel.org/ redirector (e.g., in the cover email text) to link to an earlier version of the patch series.

Providing base tree information

When other developers receive your patches and start the review process, it is often useful for them to know where in the tree history they should place your work. This is particularly useful for automated CI processes that attempt to run a series of tests in order to establish the quality of your submission before the maintainer starts the review.

If you are using `git format-patch` to generate your patches, you can automatically include the base tree information in your submission by using the `--base` flag. The easiest and most convenient way to use this option is with topical branches:

```
$ git checkout -t -b my-topical-branch master
Branch 'my-topical-branch' set up to track local branch 'master'.
Switched to a new branch 'my-topical-branch'

[perform your edits and commits]

$ git format-patch --base=auto --cover-letter -o outgoing/ master
outgoing/0000-cover-letter.patch
outgoing/0001-First-Commit.patch
outgoing/...
```

When you open `outgoing/0000-cover-letter.patch` for editing, you will notice that it will have the `base-commit:` trailer at the very bottom, which provides the reviewer and the CI tools enough information to properly perform `git am` without worrying about conflicts:

```
$ git checkout -b patch-review [base-commit-id]
Switched to a new branch 'patch-review'
$ git am patches.mbox
Applying: First Commit
Applying: ...
```

Please see `man git-format-patch` for more information about this option.

> **Note**
>
> The `--base` feature was introduced in git version 2.9.0.

If you are not using git to format your patches, you can still include the same `base-commit` trailer to indicate the commit hash of the tree on which your work is based. You should add it either in the cover letter or in the first patch of the series and it should be placed either below the `---` line or at the very bottom of all other content, right before your email signature.

References

Andrew Morton, "The perfect patch" (tpp).

<https://www.ozlabs.org/~akpm/stuff/tpp.txt>

Jeff Garzik, "Linux kernel patch submission format".

<https://web.archive.org/web/20180829112450/http://linux.yyz.us/patch-format.html>

Greg Kroah-Hartman, "How to piss off a kernel subsystem maintainer".

<http://www.kroah.com/log/linux/maintainer.html>

<http://www.kroah.com/log/linux/maintainer-02.html>

<http://www.kroah.com/log/linux/maintainer-03.html>

<http://www.kroah.com/log/linux/maintainer-04.html>

<http://www.kroah.com/log/linux/maintainer-05.html>

<http://www.kroah.com/log/linux/maintainer-06.html>

NO!!!! No more huge patch bombs to linux-kernel@vger.kernel.org people!

<https://lore.kernel.org/r/20050711.125305.08322243.davem@davemloft.net>

Kernel Documentation/process/coding-style.rst

Linus Torvalds's mail on the canonical patch format:

<https://lore.kernel.org/r/Pine.LNX.4.58.0504071023190.28951@ppc970.osdl.org>

Andi Kleen, "On submitting kernel patches"

Some strategies to get difficult or controversial changes in.

http://halobates.de/on-submitting-patches.pdf