```
.. _submittingpatches:

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 :ref:`Documentation/process <development_process_main>`.
Also, read :ref:`Documentation/process/submit-checklist.rst <submitchecklist>`
for a list of items to check before
submitting code.  If you are submitting a driver, also read
:ref:`Documentation/process/submitting-drivers.rst <submittingdrivers>`;
for device tree binding patches, read
Documentation/devicetree/bindings/submitting-patches.txt.

Many of these steps describe the default behavior of the ``git`` version
control system; if you use ``git`` to prepare your patches, you'll find much
of the mechanical work done for you, though you'll still need to prepare
and document a sensible set of patches.  In general, use of ``git`` will make
your life as a kernel developer easier.

0) 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.

It is still possible to download kernel releases via tarballs (as described
in the next section), but that is the hard way to do kernel development.

1) ``diff -up``
---------------

If you must generate your patches by hand, use ``diff -up`` or ``diff -uprN``
to create patches.  Git generates patches in this form by default; if
you're using ``git``, you can skip this section entirely.

All changes to the Linux kernel occur in the form of patches, as
generated by :manpage:`diff(1)`.  When creating your patch, make sure to
create it in "unified diff" format, as supplied by the ``-u`` argument
to :manpage:`diff(1)`.
Also, please use the ``-p`` argument which shows which C function each
change is in - that makes the resultant ``diff`` a lot easier to read.
Patches should be based in the root kernel source directory,
not in any lower subdirectory.

To create a patch for a single file, it is often sufficient to do::

        SRCTREE= linux
        MYFILE=  drivers/net/mydriver.c


        cd $SRCTREE
        cp $MYFILE $MYFILE.orig
        vi $MYFILE      # make your change
        cd ..
        diff -up $SRCTREE/$MYFILE{.orig,} > /tmp/patch
```

To create a patch for multiple files, you should unpack a "vanilla",
or unmodified kernel source tree, and generate a ``diff`` against your
own source tree. For example::

        MYSRC= /devel/linux

        tar xvfz linux-3.19.tar.gz
        mv linux-3.19 linux-3.19-vanilla
        diff -uprN -X linux-3.19-vanilla/Documentation/dontdiff \
                linux-3.19-vanilla $MYSRC > /tmp/patch

``dontdiff`` is a list of files which are generated by the kernel during
the build process, and should be ignored in any :manpage:`diff(1)`-generated
patch.

Make sure your patch does not include any extra files which do not
belong in a patch submission. Make sure to review your patch -after-
generating it with :manpage:`diff(1)`, to ensure accuracy.

If your changes produce a lot of deltas, you need to split them into
individual patches which modify things in logical stages; see
:ref:`split_changes`. This will facilitate review by other kernel developers,
very important if you want your patch accepted.

If you're using ``git``, ``git rebase -i`` can help you with this process. If
you're not using ``git``, ``quilt`` <http://savannah.nongnu.org/projects/quilt>
is another popular alternative.

.. _describe_changes:

2) 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`.

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`.

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 the patch fixes a logged bug entry, refer to that bug entry by
number and URL. If the patch follows from a mailing list discussion,
give a URL to the mailing list archive; use the https://lkml.kernel.org/
redirector with a ``Message-Id``, to ensure that the links cannot become
stale.

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 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 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. For example::

        Fixes: e21d2170f366 ("video: remove unnecessary platform_set_drvdata()")

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\")

.. _split_changes:

3) 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.

4) Style-check your changes

Check your patch for basic style violations, details of which can be
found in
:ref:`Documentation/process/coding-style.rst <codingstyle>`.
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.

5) 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 functions as a list of
last resort, 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.

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
:ref:`Documentation/process/stable-kernel-rules.rst <stable_kernel_rules>`
in addition to this file.

Note, however, that some subsystem maintainers want to come to their own
conclusions on which patches should go to the stable trees. The networking
maintainer, in particular, would rather not see individual developers
adding lines like the above to their patches.

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.

For small patches you may want to CC the Trivial Patch Monkey
trivial@kernel.org which collects "trivial" patches. Have a look
into the MAINTAINERS file for its current manager.

Trivial patches must qualify for one of the following rules:

- Spelling fixes in documentation
- Spelling fixes for errors which could break :manpage:`grep(1)`
- Warning fixes (cluttering with useless warnings is bad)
- Compilation fixes (only if they are actually correct)
- Runtime fixes (only if they actually fix things)
- Removing use of deprecated functions/macros
- Contact detail and documentation fixes
- Non-portable code replaced by portable code (even in arch-specific,
  since people copy, as long as it's trivial)
- Any fix by the author/maintainer of the file (ie. patch monkey
  in re-transmission mode)

6) 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".

.. 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

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 :ref:`Documentation/process/email-clients.rst <email_clients>`
for hints about configuring your e-mail client so that it sends your patches
untouched.

7) E-mail size

Large changes are not appropriate for mailing lists, and some
maintainers. If your patch, uncompressed, exceeds 300 kB in size,
it is preferred that you store your patch on an Internet-accessible
server, and provide instead a URL (link) pointing to your patch. But note
that if your patch exceeds 300 kB, it almost certainly needs to be broken up
anyway.

8) Respond to review comments

Your patch will almost certainly get comments from reviewers on ways in
which the patch can be improved. You must respond to those comments;
ignoring reviewers is a good way to get ignored in return. 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.

9) 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.

10) 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.

11) Sign your work 鈥 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.)

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.

If you are a subsystem or branch maintainer, sometimes you need to slightly
modify patches you receive in order to merge them, because the code is not
exactly the same in your tree and the submitters'. If you stick strictly to
rule (c), you should ask the submitter to rediff, but this is a totally
counter-productive waste of time and energy. Rule (b) allows you to adjust
the code, but then it is very impolite to change one submitter's code and
make him endorse your bugs. To solve this problem, it is recommended that
you add a line between the last Signed-off-by header and yours, indicating
the nature of your changes. While there is nothing mandatory about this, it
seems like prepending the description with your mail and/or name, all
enclosed in square brackets, is noticeable enough to make it obvious that
you are responsible for last-minute changes. Example::

        Signed-off-by: Random J Developer <random@developer.example.org>
        [lucky@maintainer.example.org: struct foo moved from foo.c to foo.h]
        Signed-off-by: Lucky K Maintainer <lucky@maintainer.example.org>

This practice is particularly helpful if you maintain a stable branch and
want at the same time to credit the author, track changes, merge the fix,
and protect the submitter from complaints. Note that under no circumstances
can you change the author's identity (the From header), as it is the one
which appears in the changelog.

Special note to back-porters: It seems to be a common and useful practice
to insert an indication of the origin of a patch at the top of the commit
message (just after the subject line) to facilitate tracking. For instance,
here's what we see in a 3.x-stable release::

  Date: Tue Oct 7 07:26:38 2014 -0400

    libata: Un-break ATA blacklist

```
    libata: Un-break ATA blacklist

    commit 1c40279960bcd7d52dbdf1d466b20d24b99176c8 upstream.

And here's what might appear in an older kernel once a patch is backported::

    Date:   Tue May 13 22:12:27 2008 +0200

        wireless, airo: waitbusy() won't delay

        [backport of 2.6 commit b7acbdfbd1f277c1eb23f344f899cfa4cd0bf36a]

Whatever the format, this information provides a valuable help to people
tracking your trees, and to people trying to troubleshoot bugs in your
tree.


12) When to use Acked-by: and Cc:
---------------------------------

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.


13) 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.

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.

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.

14) 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 (only needed if the person
  sending the patch is not the author).

- An empty line.

- The body of the explanation, line wrapped at 75 columns, which will
  be copied to the permanent changelog to describe this patch.

- 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.

A couple of example Subjects::

    Subject: [PATCH 2/5] ext2: improve scalability of bitmap searching
    Subject: [PATCH v2 01/27] x86: fix eflags tracking

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

        From: Original 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.) is
especially useful for people who might be searching the commit logs
looking for the applicable patch.  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. 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.

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.)

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

.. _explicit_in_reply_to:

15) 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://lkml.kernel.org/ redirector (e.g., in
the cover email text) to link to an earlier version of the patch series.

16) Sending ``git pull`` requests

If you have a series of patches, it may be most convenient to have the
maintainer pull them directly into the subsystem repository with a
``git pull`` operation. Note, however, that pulling patches from a developer
requires a higher degree of trust than taking patches from a mailing list.
As a result, many subsystem maintainers are reluctant to take pull
requests, especially from new, unknown developers. If in doubt you can use
the pull request as the cover letter for a normal posting of the patch
series, giving the maintainer the option of using either.

A pull request should have [GIT] or [PULL] in the subject line. The
request itself should include the repository name and the branch of
interest on a single line; it should look something like::

  Please pull from

      git://jdelvare.pck.nerim.net/jdelvare-2.6 i2c-for-linus

  to get these changes:

A pull request should also include an overall message saying what will be
included in the request, a ``git shortlog`` listing of the patches
themselves, and a ``diffstat`` showing the overall effect of the patch series.
The easiest way to get all this information together is, of course, to let
``git`` do it for you with the ``git request-pull`` command.

Some maintainers (including Linus) want to see pull requests from signed
commits; that increases their confidence that the request actually came
from you. Linus, in particular, will not pull from public hosting sites
like GitHub in the absence of a signed tag.

The first step toward creating such tags is to make a GNUPG key and get it
signed by one or more core kernel developers. This step can be hard for
new developers, but there is no way around it. Attending conferences can
be a good way to find developers who can sign your key.

Once you have prepared a patch series in ``git`` that you wish to have somebody
pull, create a signed tag with ``git tag -s``. This will create a new tag
identifying the last commit in the series and containing a signature
created with your private key. You will also have the opportunity to add a
changelog-style message to the tag; this is an ideal place to describe the
effects of the pull request as a whole.

If the tree the maintainer will be pulling from is not the repository you
are working from, don't forget to push the signed tag explicitly to the
public tree.

When generating your pull request, use the signed tag as the target. A
command like this will do the trick::

  git request-pull master git://my.public.tree/linux.git my-signed-tag

References

Andrew Morton, "The perfect patch" (tpp).
  <http://www.ozlabs.org/~akpm/stuff/tpp.txt>

Jeff Garzik, "Linux kernel patch submission format".
  <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://lkml.org/lkml/2005/7/11/336>

Kernel Documentation/process/coding-style.rst:
  :ref:`Documentation/process/coding-style.rst <codingstyle>`

Linus Torvalds's mail on the canonical patch format:
  <http://lkml.org/lkml/2005/4/7/183>

Andi Kleen, "On submitting kernel patches"
  Some strategies to get difficult or controversial changes in.

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