Testing Requirements for drm/i915 Features and Patches

November 12, 2013 • danvet

I want to make automated test coverage an integral part of our feature and bugfix development process. For features this means that starting with the design phase testability needs to be considered an integral part of any feature. This needs to go through the entire development process, from planning, development, patch submission and final validation. For bugfixes that means the fix is only complete once the automated testcase for it is also done, if we need a new one

This specifically excludes testing with humans somewhere in the loop. We are extremely limited in our validation resources, every time we put something new onto the “manual testing” plate something else *will* fall off.

I’ve let this float for quite a while both internally in Intel and on the public mailing lists. Thanks to everyone who provided valuable input. Essentially this just codifies the already existing expectations from me as the maintainer, but those requirements haven’t really been clear and a lot of emotional discussions ensued. With this we should now have solid guidelines and can go back to coding instead of blowing through so much time and energy on waging flamewars.

Why?

There are a bunch of reasons why having good tests is great:

- **More predictability.** Right now test coverage often only comes up as a topic when I drop my maintainer review onto a patch series. Which is too late, since it’ll delay the otherwise working patches and so massively frustrates people. I hope by making test requirements clear and up-front we can make the

upstreaming process more predictable. Also, if we have good tests from the get-go there should be much less need for me to drop patches from my trees after having them merged

- **Less bikeshedding.** In my opinion test cases are an excellent means to settle bikesheds - we've had in the past seen cases of endless back&forths where writing a simple testcase would have shown that *all* proposed color flavours are actually broken.

The even more important thing is that fully automated tests allow us to legitimately postpone cleanups. If the only testing we have is manual testing then we have only one shot at a feature tested, namely when the developer tests it. So it better be perfect. But with automated tests we can postpone cleanups with too high risks of regressions until a really clear need is established. And since that need often never materializes we'll save work.

- **Better review.** For me it's often helps a lot to review tests than the actual code in-depth. This is especially true for reviewing userspace interface additions.

- **Actionable regression reports.** Only if we have a fully automated testcase do we have a good chance that QA reports a regression within just a few days. Everything else can easily take weeks (for platforms and features which are explicitly tested) to months (for stuff only users from the community notice). And especially now that much more shipping products depend upon a working i915.ko driver we just can't do this any more.

- **Better tests.** A lot of our code is really hard to test in an automated fashion, and pushing the frontier of what is testable often requires a lot of work. I hope that by making tests an integral part of any feature work and so forcing more people to work on them and think about testing we'll advance the state of the art at a brisker pace.

Risks and Buts

But like with every change, not everything is all glorious and fun:

- **Bikeshedding on tests.** This plan is obviously not too useful if we just replace massive bikeshedding on patches with massive bikeshedding on

testcases. But right now we do almost no review on i-g-t patches so the risk is small. Long-term the review requirements for testcases will certainly increase, but as with everything else we simply need to strive for a good balance to strike for just the right amount of review.

Also if we really start discussing tests *before* having written massive patch series we'll do the bikeshedding while there's no real rebase pain. So even if the bikeshedding just shifts we'll benefit I think, especially for really big features.

- **Technical debt in test coverage.** We have a lot of old code which still completely lacks testcases. Which means that even small feature work might be on the hook for a big pile of debt restructuring. I think this is inevitable occasionally. But I think that doing an assessment of the current state of test coverage of the existing code *before* starting a feature instead of when the patches are ready for merging should help a lot, before everyone is invested into patches already and mounting rebase pain looms large.

Again we need to strive for a good balance between "too many tests to write up-front for old code" and "missing tests that only the final review and validation uncovers creating process bubbles".

- **Upstreaming of product stuff.** Product guys are notoriously busy and writing tests is actual work. On the other hand the upstream codebase feeds back into *all* product trees (and the upstream kernel), so requirements are simply a bit higher. And I also don't think that we can push the testing of some features fully to product teams, since they'll be pissed really quickly if every update they get from us breaks their stuff. So if these additional test requirements (compared to the past) means that some product patches won't get merged, then I think that's the right choice.

- **But …** all the other kernel drivers don't do this. We're also one of the biggest driver's in the kernel, with a code churn rate roughly 5x worse than anything else and a pretty big (and growing) team. Also, we're often the critical path in enabling new platforms in the fast-paced mobile space. Different standards apply.

Test Coverage Expectations

Since the point here is to make the actual test requirements known up-front we need to settle on clear expectations.

- Tests must fully cover userspace interfaces. By this I mean exercising all the possible options, especially the usual tricky corner cases (e.g. off-by-one array sizes, overflows). It also needs to include tests for all the userspace input validation (i.e. correctly rejecting invalid input, including checks for the error codes). For userspace interface additions technical debt really must be addressed. This means that when adding a new flag and we currently don't have any tests for those flags, then I'll ask for a testcase which fully exercises all the flag values we currently supported on top of the new interface addition.

- Tests need to provide a reasonable baseline coverage of the internal driver state. The idea here isn't to aim for full coverage, that's an impossible and pointless endeavor. The goal is to have a good starting point of tests so that when a tricky corner case pops up in review or validation that it's not a terribly big effort to add a specific testcase for it. This is very much a balance thing to get right and we need a bit of experience to get a good handle here.

- Issues discovered in review and final validation need automated test coverage. This includes any bugs found after a feature has already landed and is even more important for regressions. The reasoning is that anything which slipped the developer's attention is tricky enough to warrant an explicit testcase, since in a later refactoring there's a good chance that it'll be missed again. This has a bit a risk to delay patches, but if the basic test coverage is good enough as per the previous point it really shouln't be an issue.

- Finally we need to push the testable frontier with new ideas like pipe CRCs, modeset state cross checking or arbitrary monitor configuration injection (with fixed EDIDs and connector state forcing). The point here is to foster new crazy ideas, and the expectation is very much *not* that developers then need to write testcases for all the old bugfixes that suddenly became testable. That workload needs to be spread out over a bunch of features touching the relevant area. This only really applies to features and code paths which are currently in the "not testable" bucket anyway.

This should specify the "what" decently enough, but we also need to look at how tests should work.

Specific testcases in i-g-t are obviously the preferred form, but for some features that's just not possible. In such cases in-kernel self-checks like the modeset state checker of fifo underrun reporting are really good approaches. Two caveats apply:

- The test infrastructure really should be orthogonal to the code being tested. In-line asserts that check for preconditions are really nice and useful, but since they're closely tied to the code itself they have a good chance to be broken in the same ways.

- The debug feature needs to be enabled by default, and it needs to be loud. Otherwise no one will notice that something is amiss. So currently the fifo underrun reporting doesn't really count since it only causes debug level output when something goes wrong. Of course it's still a really good tool for developers, just not yet for catching regressions.

Finally the short lists of excuses that don't count as proper test coverage for a feature:

- Manual testing. We are ridiculously limited on our QA manpower. Every time we drop something onto the "manual testing" plate something else *will* drop off. Which means in the end that we don't really have any test coverage. So if patches don't come with automated tests, in-kernel cross-checking or some other form of validation attached they need to have really good reasons for doing so.

- Testing by product teams. The entire point of Intel OTC's "upstream first" strategy is to have a common codebase for everyone. If we break product trees every time we feed an update into them because we can't properly regression test a given feature then the value of upstreaming features is greatly diminished. In my opinion this could potentially doom collaborations with product teams. We just can't have that.

This means that when products teams submit patches upstream they also need to submit the relevant testcases as patches to i-g-t.

Process Adjustements

The important piece is really to not start with thinking about tests only when everything else is done.

- For big features we should have an upfront discussion about the test coverage and what all should be done (like any coverage gaps for existing code and features to fill, a new crazy test infrastructure idea to implement as a proof of concept or what kinds of tests would provide a reasonable base coverage). For really big features writing a quick test plan and everyone signing off on it could be useful. Especially to be able to learn and improve once everything has landed and the usefulness of the tests is much clearer.

- Tests should be implemented together with the feature or bugfix and should be ready about the same time. Having both pieces at hand should help development, testing and review.

- If we decide that new test infrastructure is required or that there's a large gap in the coverage of existing code then that should be done before the main feature is developed. Otherwise we'll suffer again the pains of rebase hell for no gain.

- Finally developers are *not* expected to run the full testsuite before submitting patches. The test suite currently simply takes too long to run and we don't have any good centralized infrastructure to speed things up by running tests on multiple machines in parallel. And then proper testing requires a wide array of different platforms anyway, so full regression testing is still squarely a job for our QA. Of course we need to improve our infrastructure and also make it easier to run a useful subset of tests while developing patches.

With that, happy testcase writing to everyone!

Share this on → Twitter Google+
Tags: Maintainer-Stuff