Developing Out-of-Tree Drivers alongside In-Kernel Drivers

Jesse Brandeburg
LAN Access Division, Intel Corporation
`jesse.brandeburg@intel.com`

Abstract

Getting your driver released into the kernel with a GPL license is promoted as the holy grail of Linux hardware enabling, and I agree. That said, producing a quality GPL driver for use in the entire Linux ecosystem is not a task for the faint of heart. Releasing an Ethernet driver through kernel.org is one delivery method, but many users still want a driver that will support the newest hardware on older kernels.

To meet our users' requirements for more than just hardware support in the latest kernel.org kernel, we in Intel's LAN Access Division (LAD) developed a set of coping strategies, processes, code, tools, and testing methods that are worth sharing. These learnings help us reuse code, maintain quality, and maximize our testing resources in order to get the best quality product in the shortest amount of time to the most customers. While not the most popular topic with core kernel developers, out-of-tree drivers are a necessary business solution for hardware vendors with many users. Our Open Source drivers generally work with all kernel releases 2.4 and later, and I'll explain many of the details about how we get there.

1 Introduction

This paper's goal is to lay out a roadmap for others to use in order to streamline the out-of-tree development process. Since many developers are able to live solely in the kernel, a secondary goal is to expose some of the business realities that our product group has to cope with, and the solutions we have developed.

Our business goal is simple: Sell hardware. This hard reality guides many of our decisions. To do this, we enable drivers for as many users and operating systems as possible. In an ideal world we would have unlimited resources, and a fully staffed development and testing team with plenty of idle time, but instead we have to make do with busy developers, constrained testing resources, and of course business and customer needs. As such, we've developed tricks and common practice within our code and development process in order to maximize the number of Operating Systems supported.

Intel® Wired Ethernet developers actively maintain multiple (eight) drivers in the kernel, and strive to be good open-source contributors and supporters while still creating an out-of-tree driver, i.e. we are not the enemy.

2 Reasons You Might Need an Out-of-tree Driver

Business Need

Our software support opens new business opportunities for network hardware sales by leveraging the (awesome) environment of the Open Source community and vendors. We sample silicon and boards months before we ship, and need something to deliver to customers to allow them to test.

We avoid a lot of thrash and introduction of last minute OS support requirements from hardware vendors by having a tested and ready "out-of-tree" driver that OEM system integrators and vendors can use to ship our hardware. We also make our out-of-tree drivers available via `e1000.sourceforge.net` in the e1000 project, and on intel.com.

Provide More Complete Hardware Support

Customers are happier if our hardware works in every kernel they might use right out of the box.

No Pre-announcing Hardware

We are unable to ship driver support to the kernel for hardware that either hasn't shipped or won't ship "real soon now." We try not to give up any competitive advantage we might achieve by not informing our competitors of our plans. The out-of-tree driver allows for a testable and feature rich launch of hardware with supporting drivers, even if the driver hasn't made it all the way through net-next into the upstream kernel. It also means we have something to ship on the software CD "in the box."

Users on Old Kernels

Some customers are using 2.4 kernels (still) in production. This creates a bit of a headache for drivers like ixgbe that have many new features that depend on newer kernels. For operating systems and kernels this old we have a policy of "just make it work" which is generally all that is required. Some customers upgrade to the latest and greatest system and network hardware but will not or cannot upgrade their OS for their own business reasons. For example, we have met enterprise customers who have an application that **will not run** on any OS newer than RedHat Enterprise Linux 3. However, RedHat is still supporting RHEL3, at least for now. In our case the right thing to do here is follow the lead of the OS Vendor and try to provide basic support. In some cases, only new hardware is available to replace existing failing hardware, forcing users to upgrade hardware while keeping their existing infrastructure and certifications in place.

Silicon Validation

One of the tasks we have as driver developers is to validate that the silicon we ship will work with our driver code as well as validating new silicon features. This often needs a lot of "non-production" code to be developed and we do that development in our out-of-tree driver, usually on a branch.

Dirty Laundry

The out-of-tree driver source contains many comments and even some code that is never published, that allow us to reference internal bug tracking databases, investigation notes, and debug code (possibly for silicon validation.) There is no value in shipping this code to the open source community and the process of building source allows us to maintain higher quality code, while still having the functionality in the code that we need.

3 Implementation

Shared Code

Our definition of "shared code" is code that is usable under multiple operating systems, with a non-encumbering license. In our environment we have factored out the code that supports functions/features common to all driver hardware tasks like initialization, reset, link management, etc.

The shared code makes up almost 10,000 lines of code (out of 24,961) in our current ixgbe driver. It is shared across multiple OS drivers, including Linux, FreeBSD, Windows drivers (all versions), Windows Testing tools (control panel), Manufacturing/Test tools, as well as customers.

In order to do such a thing without GPL violations, we **maintain exclusive copyright** to the shared code files, allowing us to release the code with any license we (as the exclusive copyright holder) need. Another option would be to dual license the code, but this has some legal implications that we weren't interested in dealing with, and that are beyond this paper's scope.

When we design this code for a family of silicon, we use function pointers to cover the initialization and setup sections that might have differing implementations for each release of silicon.

This code has #defines that are typically negatively defined to allow drivers that don't want to compile in support for a given piece of hardware to strip all the unused code from their driver build. The code typically looks like:

```
#ifndef NO_NNN_HARDWARE_SUPPORT

/* some code specific to NNN */

#endif
```

We also use #defines to mark pre-release sections of code for new hardware or feature support, allowing us to control the driver/features and hardware implemented for a particular driver build. Often for cleanliness of the code those #defines are removed after hardware or the feature first ships.

Some of the other advantages to sharing pieces of the driver initialization code are: More consumers of the code means more developers available to work bugs; more testing coverage because the shared code gets used repeatedly. This maximizes limited testing and development resources to achieve the most productivity.

Build the Code

Our drivers' code base is actually built via a Makefile. The Makefile takes several passes over the code. The major innovation is using unifdef.c from the kernel to clean out #defines and code that we don't want included. This is done via a list of #defines in the Makefile that declare which code we want to keep or strip. Using this method we implement our new hardware support, allowing us the flexibility to add/remove new hardware to a particular driver build right up to the ship date. After the hardware and software support ships for a given release, we typically leave in #defines for hardware support that we might want to discard to reduce code size (as above), but remove #defines that we might have used for new hardware support in the base driver portion of the code. The assumption here is that once the driver supports a given piece of hardware it always will. Another advantage of this build process is that the driver source can be branched and stabilized with a particular set of hardware and features supported, without the code forking from the mainline development.

Create New Drivers

One of the lessons we've learned is that a driver should not have endless hardware revisions added to it. It creates too much regression testing load, and new hardware support too often breaks existing functionality. While it is immensely seductive to reuse all the code in a driver, experience has shown us that driver code is typically brittle. Our conclusion is that whenever possible create a new driver for "the next generation" of silicon. This of course creates more work and more code to maintain. This is an issue that we are continuing to struggle with, but we believe is the correct way forward.

Coding Style

We allow ONLY Linux kernel style for the "shared code" files. In the kernel, there is a Documentation/CodingStyle file and we have implemented an internal process that requires the shared code (and our drivers' core code) to conform to the that document. This causes some discussion among the differing software camps, but saves many headaches in the long run. This is especially useful when keeping code in the out-of-tree and in-kernel drivers the same.

Internal Maintainers

We have implemented mailing lists and automated check-in notification emails that encourage and ease peer review of code. In particular for our shared code we have a single committer that is the only user allowed to commit changes. This guarantees code goes through a minimum level of review by the maintainer before commit to ensure process is followed and that code meets requirements. This has prevented many hours of pain and suffering of developers having to fix bugs or quality issues other users introduce to the shared code.

Consistent External Maintainer Interface

The internal development of the out-of-tree driver is typically followed by changes for the in-kernel driver, which are all pushed through our primary maintainer. Over the past several years we've developed and refined a relationship with the maintainers of the networking stack and networking drivers. Having a single person that is our contact with the maintainers guarantees consistent communication, process, and dramatically increases our chances of getting patches accepted. Jeff Kirsher's paper in other proceedings of the 2010 Linux Symposium explains this in greater detail.

Patch All the Time

We've consistently been asked by the upstream maintainers to not "patch bomb" the lists every 6-12 weeks. We've also found that during development the best way to do kernel (upstream) patches is to immediately introduce any change made to our out-of-tree driver to our internal kernel patch process. We have eased this process by mimicking the kernel development process internally. We use internal mailing lists, an internal patchwork server, and internal git servers. A developer who has just created a patch for the out-of-tree code is in the perfect position (just the right knowledge) to create the kernel patch for the same change. The developer creates the patch, typically uses stgit to email it to the list, and then the patch is tracked in patchwork through testing and then eventual submittal via email to the networking maintainer.

Kernel Compatibility Layer

We follow a similar model that is used by libata to provide backport compatibility to some distribution kernels. Our Ethernet driver kcompat.h and kcompat.c files allow for "upstream" looking core driver code which works on older kernels (yes it's GPL, so you can use it too.) When combined with strategically placed #defines in our driver core code, our drivers can compile and load on almost all 2.4 and 2.6 kernel versions. Of course #defines for certain OS capabilities are unavoidable and end up breaking up mainline source with #ifdefs, but using flags in the driver to advertise driver/hardware features and capabilities can minimize the "#ifdef thrash."

4 Version Control

Our current infrastructure uses CVS but we could easily switch to any other version control system that has a sufficient ecosystem to allow easy cross platform (aka Windows) development. The large features that we rely on version control to provide are branching, tagging, and change tracking. We have taken great pains to enforce adherence to committing only a single change at a time.

Nightly Labels - auto-builds

We have recently started the nightly process of automatic labeling and building of certain components of our software. The shared code is built in both a DOS and UNIX linefeed version. The drivers then consume that "built" version in our build tool when the "checkout" is prepared before a driver source build. Finally the source is built on a Linux machine via make, and compile tested on several different distributions.

Reproducible Build Process

One of the benchmarks for our process is reproducible builds. We take steps to make sure that any given build can be rebuilt in the future should something go wrong. We periodically make practice runs to prove it is working.

5 Pitfalls

Kernel standards can conflict with internal requirements

One of the issues we ran into is that the kernel community requested e1000e use C99 initializers for function pointers. We made that change, but it required our in-kernel driver to fork from the internal shared code because C99 syntax doesn't work with DOS compilers.

GPL concerns

We must maintain exclusive copyright in order to multi-license the shared code. We can't take in code changes to our shared code, when submitted against GPL source, unless we transfer copyright or rewrite code and counter-propose to maintain authorship/copyright. We don't expect everyone to understand our licensing concerns, but we do try to offer changes and alternatives to patches on the list that allow us to maintain our copyright and still not allow too much drift between our out-of-tree shared code and the kernel version.

Distributions and Backports

Backports typically come from upstream changes only, which means that the distribution engineers are often re-inventing the wheel we've already created in our standalone driver.

In all fairness Novell has a great KMP (kernel module package) model that allows us to provide them a driver from our out-of-tree code that they build and provide to users.

Bug fixes often go into distributions but sometimes don't make it upstream. Even when they do make it upstream, it is difficult to track what is required to be changed in the out-of-tree driver.

What to Test?

Pre-production

Pre-production testing tests mostly the out-of-tree driver, looking for hardware bugs and verifying driver functionality. If we had unlimited time and resources we would ideally make the driver backport for the distribution, and then test. Often distribution code submittal windows are closed before we have final silicon, and yet the silicon will ship before the distribution in question.

Early Release

Early (before release of the hardware) kernel submittal often has just basic functionality. Testing this gives you a warm fuzzy feeling and can be the basis of the initial kernel submittal, and so is a worthwhile effort.

Distributions

Testing our driver that is included in the distribution is the hardest because they likely don't even have hardware support yet for the device you're testing. The distribution is assumed to have basically the same driver that is upstream, but in practice because of the backporting the distribution has to do, the driver is an actual fork. The confusion due to differing version numbers in drivers must be managed in some way or another. In our case we add -kN to our version numbers for drivers submitted to the kernel. In addition we also ask the distributions if they make any changes to our driver to update the -kN to the next odd value, with the goal of having in-kernel drivers have all even numbers for N, i.e. 2,4,6, and distribution backports would hopefully have odd values of N, i.e. 1,3,5.

Limitations to This Path

Double work

Most driver changes must be made once to the internal version and once to the kernel version. After that, changes to the distribution drivers need to be initiated and tracked through the relevant methods.

Upstream vs Out-of-tree

Keeping the drivers in sync is a significant effort. Our solution is diligence, patience, and a significant time and resource commitment. The whole team participates in the open source community via monitoring mailing lists and submitting patches.

6 Common Questions

Why not release code earlier?

We have: 82599 driver released 4 weeks before general hardware availability. 82580 driver released November 7 2009, at least 8 weeks before general hardware availability.

Why not develop in the open?

By this, I believe the question to be "why don't we have a public git tree?" We push patches upstream as soon as they are ready. Developing in a public git repository only allows us to target one kernel version, and our requirements include other delivery vehicles besides the upstream kernel.

How come you don't just develop for the upstream kernel?

Our customers demand support in older kernels, not just the upstream kernel driver.

Why don't you use a "real" version control system?

We'd like to use GIT, but training Windows, BSD developers, technical marketing engineers, and software configuration management engineers to use GIT is a big effort. In short, we're working on it but don't expect quick changes.

7 Conclusion

Developing in-kernel and out-of-tree drivers can be done with a common source base and a minimum of work. This paper shows some of the methods and practices Intel Wired Ethernet developers utilize. We welcome any follow-up questions and discussion either directly to the author or on our public email list.

`e1000-devel@lists.sourceforge.net`

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada