Overcoming Obstacles to Mainlining

By Tim Bird
Device mainlining project lead
CE Workgroup of the Linux Foundation

Abstract

Many companies struggle with contributing to open source projects. This paper identifies key difficulties that many large companies face in making contributions of their modifications to the mainline (kernel.org) version of the Linux kernel. It provides recommendations for overcoming these obstacles. Some of the difficulties discussed are: version gap, expertise problems, development cycle mismatch, and management prioritization. This paper does not address low-level immediate issues with mainlining[1], such as the correct coding style for patches or email etiquette when submitting changes. Instead it takes a more high-level look at structural problems inside companies and the industry that inhibit meaningful engagement by corporate developers within the open source community.

The goal of this paper is to help individual developers and companies identify and implement practices that will accelerate their participation in open source, so that they can enjoy more of the value of open source besides just the open code base.

This paper is divided into roughly four parts, starting with a section identifying obstacles, followed by a more detailed examination and description of those obstacles. Following that, there are suggestions for overcoming those obstacles. Finally, the paper will end with a description of the benefits of engaging in open source, and links to additional information and resources related to this topic.

Introduction

The novel *Anna Karenina*, by renowned Russian author Leo Tolstoy, begins with a dramatic sentence: "Happy families are all alike; every unhappy family is unhappy in its own way" This statement expresses an important principle[2] about complex systems (especially complex *social* systems), namely that there are lots of ways to fail, but only a narrow set of activities which will lead to success.

The same is true of companies that successfully contribute to open source. There are myriad ways for a company or group to "get it wrong", but it requires a sequence of correct actions to succeed.

The very first step on the path to success is deciding to make contributing to open source a priority. Yogi Bera, a famous American baseball player and layman philosopher once quipped: "If people don't want to come out to the ballpark, nobody's going to stop them." The same is true of engagement with open source. There is nothing stopping a company from merely taking open source software and using it in their products. However, some of the key benefits of open source are missed by doing so.

1 This document uses the term "mainlining" as a verb. Mainlining refers to contributing to the original open source project for a particular component. For the kernel this is the kernel.org version of the kernel. See the glossary for these and other definitions used in this document.

2 See http://en.wikipedia.org/wiki/Anna_Karenina_principle

Towards the end of this paper, some of these benefits will be described – and hopefully provide some motivation for companies to put more effort into engaging with the open source community. But to start, let's describe some of the obstacles to be overcome...

Identifying Obstacles

Many developers within the open source community have a sense of the difficulties faced by new contributors. Numerous talks have been given that are laced with advice about when to begin development, how to format patches, how to interact with maintainers, etc. However, some of this advice is based on hunches and guesswork (although, to be fair, often well-informed guesswork) about the actual problems that contributors have. In an effort to obtain more objective data from potential contributors themselves, the CE Workgroup of the Linux Foundation conducted a survey. The survey was conducted in September of 2014, and had over 600 respondents. It was hoped that through directly questioning actual Linux kernel developers, some quantifiable data could be collected on the difficulties these developers perceived in contributing to Linux.

The survey[3] consisted of 39 questions. One thing that distinguished this survey from other similar surveys is that it targeted qualified developers who did *not* contribute to mainline. A "qualified developer" was defined as someone who actually made a change to the Linux source that was shipped in a commercial product. It was presumed that this level of proficiency indicates someone who is potentially capable of contributing to Linux mainline. An effort was made to make this survey available to developers who do not participate on community lists or in community events. The survey asked these developers about their own reasons for not contributing.

A total of 653 responses were received. Of those, 278 responded "yes" to questions asking if they had made a change to the Linux kernel, and had that change shipped in a product. Those respondents formed the pool of qualified developers. Some questions in the survey were used to determine if the developer had contributed changes to the community, and about their frequency and success rate in doing so.

However, the majority of the questions in the survey presented potential obstacles, and allowed the respondent to choose from 5 choices (strongly agree, agree, neutral, disagree, and strongly disagree) to indicate whether that obstacle applied to them.

Survey Results

The top perceived obstacles, based on those agreeing or strongly agreeing in the survey, are shown in Table 1. The right-most column in the table shows the mode (most selected answer) and the percentage of each choice selected, for each of the obstacles listed.

3 You can find the survey questions here: https://docs.google.com/forms/d/1brPQlGfEm7wev36m7BD9hJQ6QF-lFuNpipSiBjKhP_8/viewform

Obstacle	Those Agreeing	Strength
Older kernel version	54%	mode=agree (13%,41%,28%,10%,8%)
Depends on other code not upstream	50%	mode=agree (15%,35%,23%,15%,12%)
It’s too hard	45%	mode=neutral (9%,36%,44%,9%,2%)
Could not test	41%	mode=neutral (15%,26%,33%,19%,7%)
Patch not good enough	35%	mode=neutral (3%,32%,40%,20%,5%)
Employer does not provide time	33%	mode=neutral (11%,22%,31%,27%,9%)
Afraid of rejection	33%	mode=neutral (15%,18%,25%,25%,17%)

Table 1: Top perceived obstacles, according to qualified developers

Additional insights

These different obstacles will be described in more detail shortly. However, a few other questions from the survey yielded additional insights. Two questions investigated the self-motivation of the developer to contribute, to see if that was a factor. 90% of developers reported that they thought it was important to submit changes upstream, and 85% indicated that they would like to personally submit changes upstream.

Other questions investigated the developer's perception of their management's attitude toward contributing. Surprisingly, only 18% of developers reported that their management did not approve of them contributing to upstream, while 33% said that their employer does not provide enough time to do it.

The survey also revealed some other issues which were not perceived to be obstacles to mainlining. Only 7% of developers thought their English skills were not good enough to work with the community, and only 2% thought that someone else should be responsible for contributing their patches.

Obstacles Detail

Next, some of the obstacles identified in the survey are described in detail.

Version gap

The number one obstacle that developers agreed with in the survey was working on a kernel version

that was too old. Working with an older kernel is something referred to as “version gap”. This is the gap between the source code that corporate developers are working on for a product release, and the current version of mainline software.

Version gap comes about due to delays that occur as software is handled and developed by different parties and makes its way through multiple processes on the way to shipment in a product. Most consumer electronics companies use a Linux kernel in their products based on one they receive from a CPU vendor, rather than working directly with a mainline kernel (the one from kernel.org). In the mobile phone space, the amount of code that is added to the Linux kernel can be quite large, ranging from 1 to 3 million lines of code. For a product based on Android, the code goes through multiple parties before it finally becomes available to consumers. For example, a phone recently shipped by Sony Mobile Communications included the 3.4 version of the Linux kernel. At the time the phone shipped, the current mainline version of the kernel was 4.0. This represents a difference of 16 kernel versions, covering a time period of almost 3 years. Looking back at where this code came from, Google first obtained a copy of the kernel from kernel.org and added changes to support features specific to Android on top of that. Then Qualcomm added additional changes to support aspects of their processor family[4]. Finally, Sony[5] made their own set of patches to the kernel to support specific hardware and for bugs found in testing.

This table shows the total number of commits (individual change sets) and authors for a single branch of Sony's internal development tree.

Commiter e-mail	Commits	Authors
Google/Android commits	963	61
Other	2677	828
Qualcomm	20395	635
Sony	1799	203
Between Sony's tree and mainline base (3.4)	25843	1757

Table 2: Number of commits in Sony mobile phone Linux source tree

The commits are divided into groups based on the domain name of the email of the author who made the commit. There were 61 developers with e-mail addresses from google.com or android.com and together they made a total of 963 change sets to the Linux kernel. The startling number in Table 2 is for commits made by Qualcomm developers, at over 20,000 change sets for this version of the kernel. In fairness to Qualcomm it should be noted that, in terms of number of lines of code, most SoC vendors make a similar number of changes (within an order of magnitude, by count of lines changed) to the Linux kernel, for shipped mobile phone products. The problem for Sony is that by the time it receives the kernel, the source already includes more than 23,000 change sets. Sony itself adds another approximately 1800 changes, authored by more than 200 kernel developers.

4 Actually, it's a bit more complicated than that. For some SoC vendors, Google and the vendors work together to modify the mainline code. But by the time a product company receives the kernel source, it has been modified by multiple parties.

5 Unless otherwise stated, “Sony” is used to refer to Sony Mobile Communications.

Almost every embedded product will have some amount of patches on top of a vanilla mainline kernel. It is not uncommon for desktop distributions of Linux (such as Ubuntu or Fedora) to have thousands of patches relative to a mainline kernel in their official releases. But things in the mobile space are particularly bad. It takes time to apply and test this much code, even when it is being carried over from release to release, and not being originally developed. The amount of time for development, application of patches, and testing by all these different contributors to the kernel results in the 2 to 3-year time delay, and a version gap of approximately 20 Linux kernel versions customarily seen in mobile phone products on the market today.

The problem with version gap, and the reason it constitutes a major obstacle to mainlining, is that changes made to an older kernel are difficult to apply to a newer version of the kernel. The Linux kernel itself is under constant development. Features and sub-systems are constantly being enhanced and refactored. This often makes it difficult to take code that was written for an older kernel and apply it to a newer kernel. For example, a driver for USB phy hardware written for 3.4 would have to be re-written to apply it to the 4.0 kernel (the version of the kernel at the time this document was written).

One requirement of mainlining is that the code submitted to upstream should apply cleanly to the current code base. But this is difficult in the face of changes being made in the upstream software.

It should be noted that although the Linux kernel has a stable user-space ABI, internally (that is for kernel modules or source code) it has neither a stable ABI nor a stable API. This ultimately is the root cause of version gap – that internal interfaces are subject to change with no backwards compatibility. While this creates problems for long-term maintenance of out-of-tree code, it is viewed as an essential element of the kernel's adaptability and longevity[6].

Patches to non-mainlined code

Another obstacle indicated by survey respondents is dependency on non-mainlined code. When code has passed through multiple parties, and had significant additional code added, there is a greater likelihood that changes made by a company will be against code that is itself not in the upstream kernel to begin with. When a company has patches to non-upstream code, it is usually more difficult to contribute this code, if it's possible at all.

Let me illustrate with an example. Some Sony phones ship with Synaptics touchscreen hardware, which did not have a driver upstream at the time of product creation. Sony integrated this driver from sources directly from the hardware vendor. But additional patches were made to this software, to fix bugs and modify it for Sony's specific phone configuration. If this driver were upstream Sony would have a clear path for where to send the changes. But working with an individual company rather than the open source community presents a different set of challenges. Sony approached Synaptics with patches, but for a variety of reasons, Synaptics did not integrate the changes in their code base. This means that Sony had to carry these patches in its own source repository, and maintain them over time by itself.

This situation is by no means unique. It is common for hardware vendors not to integrate patches for

6 See Documentation/stable_api_nonsense.txt inside the kernel source tree.

their drivers from their customers, when the drivers are out-of-tree. The situation gets even more complicated for an IP block that is integrated into an SOC. In this case, the driver for the IP block may have originally been written by the IP block creator, and then modified by the SOC vendor. In this case, the authorship of the software for the IP block has become muddled, and the final customer of the chip may not have a direct relationship with the software authors for the IP block.

In the case of patches against non-upstream source, it can be difficult to find the right party to submit the patch to, and they may not accept the patches. While it may be possible for the company that makes the final product to try to contribute the base driver to mainline in place of the hardware vendor (for example, Sony mainlining the driver on behalf of Synaptics), there are often difficulties with this. The product company doesn't know as much about the hardware or IP block as the original creator, and they don't have experience with multiple users of the hardware or IP block, or know its future development roadmap, so they can't generalize the software as effectively. Also, the cost of mainlining the driver is often very much more than the cost of maintaining patches against it, so the business motivation for contributing the entire driver by the final product company is low.

Difficulty of mainlining itself

Another obstacle indicated in the survey was the difficulty of the actual act of mainlining itself. That is, survey respondents perceived that it is a difficult process to submit a change to upstream. These difficulties can come from internal processes in a company, or from the process of interacting with upstream developers.

Internal obstacles to mainlining

Some companies have no clear processes to follow in order to get permission to submit a change. Many companies have a culture of protecting the intellectual property of the company and an inherent aversion to releasing information or code that might put strategic assets of the company at risk. The default condition in most companies is to avoid releasing code, and it takes a strong will to overcome this tendency.

Part of the culture of restraint in releasing code is due overestimating the value of code that companies have previously written. One cause of this is that companies sometimes fail to separate the differentiating features in a product from those which are non-differentiating. Non-differentiating features are those that provide to the customer no value that distinguishes one company's products from another. This categorization is very important. It is crucial to recognize that differentiating code that ships in a product is an asset, but that non-differentiating code is a cost. And, due to the changing perceptions of customers and the ever-expanding base of open source code, the non-differentiating features in any particular product category increase over time. Some code that was costly to develop and valuable in the past can become a burdensome expense over time which provides no differentiating product value.

Upstream difficulties

For difficulties related to interacting with upstream developers, the process actually *is* pretty cumbersome for those not familiar with it. The basic steps for submitting a patch and checking that it meets acceptance criteria for upstream are described in files in the kernel documentation directory:

SubmittingPatches,SubmitChecklist, and CodingStyle. These documents are valuable, but are by no means comprehensive. There are guidelines for when to submit patches, who to submit them to, and unwritten social rules for interacting on mailing lists and when to contact maintainers if you have not heard from them yet. In other words, there is a lot to learn. The problem is not that any particular rule is difficult, but that there are so many rules to follow. And it's possible that getting a single one wrong can result in delays in getting your code accepted upstream.

For developers who are familiar with the process, it is easy to lose awareness of the many steps involved. Kernel maintainers are very often extremely busy, and they don't have time to correct mistakes themselves, so there tends to be a high threshold for quality. A single trivial mistake might result in a patch being rejected, or worse, just ignored.

Speaking from experience, this is less of a problem than it used to be. There are now tools that help a developer correct or avoid mistakes before sending a patch. Tools such as checkpatch.pl and get_maintainer.pl are very helpful to newcomers in correcting patches and finding the right place to send them. But there is still definitely a lot to get right when submitting a patch, and it can be quite daunting for a new contributor.

While contributing to open source has a large set of rules and processes, the same is true for internal development inside companies. Corporate developers may find themselves navigating between two sets of rules and processes – the ones used by their open source project, and the ones used internally. This imposes an extra burden on developers who try to do both consistently on a day-to-day basis. It takes some time to switch between the mindset of internal development and mainline development, and this is made more difficult if there is a big difference between the two sets of tools and processes.

One example of this from my own experience was that for a particular project I worked on, the kernel source was kept in perforce - a popular commercial source code management tool. However, the preferred source code management tool for working with the kernel within the open source community is git. I found that switching between perforce and git introduced extra overhead in my day-to-day interactions.

In general, switching between two development worlds results in overall poor performance, similar to how high-latency scheduling results in poor performance in a real-time system. Not doing full-time contributing means that proficiency in open source methods is developed more slowly. It takes more time for the practices to become integrated into a developer's mindset and workflow, resulting in less overall efficiency.

Another thing that adds to the difficulty, is that in some areas of the kernel, the required expertise for contributing is very high, and is increasing. The Linux kernel is very mature (the project is now over 20 years old). The Linux kernel is used in many different markets and products, from high-speed stock trading, to digital watches. In some key areas, the skill required to make a change to the kernel, and not upset the requirements of these other areas, is very high. For example, the Linux process scheduler has been carefully developed and tuned over many years to make it support a lot of different workloads well. Therefore the skill and experience required to make an acceptable change to this area of the kernel is very, very high.

But there are other areas of the kernel where this high threshold for contribution does not exist. About

half of the code going into current releases of the kernel is in the form of drivers for specific hardware. There is a lower bar for entry for this type of code.

Finally, one common error that contributes to the difficult of actually submitting a patch is the practice of developing code in isolation, and trying to submit a large implementation at the end of a development cycle. This is an almost certain way to increase the difficulty of having a patch accepted. It is very common to see large patches need substantial re-work to fit within the greater design and architecture of the kernel. The code in the Linux kernel has to support a large number of use cases and consumers, and is very generalized. Almost all code that is integrated into the kernel goes through multiple revisions before being accepted.

The more time that code is worked on without feedback or input from others whose requirements the code must meet (or at least not disturb), the greater the chance that the code will fail to meet those requirements and thus the greater the chance of rejection or need for big changes. There is a saying in the open source community that is very widespread: “release early and often”. This means that code and ideas should be presented to other developers early in the development cycle. This gives a chance for bad designs to be corrected early, when the cost of correcting them is low.

In the case of code developed in isolation, the difficulty of getting it accepted is due to a faulty original development strategy.

As an aside, this situation can be one of the most discouraging things for a new contributor. A developer who believes they have a good patch, that they have invested a lot of time in, can be very disappointed to find out that their whole approach is unacceptable and they have to go back to the drawing board to get their feature into mainline.

Patch not good enough

Another obstacle to mainlining can be that the patch itself is not suitable for contributing. This can be for two reasons: 1) the code may not be of sufficient quality for the mainline kernel, or 2) the code may be too specialized. Let's face it. In the rush to ship a product, sometimes compromises are made. Code that may work in a particular setting or configuration may not work in the general case. And sometimes a quick hack is created that solves an immediate problem, but is ugly and unmaintainable in the long run. This can be especially true in the embedded market[7]. When the set of software that will run on a system is fixed and known, it may make sense to not develop fully generalized solutions. But these things have a way of coming around to bite you later.

Sometimes software developers do not believe that their code will be re-used in future products. In this case, a specialized solution may be perfectly acceptable and the most cost-effective way to ship the product. But experience indicates that sometimes code that is considered “throw-away” or “good enough” for one release, ends up being shipped in subsequent releases or in other products, where the flaws show up in unfortunate ways.

So, at the end of the development cycle there may be a set of patches against the kernel that are not of the highest quality, or may not be generalized for a wide set of use cases. Specialized code may in some circumstance not be inherently bad, but it may not be mainlinable. As stated previously, the

7 The author does not consider current mobile smartphones to be embedded products.

Linux kernel is used in an incredibly large number of markets and supports many, many different use cases. It is a huge code base with an extraordinary number of volunteer contributors. Upstream maintainers are particularly sensitive to ensuring that the code is maintainable in the long run, and that it supports the widest possible set of uses.

Often, the creator of a piece of code has an instinctive sense that major rewriting of a patch will be needed to make the code acceptable to mainline. And quite often, this sense is correct.

Finally, sometimes the reason that a specialized patch might not be acceptable into mainline may actually be a flaw with mainline itself. In some areas, the kernel is still maturing and being developed. There may not be a framework or sub-system in the kernel that supports a particular driver or feature in a generic way. This was true, for example, of NFC support in the 3.4 version of the kernel. This Linux kernel did not have a framework for NFC drivers that was sufficient for the needs of many products. In this case, it was not possible to mainline the NFC drivers that were being used in phones at the time.

Not enough time

The final obstacle to mainlining covered here is the obstacle of “not enough time”. Even though many developers would like to contribute to mainline, they find that they do not have enough time to do so. It is true that sometimes the managers of a company simply provide no time for their developers to contribute to open source. This is a problem that needs to be addressed, and some survey respondents agreed that their management could be better at providing time for this activity. But the situation is a bit more complicated than just a deficiency in the amount of time provided by the employer.

The process of contributing something to mainline is open-ended. Depending on the circumstances and the individual patch it can be difficult to predict how long (calendar-wise) it will take to get a patch accepted. The process involves sending a patch, waiting for responses, making fixes based on those responses, and repeating this process until all interested parties are satisfied. Because open source developers are outside of your company, they have no commitment to your company to respond on any particular time schedule. The actual personal time required by the submitter of the patch might be small (a few hours or days), but the overall process of getting a patch accepted upstream might stretch out over days, weeks or even months.

It is also important that a contributor respond quickly to feedback. This ensures that other developers can review the patch modifications while the patch is still fresh in their minds. Waiting a few days or weeks to respond will result in a complete restart of the submission process. This may seem a bit unfair, that maintainers and other upstream developers don't have to respond quickly but submitters do, but this is an artifact of the many-to-few relationship between contributors and maintainers and the open nature of the community.

However, in the face of this process, software developers working on products have deadlines to meet. I like to refer to this as the “product treadmill”. Developers who have tight deadlines cannot wait for responses from the open source community to finish their code. And they might be too busy themselves to respond quickly to upstream feedback. Therefore, there is an inherent conflict between their time availability and the needs of the upstream acceptance process.

For managers who are used to being able to control or influence the schedule of development tasks, this aspect of mainlining can be frustrating.

Overcoming Obstacles

Any one of the previously mentioned issues can be a major impediment to contributing to open source, but when taken together the situation looks downright bleak. But there are companies that are successful at contributing, and lessons have been learned over the years to help companies overcome these obstacles.

Here are some recommendations for practices that can help overcome these obstacles.

For certain market segments it is likely that some degree of version gap will persist for the foreseeable future. However, there are steps you can take to overcome your current version gap, and work towards the use of less out-of-tree code.

The first recommendation is to pay attention to version gap during your sourcing. If you have a choice between two processors, or two pieces of hardware that do the same thing, and one has better mainline support, choose the one with better support. This reduces the amount of patches you will maintain yourself which will be based on out-of-tree code. While this sounds nice in theory, it is common for other factors such as price or features to weigh more heavily in your purchasing decisions. You will need to educate the people in your organization who source your hardware, to be aware of this aspect of the hardware (that is, whether there is mainline support for it). They need to understand that this has a big impacton the long-term software cost for each piece of hardware.

In the case that the hardware you choose is not supported in mainlin yet, try to work with your vendor to mainline as much functionality as possible over time. While you and your vendors work on this, try hard to get at least a minimal core of the mainline kernel running on your hardware platform. You may never be able to run an unmodified mainline kernel as the shipping kernel on your product, but you can get portions of your hardware supported in mainline. You might do this on a publicly available development board that has hardware components the same as your product, or on your actual product hardware itself. The ideas is that as you expand the base of support for your product hardware in mainline, other interested parties including your own development teams and your customers can increasingly help with the mainlining work.

This is really important. Even if you can only get enough support in the mainline kernel to run a shell on your hardware, that is a start. If you can't do this now it should be your first goal. This is a critical base on which to build additional support and port additional drivers to the latest Linux kernel version. This mainline base won't be usable for your product releases (yet), but over time it will get closer and closer to being product grade, and closer to mainline, reducing your version gap. Note that this likely means in the short term that you will have a group of developers working on a kernel that you did *not* receive from your SOC vendor (the mainline kernel), and it will result in your having to manage two kernels for your product hardware, one for mainlining work and one for product work.

You should have a small team dedicated to working on mainlining, composed of developers who are not on the product treadmill. IBM, Intel, Samsung and Texas Instruments are all companies that are successfully contributing to open source. These companies have dedicated “Open Source Technology”

teams, consisting of developers who are not directly responsible for product releases. Such teams need not be large, compared to the overall investment your company has in Linux developers. But they should be dedicated 100% to open source operations. This allows them to avoid the switching cost of multiple "worlds" of development practices and tools. It also allows them to learn and use their open source skills more quickly and effectively. It helps to seed this group with individual developers who already have experience with open source contributions. If you can get an existing kernel maintainer to join your team, that's an excellent way to start to build your skill set. Finally, this team should use the same tools and processes for its internal work as the open source community. For example it should use git to manage the kernel source. It should use mailing lists and patches for code reviews. Patches sent internally should follow all the same guidelines for formatting and style as used by the open source community. The reviews should follow community style of feedback and follow-up. All these practices will help reinforce the methods and skills needed for effective work with the external community.

In terms of overcoming the difficulty in learning these skills, you should use a combination of training and mentoring. The Linux Foundation has technical events where many presentations have been given (and continue to be given) about good practices for mainlining. Many of these presentations are available in slide and video format on the embedded Linux wiki[8], or from the Linux Foundation events web pages. The CE Workgroup "Device Mainlining" project is in the process of collecting resources you can use for training your developers in open source practices. You should also have your more-experienced developers train and mentor your less-experienced developers. Samsung has a formal open source mentoring program where product team developers are trained over a 6-month period and given objectives to mainline real code to upstream projects. It has proven very successful.

To overcome the internal difficulties of submitting code, your company should have a streamlined process for approving patches for submitting to mainline.

One example of streamlining is that if your company has already published the code for a product (for example due to your obligations under the GPL), then your developers should have a green light to contribute any of the changes you made to that kernel to mainline. The rationale for this is that if your company has already published the code, there's no additional IP or asset exposure for this software in mainlining it. That is, if there were IP issues to consider, those should have been analyzed before the code was added to the Linux kernel in the first place. This single policy (of granting automatic permission to mainline already-published code) can dramatically reduce the authorization bottleneck in your company, if one exists.

Also, your company should periodically review the status of code within your products, to identify differentiating vs. non-differentiating code. It should be your goal to use open source software for all non-differentiating code, to reduce the ongoing cost of maintaining this category of software in your product. This may mean that you contribute code to open source projects which cost you a lot to develop. But there's no sense trying to wring differentiating value out of software once it has moved into the non-differentiating category.

With regard to management not providing enough time, there are a couple of recommendations. First, it is important to illustrate the benefits of mainlining to the management of your company, to persuade them to allocate resources for this effort. This white paper, and other documents and data that have

8 See http://elinux.org/Events

been and are planned to be published by the Linux Foundation, should be useful for this purpose[9]. Realistically though, your company will always be more concerned with shipping product than it is in engaging with open source. Having a small team that works on open source allows management to better quantify the cost (and benefits) of this engagement.

With regard to low-quality code or specialized code, work with your product teams to identify and segregate such code. Have your developers who are experienced with mainlining review code that goes into the product trees, and try to catch errors which would make eventual mainlining more difficult. The code developed for a product may not be of sufficient quality or generality to be submitted to mainline immediately, but if you are careful you can dramatically reduce the cost of mainlining the code later. One practice used at Sony was to keep mainline-able and non-mainline-able code separate. (They were kept in different quilt series.) Having product developers recognize patches that have upstream applicability helps keep those patches in better condition for eventual mainlining.

In terms of interaction with the community, your developers who are focused on open source should get deeply involved with their relevant communities. This means they should actively participate in mailing lists, to the degree that the maintainers in their focus areas come to know their names. It is often helpful to attend (and present at) Linux conferences and events to meet face-to-face with other developers. Building a social connection with other community members, and particularly maintainers, is extremely beneficial to build better understanding, and to have more of your developers' work accepted by other upstream developers. The bottom line is that you should consciously work on the social element of your community engagement.

Coming back to sourcing, it would be great if you communicated to your supplier your desire for mainline support for their hardware. One company put language in their sourcing contracts that made it the default that an open source driver was delivered with the hardware. If a supplier could or would not fulfill that requirement, they had to ask for that clause to be removed during their contract negotiations. This extra step caused everyone to be aware of the desire for open source drivers. If the entire industry went a step further, and asked not just for open source drivers but for driver support in mainline, it would send a strong message to hardware and IP block vendors about the desirability of mainline support.

The proxy problem

While following these recommendations will have a positive effect, some of them create new problems of their own that need to be overcome. In particular, when one group of developers tries to engage with the open source community on behalf of others, and in particular when they try to submit patches that they themselves have not authored, it can lead to a difficult situation.

The open source-facing developers may be experienced with community methods, but they might not be subject matter experts. I refer to this as the proxy problem. To submit patches to the community, you should really have a good idea of the technology you are submitting. You should have the ability to respond quickly to community feedback, change the code, test the code, and re-submit it for additional community review. When a “proxy” developer works with the community, they might need assistance from internal teams that actually wrote the code. Although the open source team has the designated

9 See the “Incentives” section of this paper for one list of benefits

resource allocation to do the mainlining work, some time and resources may need to be allocated from product team developers, so they can help if needed.

Incentives

This all sounds like a lot of work and expense, to get a few patches mainlined. This raises the question: Why do this? What is the benefit of making this effort?

One benefit is illustrated by referring back to Table 2. Note that in the table, for a recently shipped phone Sony had approximately 1800 patches authored by more than 200 Sony kernel developers used for that product. This was for one branch of the kernel for one product. Over the last 3 years, Sony has had over 1100 developers make a patch to the Linux kernel, and we've managed over 800 different development branches for our products. We find ourselves applying the same changes over and over again, and carrying patches from one branch to another over time. Every patch that was upstream would have reduced that overall maintenance effort. Developers who spent time managing these patches could been better utilized to make new features for our customers.

But more important than reducing the maintenance cost, is the time cost to move those patches and re-integrate them across software releases from our SOC vendor, and over newer releases of the kernel. The mobile phone industry, like many consumer electronics markets, is an aggressive business. Product success can hinge on the ability to ship a product within a particular market window. And it is often critical to not be behind your competitors in terms of release dates. The delays introduced by the extra overhead of patch cherry-picking, re-integration and re-testing is a drag on the release schedules of products.

Therefore, reducing time-to-market for new products is extremely valuable. And reducing the cost of patch maintenance and overhead is a key factor in this reduction. This even is more important from a business perspective than the cost of maintaining thousands of patches over many years.

Other Benefits

But aside from money and time-to-market considerations, there are several other benefits to engaging with the community to mainline your patches. These benefits are less tangible than money or time-to-market, but are no less real.

One often overlooked benefit of submitting code to mainline is that it results in higher quality code. This is due to two factors: first, developers are more likely to improve the quality of code that will be seen in public. Developers do not wish to have bad code associated with themselves, as it damages their reputation and, quite frankly, hurts their ego. If they know they will submit code upstream, it provides extra incentives to get the code right. Second, developers can get feedback on the code during the open source review process. Even if the code is not accepted upstream, it's very likely that just going through the process of submitting it will result in improvements in the code, in areas such as style, maintainability, robustness and performance. This is free code review that should be taken advantage of.

Another benefit is that submitting code upstream communicates your requirements to others in the open

source community. This is valuable because even if the code is not accepted, it influences others in the community to see your needs. Sometimes, this is all it takes to move towards getting your requirements met. Your submission may influence an upstream maintainer to support your use case, or it may help other community members to submit code which supports your hardware. It is important to realize that the end goal of submitting code upstream is not to have your specific patches adopted, it is to have your product requirements met. Andrew Morton, a very experienced kernel developer, and one of it's current development leads, has told the story that some of his first patches were ones which he knew were not good enough to be accepted upstream. But someone else would come along and correct his mistakes, or fix up the patch for him, and his feature would ultimately get implemented. Of course, no one encourages the submission of intentionally malformed patches, but this story demonstrates that patches needs not be perfect to add value to upstream project. (It should be noted that as Andrew became proficient over time, his contributions became more and more acceptable, until he became one of the top reviewers of code going into the Linux kernel.)

There are even more benefits if your code is accepted into the kernel. If it is included in upstream, your code will get additional testing, both as part of automated suites that are run against the mainline kernel, and as other individuals and groups adopt the code and start using it. Other people will test and use the code in ways that you intended, and probably in ways you did not. Other kernel developers will fix things that are broken, and modify it when kernel interfaces change. This will improve the quality of the code,, often times without additional labor on your part. This is one of the biggest benefits of the open source model, that you miss out on by not engaging with mainline.

Another reason to submit changes upstream, which is more strategic than tactical, is that by mainlining something, you can gain a competitive advantage for your implementation. You might have an implementation for something which the Linux kernel does not currently support. If you get your code accepted upstream first, it reduces your cost of maintaining that code, but also increases the cost for others to get duplicate or overlapping code into the kernel. This makes your job easier, while simultaneously increasing the maintenance cost and development burden for any of your competitors who use a different implementation.

Note that the dynamics of this are very similar to the dynamics of industry standards, where the implementation which is adopted as a standard gains an advantage over other implementations due to network effects. The alternative, if you don't get your implementation upstream, is that you will end up either maintaining your implementation out of tree, or ultimately having to adopt the (now upstream) implementation of your competitor. Either of these can be more costly in the long run than getting your code upstream.

An important benefit of reducing version gap is the ability to utilize code more easily from current open source releases into your product. If the version of the kernel used on your hardware is closer to the mainline version, you can more easily integrate software from mainline, including the latest security fixes, into your product releases and your product updates. This can be very important and valuable to your customers.

Another reason for your company to contribute code upstream has to do with motivating your own employees. As seen in the survey (and from substantial experience), developers *want to* contribute to open source. In the survey, over 90% of developers said they personally wanted to contribute to open source. The reasons for this are varied but they include things like: receiving recognition from their

peers, wanting to contribute to something which benefits society, and wanting to have their code used by lots of people.

This is illustrated by own experience with a small patch. Many years ago I contributed a change to the kernel called “printk times”. This is just a small feature that adds a timestamp to each message printed by the kernel. But I was very happy as a developer to see it accepted, and I continue to feel a bit gratified when I see it continue to be used today. The developers at your company will similarly feel satisfied and happy if they can have their code accepted into open source projects. And they will be grateful to their company for giving them the time and resources to do it. Also, it is pleasing to employees to be part of an organization that contributes back to open source and to society. The effect of this on the morale of your employees should not be underestimated.

Another indirect benefit of participation in open source has to do with employee improvement. When the developers from your company interact with members of the open source community – as they read mailing lists, and engage in discussions – they will be interacting with some of the smartest people in the world. This exposure helps educate them on high quality design and architecture, and can help them improve their own skills at developing software.

Contributing to open source can be seen as a reward to your own employees, both for their personal technical improvement, and as a boost to their morale.

Finally, the open source community is as much a social construct as it is a technical one. As you engage in open source, and are seen a contributor, other members of the community will help you with your problems. It's human nature that as you help others, they will reciprocate and help you. And in general the quality of the developers in open source communities is very good, so the help you receive can be very useful. Contributing to open source helps generate good will within the community, which can be very valuable to your product development efforts.

You should note that the previously mentioned reasons for mainlining your code are all justifiable in the framework of helping your company better succeed in business. Reduction in costs, improvement in time-to-market, higher code quality, employee education and satisfaction, and community goodwill are all valuable to your business. And all of these are completely aside from any legal requirements of open source licenses and from any ethical and social arguments about contributing back to a community that provides you with software that is worth literally billions of dollars[10]. It turns out that while these other (ethical and social) arguments are worth consideration, it is possible to completely justify the cost of engaging in open source on a purely economic basis.

Conclusion

There are a number of difficult obstacles that companies face when trying to mainline their changes to the Linux kernel. A survey conducted by the CE Workgroup of the Linux Foundation was useful to determine some of the top obstacles perceived by current non-contributors. These included things like: working with an older kernel, having patches based on out-of-tree code, having low-quality or specialized patches, and not enough time provided by management. Some of these obstacles come

10 See “Estimating the Total Development Cost of a Linux Distribution” at http://www.linuxfoundation.org/sites/main/files/publications/estimatinglinux.html

about from bad practices, but some are a natural result of the current state of the industry and current product development processes.

By utilizing a small team of dedicated open source developers, and taking key steps to train them, give them time to perform the activity, and by streamlining their approval process, they can make significant progress in submitting code upstream. Keeping them off the product treadmill, and getting them deeply involved with the Linux community will help their progress.

A key factor is establishing a minimal base of mainline support on your product hardware, from which to mainline additional drivers and features. Sourcing hardware with already-upstream support, or at least less difference from mainline, is helpful to begin your development from a better starting point.

There are numerous benefits to engaging with the open source community, including reduced long-term maintenance cost, improved time to market, and many others. Many companies have learned to successfully contribute to open source. It is hoped that by following the recommendations in this paper that your company become a full-fledged, contributing member of the open source community and also recognize these benefits.

Credits and Resources

Information and ideas for this paper were gleaned from a number of resources, including talks at previous Linux events. Ideas from the following community leaders were especially helpful: Andrew Morton, Deepak Saxena, James Bottomley, Kevin Hilman, and Jon Corbet.

Original research in the form of the “obstacles” survey, and analysis of source trees was performed in the context of the Device Mainlining project of the CE Workgroup of the Linux Foundation. The purpose of the Device Mainlining project is specifically to help large companies overcome obstacles to mainlining code for their products. Some of the activities of this project are:

- Promotion of best practices for corporate guidance (e.g. this white paper)
- Collection and organization of links to mainlining tutorials and training materials
- Development of tools and information to assist companies with mainlining-related tasks
  - Mainlining status analysis tools (see https://github.com/tbird20d/upstream-analysis-tools)
  - Quantification of costs associated with out-of-tree code
- Providing assistance for upstream maintainers

Further information about this project is located at:
http://elinux.org/CE_Workgroup_Device_Mainlining_Project

If you are interested in participating in any of these activities, please contact the author at tim.bird@sonymobile.com

Additional resources are available on the elinux wiki, at http://elinux.org/Kernel_Mainlining

Finally, a special thanks goes to people who reviewed this document and gave valuable feedback and suggestions: Kevin Hilman, Frank Rowand, and Kate Stewart.

Glossary

IP block

A definition of a feature implemented in hardware that can be integrated into a larger chip. One example would be a USB controller block on an SOC. The same IP block may be integrated into processors from different chip vendors. IP stands for “Intelectual Property”

Mainline kernel

The kernel source as published by Linus Torvalds on kernel.org.

Mainlining

The process of contributing code to the mainline kernel.

Out-of-tree

Code that is not in the mainline kernel.

Product treadmill

A metaphor for the continuing, focused effort of a product team to meet product delivery deadlines.

Proxy problem

The problem where someone besides the original author of a patch tries to mainline it, and may not have as much knowledge of the hardware or problem domain as the original developer.

Upstream

The origin of the software that you are working with. For example, a phone vendor's upstream kernel supplier might be an SOC vendor. Also (further) upstream might be Google's Android team and mainline.

Version gap

The difference between the version of source shipped in a product and the current mainline source.