Leveraging Open Source Software in Consumer Electronics

Sony ▌Tim Bird

1. Introduction

Software is becoming an increasingly important part of consumer electronics products. Where in 1988 a television set commonly contained only about 8 Kilobytes of ROM, nowadays (2008) it is possible to find high-end television sets with 64 Megabytes of flash. This represents an 8000-fold increase in the storage space for software, over a time span of about 20 years. A substantial portion of this storage is used to hold actual code - including the operating system, drivers, libraries, middleware and applications which operate the television hardware and provide advanced features and user interfaces. Products in other categories, such as mobile phones, digital set-top boxes, personal digital assistants, audio players, and cameras have seen similar software size increases.

As the amount of software in a product grows, and as the amount of time desired to ship a product shrinks, it becomes more important to utilize existing software to reduce development costs. However, as products provide more and more features to satisfy customer expectations, the complexity of the required software grows. The consumer electronics market long ago reached the point where it is not feasible for an in-house development team to write all of the software, from scratch, for a modern CE product.

Almost every major consumer electronics company now ships at least some products containing the Linux operating system. Some companies have adopted Linux for many of their products. Besides the Linux kernel, several other pieces of open source software are often utilized as well, ranging from the GNU C library and the busybox utility suite, to graphics middleware and vertical application stacks. The primary reasons cited by companies for choosing open source software, versus external proprietary software, are 1) to retain control over the software used and 2) to reduce development costs.

2. Software development costs

There are several factors which affect the cost of software in a product. Obviously, the cost to create the software in the first place is a key expense. The desire to reduce this expense is the primary motivator to use existing open source software.

However, several other software-related costs are also important. In the short-term (a single product development cycle), a company also has to:

1) customize existing software,

2) integrate software from different sources, and

3) test and debug the software

In the longer term, a company also has to maintain the software they use. This consists of debugging, fixing and testing the software for product support. It also includes, over time, working on the software for subsequent products releases.

The central tenet of this paper is that how a company chooses to interact with the open source community has a large impact on all of these costs, especially over the long term.

Before moving on, it should be noted that open source is not the only source of pre-existing software for products. Most consumer products which include Linux also include software from a variety of other sources, including custom-built proprietary software from in-house developers or contractors, off-the-shelf components from 3rd party software vendors, and drivers and software libraries from hardware makers. Re-use issues for these non-open-source components is outside the scope of this paper. However, it should be noted that the variety of these sources adds to the complexity and integration costs for a product.

Let's discuss the above-mentioned development costs in turn:

Two key factors in using any external software (including open source) are the cost to customize the software for use with the chosen hardware for a product, and to integrate different software components together for the final product. The amount of customization required for a product depends both on the hardware being used (for example, whether it is similar to existing, supported hardware) and on the requirements of the software.

Open source software has a spectrum of maturity levels. The Linux kernel, C library, and standard Linux utilities (e.g. Busybox) are quite mature at their core. However, there is always work to be done to support new hardware (for example, with drivers and board support packages). Also, work is ongoing in the Linux kernel to support changes in product requirements and specifications. For example, one issue actively being worked on in the Linux kernel is the scalability of flash filesystems. As flash memory sizes increase, different file system and memory management algorithms are needed to make most efficient use of these parts, to provide good boot-time and run-time performance. A few new flash file systems for Linux are under development at this time[1].

As the body of Linux software grows, it becomes easier and easier to find examples of similar drivers and board support packages which can be re-used for your embedded needs. This is the "network effect" (using a term from economics) which drives open source software's success. That is, the more people who are using and developing a particular piece of open source software (like the kernel), the more utility it has to all of it's users.

Another large development expense is the cost of integration, which is the cost of putting different software pieces together. As the number of different sources of software increases, the cost of integration also increases. With open source software, companies that use components that have already been used together benefit from testing and debugging of those prior uses.

Both customization and integration expenses are related to the ratio of newly created software to already-existing software in a product. In general, as the amount of software required for a product increases, there is a natural incentive to replace all non-differentiating software with code that is externally created, available for little to no expense, and which has been previously integrated within itself.

That is, it is desirable to use open source as commodity code, so that a company can focus as many resources as possible on the differentiating features (and software) in their products.

It is interesting to note that for some components, once a piece of open source software reaches a particular maturity and feature level, it is basically senseless to not use it. For some product categories, the Linux kernel has already reached this state. Since its acquisition cost is zero (assuming one obtains it from kernel.org), the cost of using the Linux kernel for a project consists of ONLY the other three costs (customization, integration and testing). Once the decision to use Linux is made, reducing the cost for these activities becomes an important goal.

Eventually, as other open source components become mature and featureful, they will also be adopted into products. Thus, the amount of open source software in Consumer Electronics products will inevitably grow, and the importance of efficiently leveraging open source will grow as well.

Finally, another large expense in product development is testing and debugging. Any changes made to existing software have to be tested, and checked for correct functionality. This means that for larger amounts of customization, more testing is required. At its core, Linux is extremely well-tested, since it is used across a wide spectrum of hardware and usage scenarios (ranging from very small devices, to desktops, to servers and supercomputers). This broad range of use and huge number of developers means that Linux receives very thorough testing. However, CE products often have unique hardware and new features. In these areas, the Linux kernel is less well-tested. But the amount of testing available on similar hardware, and for similar features, that is performed by outside developers is still a valuable benefit of using open source.

Using open source incurs costs similar to other software in terms of customization, integration and testing. However, it adds additional obligations for license compliance. Why, then, do companies use it over software from other external sources? The simple answer is that Linux software is growing in value based on the efforts of thousands of developers. The value of re-using such a large body of software outweighs these other costs.

Linux is now a very commonly used operating system in embedded devices. It will be used more and more in the future. The question then becomes, what is the best strategy for leveraging open source software. How can the above-mentioned costs be reduced? In general, the answer is "by participating in the open source process". This consists of working with the other developers in the open source community to actively push your changes back to the projects of origin. This adds an additional expense to using open source software, but one which pays off in the long run.

3. Publishing vs. mainlining

Most open source software is provided under a license which requires that the entity which distributes it also publishes any enhancements that they have created. The most commonly encountered license is the GNU General Public License, but there are other open source licenses as well[2]. Legally, a company may not distribute such software in their products unless they abide by the license and publish any derivative works that they have made. Compliance with this obligation is now common throughout the industry, and needs no additional discussion here.

However, there is a difference between merely publishing one's derivative works, and actively working to incorporate one's enhancements into the original body of software. This latter activity is called "mainlining". Mainlining encompasses a number of different activities, ranging from describing required features, to submitting bug reports and code to upstream projects, to justification and promotion of new implementations.

The notion of mainlining one's derivative works to a code base is a relatively new one, and may be surprising to some established businesses. Traditionally, developers using software from an external source have had limited means to incorporate their changes into the original software base. In the case of software obtained from proprietary vendors, a developer could request feature enhancements, but it was not common for them to actually deliver their own code, developed in-house, to the 3rd party vendor. In the case of open source this is not only allowed, it is strongly encouraged. And it is the normal manner by which open source software progresses.

Mainlining is an additional cost that is not strictly required by open source licenses. However, companies can benefit from mainlining their changes, if done correctly.

The specifics of how to mainline code and/or participate in the community are dependent on each software package, and the details are too numerous to mention here. However a general overview of recommended steps is as follows:

- Hire or designate one or a few developers who will specialize in this activity. Often, companies hire people who are already experienced community participants, to overcome the steep learning curve involved (discussed later).
- Have your specialists sign up to mailing lists where relevant items are discussed (e.g. linux-embedded@vger.kernel.org for kernel issues related to embedded products)
- Have them learn the accepted practices for formatting and code submission.
- Have them take the customizations you are making (both ideas and code), and send them back to the original project. This last step sounds easy, but depending on the project can be quite difficult.

Some additional recommendations for this process will be explained below.

Depending on the extent of your Linux usage, the cost of this could be the part-time effort of a single engineer, or the full-time effort of several engineers. It is recommended to assign somewhere between 5% to 10% of your total engineering effort on Linux to the mainlining activity.

3.1 Benefits of mainlining

There are many benefits, both short-term and long-term for mainlining.

3.1.1 Quick feedback

First, by describing your problem and/or posting your code, you can receive quick feedback from other developers on your ideas and implementation. This is true whether your code is ever accepted or not.

Mainline project developers may comment on your code, including your basic ideas and your implementation. This can give you useful feedback while you are still developing a feature, to refine it. In some cases it may prod you to take an entirely different approach. Or, you may find that a solution already exists, that will save you time to use. In order for this feedback to be useful, it is obviously better to receive it while you are still developing the software in question. Thus, bug reports, fixes, and new code should be submitted as soon as possible in your development cycle.

3.1.2 Testing and bugfixes

Code which is actively pushed to the community may be tested by a large number of other developers, in ways you never imagined. This can reveal bugs in the code which would not otherwise be found.

For example, one change I submitted to the Linux kernel was to add a high-resolution timestamp to the kernel print function, to aid in measuring kernel bootup time. I had been using this code for a few years in my own projects (in violation of my own current advice to submit things early). However, within a few days after submitting it I received bug reports from developers who found problems when the code was called with bizarre parameters and in weird configurations. These bugs could have appeared in my own product, without warning, requiring substantial time and effort to find and fix. By mainlining the code, this problem was avoided.

In this particular case, the feedback was received very quickly. Note that I received this feedback even before the code was accepted for inclusion. When code is accepted into the project of origin, there is more possibility for widespread use, testing, and improvement over a period of time.

3.1.3 Free maintenance as Linux changes

Linux changes quickly. For the past few years, the Linux kernel has averaged approximately 6,600 lines of changes per day[3]. Because of customization, integration and testing costs, CE companies tend to stay with a particular release of Linux for a long time, for a particular product line. However, eventually a company will want to adopt a new release of Linux in order to use new features in the OS. When this happens, any customization that has occurred is likely to not apply (or not apply easily) to the new version. It often requires a lot of work to move features and bugfixes to the new release. Sometimes, the effort is so large that the feature is just dropped in the move to the newer version.

When a change is accepted into the Linux kernel, it becomes possible for other developers to work on it. Often, when changes are made to related systems or interfaces, other developers will perform the work necessary to change your code as well. Sometimes you will be asked to do the work, but in any event, you will get timely notification of the changes required. This makes the adaptation effort much easier -- either because it is done for you, or because you can do it while the change required is still minimal and other interested developers are available to provide fresh information and assistance.

I once submitted a change to the kernel to improve bootup time by changing the way a particular calibration was performed on kernel initialization. I continued to use the feature via a configuration option for several kernel releases (It is still used in Sony products today). After a few years, I wanted to examine the change, and couldn't find it. After searching for it, I found that it had been moved and rewritten to adapt to changes in the calibration function. This is an isolated case (I don't recommend losing track of your code), but I had been using an improved version of my own enhancement for a long time, without even realizing it!

The end result of this is that mainlining allows you to reduce your development cost, both immediately in terms of quick feedback and testing, whether your change is accepted or not, and over the longer term, as you reduce or eliminate your effort to maintain your own enhancements.

3.2 Obstacles to Mainlining

While mainlining brings benefits, it can be an expensive and frustrating process, especially to the inexperienced. Several factors raise obstacles that must be overcome to be effective at mainlining.

- Infamiliarity with process
- Version gap
- Product treadmill consumes all resources

One of the main costs of mainlining, as opposed to just publishing, is the amount of process involved. There is a very steep learning curve for people who are inexperienced with open source practices. For a change to be accepted by other developers, it must be of high quality, and it must conform to many rules. Also, it must be submitted to the right developers, in the right manner. Responses to submitted code is often terse, and sometimes quite harsh. The level of brutal honesty, particularly in the kernel development forums, can be discouraging and takes a bit of getting used to.

For reasons of efficiency, code from new contributors is not given the same attention and review as that of well-recognized contributors. New contributors are not trusted. Often, code must be reworked and submitted multiple times, in order to make it acceptable for inclusion. This may take weeks or months of steady effort.

For these reasons, it is important that a company have specialists who work on the task of mainlining. It is not efficient for a company to try to get all their Linux developers involved in community processes. As a developer gains proficiency, adhering to community processes becomes easier. Also, they build up credentials within the community which lowers the barrier to entry for their submissions. A single well-known, experienced, contributor can have much greater success at mainlining than a group of infrequent contributors.

Another big obstacle to mainlining code is that often, the version of the kernel used for embedded projects is somewhat behind the one actively being developed in the original project. For example, some companies are still doing development on kernel version 2.6.11, which was a popular release of the Linux kernel. That version was released over 3 years ago. Any changes made to this kernel are likely to be difficult to use with the current kernel version. Even changes which are useful will require an additional porting effort to make them work with the latest kernel version. This additional porting effort, and lack of immediate relevance to the current kernel, is a significant barrier to the mainlining effort.

Finally, companies often have their engineers so busy developing for each product release, that no time is allocated for the extra work of mainlining code. Product engineers are specialists who are in short supply, and companies are always under extreme time pressure to release their products. For most engineers, the extra expense of additionally working with the community in mainlining efforts is not worthwhile. It makes more sense to utilize engineers who are outside of the normal product development cycle.

3.3 Concern over loss of differentiation

Another big obstacle to mainlining is a concern within the company about loss of differentiation due to publishing code to competitors. Hopefully, the obligation to publish derivative works is fully understood when a company decides to use open source. However, there may still be people who believe that a competitive advantage can be obtained by delaying or not actively promoting the required publication of enhancements. The act of not mainlining enhancements is sometimes called "hoarding".

The rationale for hoarding is that a competitor may gain access to the implementation, and subsequently be able to use it for "free". Companies have long considered their development efforts as one of the principle means to differentiate themselves from their competitors. Thus, in every corporation the result of software development is, by default, viewed as a highly valuable item to be guarded closely and prevented from disclosure at all costs.

For some software (the part not based on open source), this approach is correct. However, differentiation in the area of commoditized software is a bad. You actually don't want you software differing from that of other companies, because then your costs to customize, integrate and test that software will be bigger than your competitor's. You would end up losing some of the key benefits of open source.

A better way to look at this is that you want as much of your development work integrated into the commodity base as possible, in order to avoid losing your development effort.

In this regard, open source software is very much like standards. Standards are used to de-fragment an industry – to provide an interoperability benefit to the customer and ultimately to allow the industry to grow. Companies often act vigorously to have their implementation adopted for a standard, because they know it will create delays and obstacles for their competitors, who have to adopt their code and protocols instead of using their own.

The same is true of open source. A developer who successfully contributes their code to an open source project reduces their own maintenance expenses, while simultaneously increasing the long term development cost for anyone else currently using a competing implementation.

As a hypothetical example, let's say that company A has developed a large set of bugfixes for the USB stack of the Linux kernel. Company A could "hoard" these fixes, and use them to gain a competitive advantage. However, if the interfaces to the USB stack change in a new version of the kernel (which is very likely), then Company A's fixes will no longer be relevant. In order to use a new version of the kernel, company A will have to do a costly re-write and re-test of their fixes. Over time, the sunken cost of USB testing and fixing for Company A could actually turn into a barrier for them to adopt a later kernel version. Meanwhile, company B (a competitor to A) never received the benefit of company A's USB testing and fixing, but they were able to move to a new kernel more easily, and they received the benefit of the whole rest of the community's testing and fixes. It is very likely that the fixes from the community were more numerous and better than the ones from Company A, since the community itself is much larger than company A's Linux development staff.

In this case, when company A wishes to move to a new kernel version, its own hoarding activities force company A into having to decide between their own development effort, and the development effort of the community. Hoarding can result in a short-term advantage, but ultimately it puts company A at a long-term disadvantage relative to its competitors (Note also, that even with hoarding, company A's advantage is short-lived since they are required by license to publish their changes when they ship their product anyway).

Put another way, you can't prevent your competitors from accessing your enhancements to open source for very long, and mainlining is the best way to preserve your development effort for your own re-use. So you should actively mainline your changes as part of your open source strategy.

4. Recommended Practices

This section describes a few practices that have been found to be effective in overcoming obstacles to mainlining. This is something of a random collection of different practices useful to avoid costs and help the mainline effort. Additional good guidelines for embedded developers are available[4].

4.1 Submit early and often

As the size of code grows, it becomes harder to change it. Small corrections performed early in the development of a new feature can avoid expensive major re-writes later on. With a mature and complex code base like the kernel, the chance of getting your code right the first time is extremely small. Getting feedback early is important to avoid wasting time and effort.

Don't wait until the end of your product development cycle to release your code (This is a common mistake).

4.2 Break changes into the smallest chunks possible

The chance of having code accepted decreases with increasing size of the code base. Other developers are as time-constrained as you are, and often do not have time to look at large change sets. Submitting code in small chunks is much more effective at getting feedback.

4.3 Keep track of your patches

A specific problem for many CE product developers is that their changes are not maintained in a format that is easy to mainline. For example, developers often do not manage their source code so that their change sets can be easily turned into standard-format patch files. If possible, use a source code management system that is similar to, or compatible with, those used for the original project (eg. 'git' for the Linux kernel).

Keep your changes available as patches, and keep track of patches you have integrated from other sources, which themselves have not been mainlined (The 'quilt' patch management tool is good for this, and is highly recommended). If you decide to upgrade a patch you have integrated from another source (for example, moving to a newer release of squashfs) it is much easier if you have kept the original change set boundaries intact.

Whatever you do, don't just keep all your code in some monolithic source tree, with no means to discern the distinct change sets relative to upstream sources.

4.4 Do initial development on the latest kernel version

This may be counter-intuitive, but sometimes it is useful to do your initial development of some feature or bugfix on the latest version of the kernel, and backport it to the version you are using in your product[4]. If you do this, you can take advantage of input from other developers in the community during your development. It is often less expensive to back-port code to an older kernel version than to up-port it to a recent version (although you should definitely check to make sure that required interfaces are similar between the two kernel versions).

5. Conclusion

Open source software will continue to be used in CE products. Companies can reduce their development costs associated with this use, by actively participating in the open source community. Mainlining one's code to upstream projects yields short-term and long-term benefits which outweigh the expense involved, and the advantage of temporarily keeping enhancements away from one's competitors.

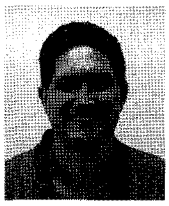
References

[1] LogFs and UBIFS are two recently created file systems that address scalability and other issues with previous Linux flash filesystems.

[2] The Open Source Initiative maintains a list of licenses which meet the open source definition. See http://www.opensource.org/licenses

[3] Greg Kroah-Hartman, et al., "Linux Kernel Development - How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It" April 2008, 29 May 2008 〈http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php〉

[4] Andrew Morton, "kernel.org development and the embedded world", April 2008, 29 May 2008 〈http://www.celinux.org/elc08_presentations/morton-elc-08.ppt〉



Tim Bird

Tim Bird is a senior software engineer for Sony Corporation of America, in their Silicon Valley Software Group. Tim helps customize the Linux kernel for use in Sony products. Also, Tim represents Sony in the CE Linux Forum. He is Chair of the CELF Architecture Group, where he directs initiatives designed to improve Linux for use in embedded products. Tim was formerly CTO of Lineo, one of the first embedded Linux vendors, and has been working with Linux for over 15 years
E-mail : tim.bird@am.sony.com