WHITEPAPER

QNX | BlackBerry Subsidiary

Total cost of ownership – Linux vs. QNX realtime operating system – Part 2

Chris Ault
Senior Product Manager

QNX Software Systems

2/12/16

Total cost of ownership – Linux vs QNX realtime operating system – Part 2

Overview

The Linux operating system provides for open access to its source code. This has led companies to initially choose Linux as a viable development platform, on the perceived basis that its cost is less than commercial alternatives.

To examine this in earnest, we present a three-part whitepaper series that closely examines the total cost of ownership (TCO) of Linux and how that compares to a commercial off-the-shelf realtime operating system like QNX. Specifically, we illustrate life-cycle costs that aren't initially considered, but that have a far bigger contribution to the TCO than a commercial production licensing cost.

In the 1st whitepaper of this series we examined:

- *Upfront costs*
- *Selecting the right version of OS*
- *Time to market*

In this paper we will focus on the costs of maintaining Linux and delve into:

- *Patch management and version alignment*
- *Supporting hardware – drivers*
- *Maintenance costs*
- *Design/support services*

In our 3rd and final paper of the series, we will examine:

- *The challenges of certifying a Linux-based system*
- *GPL concerns.*

Content Management Overhead: Build, Support, Test, and Maintain

In Part 1 of our series we examined the challenges as they pertained to selecting the right version of Linux. In the spirit of open source, there are numerous versions of Linux that have been created by users to meet their needs. Users create unique versions of tools, libraries, drivers, or even operating system components. In some cases, there are already multiple, competing implementations of a particular component (e.g., in the domain of security, there are multiple approaches – grsec, AppArmor, SELinux, SMACK, and TOMOYO to name of few of the popular ones), and all that's required is that the person chooses a different component than the one that's shipped with a particular Linux distribution.

The important realization, as it relates to TCO, is that it now becomes your job to configure, build, support, certify, test, and maintain a large code-base that is the foundation of your product but likely has nothing to do with your value proposition or differentiation. You now need to hire Linux experts just to provide those functions, let alone the developers you need who are actually concerned with your product proper. And, not just "generic" Linux experts – a medium to large project might require such diverse skill-sets as kernel, GUI, middleware, drivers, networking stack, and so on, which are often hard to find, let alone embodied in one individual. If you want to minimize the amount of recurring cost for each new project, you'll want to ensure your fixes or patches are included in future releases. This requires a level of commitment to the target community as well as a development team that is not just technical but also contains consensus builders and influencers in that community – another major source of recurring cost.

In the 1st whitepaper we discussed configuration, in terms of determining which version and patches to get. License compliance is another major area to consider. Using Linux, you are on your own – it's up to you to ensure that you are using the correct licenses, and that you've attributed them properly. This isn't just a simple matter of duplicating the copyright notice in an appendix – an inadvertent "copy and paste" of open-source code can result in contamination of your proprietary source base. When using QNX, there's a licensing support system that helps ensure the provenance of all of your source code.

Support and maintenance, are yet another significant cost. This is because the individual Linux components often come from many diverse sources, especially if you had to go and get additional patch-sets. Firstly, these are not all distributed at the same time – they are independently developed, independently maintained, and independently distributed. This means that at some random point in time, an update becomes available for a part of your system. You need to continually evaluate all such updates as they become available, and determine if you will accept, defer, or reject each one, and what impact that will have to your integration, testing, certification and release schedules. So now, in addition to increased development team investment, you're locked into a commitment of distilling patch deltas from the community for the life cycle of your product.

Patch Management and Version Alignment

Another wrinkle to consider with Linux is the sheer number of possible combinations of configurations and the fact that some updates may not work with other updates. This means that when a critical update arrives, it may not just be a matter of simply adding it to the release. Many patches or changes have ripple effects and dependencies that can be far reaching, and

most Linux development communities don't regard embedded development requirements as a high priority in most decisions on breakages or impacts.

It's highly possible to have a small library fix that requires a large ripple of rebuilds and changes that, while annoying for desktop users, could be an expensive retest and recertification effort for embedded or safety certified devices.

It's highly possible to have a small library fix that requires a large ripple of rebuilds and changes that, while annoying for desktop users could be an expensive retest and recertification effort for embedded or safety certified devices.

With regards to certification, since the patch isn't being specifically developed for your class of device (e.g., a medical device), you will need to re-certify the entire component that the patch affects. Even more challenging, you will need to address the "dead code" issue for each patch set that arrives – does the current patch contain code that is pertinent to the operation of your device?

We will examine the dual issues of dead code and software of unknown provenance (SOUP), and their impact of certifying a device, in further detail in the 3rd part of our whitepaper series.

Endless Velocity of Patches

According to The Linux Foundation, there are over 1,400 developers, from over 200 different companies, contributing to every kernel release. There are more than 10,000 patches going into each recent kernel release. Since 2005, 11,800 individual developers, from nearly 1,200 different companies have contributed to the kernel. This is just the kernel alone, let alone other large pieces such as the development tools, GUI, libraries, and so on. While impressive as a feat of software collaboration, it should also be sobering: that much change means that areas of the system that you aren't thoroughly testing are being changed – constantly! This then means that you must perform exhaustive testing on all aspects of the system. This is a massive test effort, compared to the test effort that you exert on the pieces of the system that are relevant to your product. To put that into perspective, consider the numbers since the 3.10 kernel release (October, 2011)– the Linux Foundation states, "Since the release of the 3.10 kernel, the development community has been merging patches at an average rate of 7.71 patches per hour." This works out to just under 8 minutes per patch. Keeping up with this rate of change in terms of accepting or rejecting the patch, validating the ripple through effects of the change (e.g., dead code), and testing is a monumental task. It's important to keep in mind that this isn't an optional activity – a patch may come in that fixes a serious bug. To re-iterate: this is for the kernel only!

> "…the development community has been merging patches at an average rate of 7.71 patches per hour"
> – The Linux Foundation

Supporting Hardware

Another important aspect when considering TCO is finding a version or configuration that supports your hardware. There are several paths:

1. Find a version that supports your hardware "out of the box."
2. Find a hardware vendor that's willing to port / update their driver to work with your selected version of Linux.
3. Write your own driver.

If there is a driver available, Linux turns out to be its own worst enemy on two fronts (we discuss the second one in the next whitepaper). First of all, because of the massive number of developers making changes, Linux has a kernel / driver "interlock." This mechanism is in place to ensure that the kernel's internal data structures are compatible with those used by drivers. Because drivers can be loaded dynamically, both the kernel and the driver need to agree on the layout and content of certain shared data structures. Should this layout and / or content change from one version of the kernel to another, the driver may misinterpret the data. This will most likely result in a crash or hang. For this reason, an interlock mechanism exists: it ensures that drivers written for a certain version of the kernel are operable only with that version of the kernel.

The impact of this is that a third-party driver that works with your hardware today may not necessarily work with updated kernels in the future; you are now at the mercy of the third-party to update their drivers to keep up with future versions of the kernel. Because the third party also has limited resources, they may not necessarily update their drivers in lock step with kernel updates, and may instead only produce drivers on a calendar basis, or an on-demand (i.e., paid NRE) basis, etc.

Kernel / Driver Interlock

If you take the third route, that is, write your own driver, you will be subject to all of the problems with the GPL (as well as an additional, follow-on impact), and you will also be in a position of maintaining even more source code that isn't directly related to your product (we'll talk more about the GPL in the 3rd part of this whitepaper series). Even if you are comfortable with releasing the driver's source code (as required by the GPL), you may find that you're not actually in a position to do so, even though you wrote the code. This is because the hardware vendor most likely won't release their documentation to you in the first place, unless you sign a non-disclosure agreement (NDA). Of course, the NDA will prevent you from releasing details of the hardware. Since the driver is mostly concerned with the details of the hardware, you'll be in violation of the vendor's NDA if you release your driver's source code. Another consideration

relating to this is that leading edge hardware will generally have more IP associated with it – IP that the vendor doesn't want their competition knowing about. Thus, you may find limits imposed on hardware availability that work directly in opposition to your requirements for hardware. That is, less desirable hardware has readily available drivers, whereas more desirable hardware doesn't. It's ironic that the very license that was supposed to prevent this kind of thing from happening exactly exacerbates the situation.

Maintenance

As mentioned above, when an update to a Linux component is made, it arrives asynchronously to other distributions.

The challenge is not only in determining when and if to apply the updates, but how. Generally, patches are implemented as a set of differences ("diffs") from the last release to the current release. If you are maintaining multiple patch sets (i.e., multiple, different systems impacting overlapping portions of the source base), you may find that the amount of effort required to merge the many patches becomes a significant component of your TCO. This is because you will be receiving patch-sets relative to the main branch that the developers are maintaining, not your particular version of it. As your branch deviates further (and further) from the main branch, the patch sets may become less and less directly usable, and your maintenance costs will increase further.

However, standards and compliance requirements in certain fields (e.g. IEC 62304), require you to maintain the system post-deployment. This means that the overhead associated with merging updates and patches doesn't end when the product ships out the door, but rather is an ongoing activity. This has a further "knock-on" effect in that your test and (re)certification efforts run at full budget post-ship.

As mentioned above, when you assume the burden of maintenance, you are defocusing your development effort. In terms of coding styles, development languages, scripting languages, build systems, and so on, your organization has probably standardized on a very small set of choices (i.e., one or a few). This is usually done for reasons of developer efficiency – finding a developer who is proficient in many different system programming languages and scripting languages is more difficult than finding ones with more focused skill sets. When you add the maintenance burden of a multi-million line Linux distribution to your workload, you suddenly require all kinds of additional expertise that would not be otherwise required.

Support Services

Due to Linux's "community" development model, it's often difficult to know where to turn to for help. Especially when it relates to a system-level problem (such as speed or memory usage, for example), that spans multiple subsystems. Volunteers, who have experience ranging from newbie to kernel expert, across one or more systems, do most of the Linux support. In such systems, a polling method needs to be applied to various IRC chat channels and wiki sites. If you're lucky, there might be an available expert on IRC able to help you out. Unfortunately, though, timeframes are chaotic – an expert may help you immediately, or your issue may be sitting for weeks on end. Worse, the answer to your issue might be "upgrade to the latest version" or "apply this patch" – both of which are fertile ground for additional support issues. Commercial Linux support exists, of course – but do realize that the commercial support

company is going to have to incur similar TCO costs as you will in order to support your customized version of Linux – and since this is the basis for their business, they will need to not only cover costs, but also make a profit.

Another factor to consider is level of service. Linux is used in a large number of diverse environments, and includes environments that are at the opposite ends of the spectrum from embedded systems (e.g., desktops and servers with almost unbounded memory and disk space, and non-realtime processing requirements). The key fact here is that because it is a community development model, the community is not obligated to help you make any particular changes to any system. For example, if you find that something is using up too much memory, it may be the case that the design goals for that component don't consider memory as an important requirement. Therefore, you'll get very little traction with bug reports. The worst-case impact to TCO in this case is that you will need to fix it yourself (and then maintain your changes against all future patches from upstream). After all, the Linux model encourages you to get the code and fix it yourself, or fork your own project and run with it.

You can contract Linux experts on an “as-needed” basis, but it makes cost control and budget control very difficult and costly. Additionally, availability of these experts may not fall within your timelines. Traditionally, patch sets and fixes that are not upstreamed to the community become development costs that increase with each new update you merge and maintain. Slowly, projects begin to shy away from updates, in order to minimize the cost of the constant merge / maintenance work. But you can’t really get away from it, because of post-deployment maintenance obligations (you at least need to keep up to date with security patches). One of the big goals of using open source is to continue to leverage the “goodness,” but if you must sacrifice that pillar to minimize other overhead costs, you’ve lost that value proposition as well.

Conclusion

Maintaining a Linux-based system is very costly. The decision to adopt a Linux OS diverts focus from product innovation to OS integration and maintenance. A great deal of resources and time need to be spent on configuring, building, supporting, maintaining and testing the OS. Testing Linux alone contributes greatly to the total cost of ownership of the OS. With the constant changes, patches and versions of Linux, a huge investment in exhaustive testing becomes imperative.

In the 3rd part of this whitepaper series, we explore the differences of certifying a Linux-based system versus a QNX-based system and the applicability of Linux for various market applications.

About QNX Software Systems

QNX Software Systems Limited, a subsidiary of BlackBerry Limited, was founded in 1980 and is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Siemens, General Electric, Cisco, and Lockheed Martin depend on QNX technology for their in-car electronics, medical devices, industrial automation systems, network routers, and other mission- or life-critical applications. Visit www.qnx.com and facebook.com/QNXSoftwareSystems, and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

www.qnx.com