WHITEPAPER

QNX | BlackBerry Subsidiary

Total cost of ownership – Linux vs. QNX realtime operating system – Part 1

Chris Ault
Senior Product Manager

QNX Software Systems

1/19/16

Total cost of ownership - Linux vs QNX realtime operating system - Part 1

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 this introductory whitepaper we will examine:

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

In our 2nd paper of the series, 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.*

Free software

Linux is generally touted as a “free” operating system, and in certain regards, this is indeed the case. For the most part, you can download the source code to all components of the Linux distribution or “distro” (that is, the operating system, drivers, utilities, tools, libraries, and so on). After all, Linux originated as a hobbyist-level effort to come up with a free version of Unix, and belongs to the general class of software known as “Free Open Source Software” (FOSS). Linux is maintained by the Linux community, which includes hobbyists, volunteers, academic institutions, commercial hardware vendors, and commercial support organizations, to name a few. You're free to modify any part of the Linux system you see fit, create a product, and sell it. In this respect, Linux appears to have a cost advantage, in that there are neither source code nor end-product licensing costs.

However, considering only those costs gives an incomplete, and therefore skewed, analysis of Linux's TCO. A better way to think about it is that Linux is free like a free puppy. Anyone who has a dog knows the never-ending expenses that pet ownership can entail, from veterinarian bills to food costs to the time invested for training — together, these represent the true total cost of ownership of a puppy. Likewise, the total cost of ownership for a “free” Linux OS includes the extra effort and testing needed to certify a system that uses an open source OS, the cost (or revenue lost due to delays) in bringing the device to market, and the investment needed to sustain an in-house team of OS experts.

It is critical when evaluating the cost of an OS to include such big-ticket items as development cost, maintenance cost, support cost, and opportunity cost – not to mention the fact that by building your own Linux distro, you end up with a build of the OS that is unique to you. Also, the contribution to the TCO of Linux's licensing model, the GNU Public License, (GPL for short), is a significant factor.

Development life-cycle

If you're using Linux, your product development life-cycle loosely consists of the following steps:

1. Select, get, and configure the Linux source base.

2. Configure for cross-compilation environment for the target.

3. Create / assemble a target image.

4. Develop, test, and release your product.

5. Provide product support post-release.

The first two steps are unique to using Linux – you need to find an appropriate version of Linux to use, and then get the tool-chain working so that you can produce executable images for your target platform. The remaining steps are, at a high level, the same for Linux and QNX with the exception that with Linux, you also are also taking on the ongoing maintenance and enhancements for your custom Linux OS build. Let's examine the steps that are unique to Linux to better understand where the hidden costs are.

Getting the right version

As an open source solution, Linux encourages lots of people to make lots of contributions and changes to the source base. When someone disagrees with the direction a certain tool, library, driver, or even operating system component has taken, that person has a number of options open to them. In some cases, there are already multiple, competing implementations of a particular component, and all that's required is that the person chooses a different component than the one that's shipped with a particular Linux distribution. They then maintain this change going forward, and have effectively customized their distribution. On the other hand, if there is no other particular component that suits them, then they are welcome to “fork” their own version of that component. What this means is that they take a snapshot of the component's source code as it exists at some moment in time, and go off and make their changes. They may or may not want to, or indeed be able to, re-integrate their changes with the main branch. It turns out that this is sometimes driven as much by politics and personalities, as it is by technical merit.

To put into perspective how many people contribute, customize and alter the Linux kernel, a February 2015 report from the Linux Foundation, entitled Linux Kernel Development, stated that nearly 12,000 developers from more than 1,200 companies have contributed to the Linux kernel since tracking began 10 years ago.[1]

To put into perspective how many people contribute, customize and alter the Linux kernel, a February 2015 report from the Linux Foundation, entitled Linux Kernel Development stated that nearly 12,000 developers from more than 1,200 companies have contributed to the Linux kernel since tracking began 10 years ago.

As a solution for hobbyists, researchers, certain end-users, etc., Linux can be a good fit. However, as a solution for basing a stable, safety-critical product (such as in the medical, automotive ADAS, industrial safety or safety critical military devices, for example), this leaves much to be desired. In fact, many distributors, network providers, and customers will not buy a system unless they can trace everything, including the OS, back to its original sources. There was a recent case where a cell phone maker was using a distro based on the Linux kernel and because of the lack of traceability with Linux, was unable to sell their phones through discerning partners until the traceability was resolved. Meanwhile, QNX has unique, patented systems that

[1] http://www.linuxfoundation.org/news-media/announcements/2015/02/linux-foundation-releases-linux-development-report

trace every build back to source origins and specific commits. The lineage of all source code is known and tracked.

Selecting the proper version of Linux can create a litany of questions. Which version and configuration of Linux is most suitable to my project? Is it the latest version of a particular distribution? Is there a qualitative difference amongst the various different versions? As far as standard support for your C runtime environment goes, do you use glibc? libc4? libc5? uClib? dietlibc? Do you use the same kernel as is used on the desktop, or do you use a specialized set of patches for security (grsec? AppArmor? SELinux? SMACK? TOMOYO?) Do you need realtime support, and thus yet another, different kernel (or at least set of patches)?

At issue is, do all of your selections above result in a tested, stable, complete and maintainable version? Is your custom configuration of patchsets going to be supported by the community, or will you be bearing the full load of supporting this software? More importantly for critical systems, do you have the resources to certify the complete system? According to, for example IEC 62304 (and included standards), there are stringent requirements related to producing risk analyses for all components of the system. Such analyses include traceability of patch sets and modules, ensuring that the development processes are strictly followed, code reviews recorded, branch coverage and code coverage performed, as well as the removal of dead code (that is, code that is not pertinent to the operation of the system). At the very least, you have to be able to demonstrate that the code's presence does not affect the operation of the system. Proving a negative such as that is quite difficult. These are all likely major areas of focus and areas needing control within your project.

Build determinisim

A key aspect to consider in the build cycle is that of build determinism. In certain cases, if you can provide a binary-identical (or at least assembly-language code identical) version of a subsystem, you may be able to bypass, or at least minimize, testing on that subsystem. One of the concerns with Linux, (apart from the fact that it's a monolithic kernel, and any one change can mean a complete retest), is that small or even seemingly unrelated changes, like toolchain changes, may cause ripple-through effects.

In contrast, the QNX realtime OS (RTOS) is designed with a microkernel architecture. This architectuture provides an extensive level of fault containment and recovery. The microkernel architecture means that every driver, protocol stack, filesystem and application runs outside the kernel, in the safety of memory-protected user space. Virtually any component can be automatically restarted without affecting other components or the kernel. The QNX kernel and all of the QNX subsystems are version controlled, tested and you benefit from the collective field testing across millions of other embedded systems using these exact same kernel and subsystem binaries – you do not have to take on the risk of having a unique-to-you build of a kernel.

Because of the extensive development going on in the Linux community, changes are constantly being made. However, they're not just being made to the source code of each subsystem, but also to the very instructions used to build the subsystem. Changes to those instructions, such as changing a compiler optimization flag, can have massive retest requirement effects throughout the system. Since the compiler is now producing different code, in critical devices this means that you need to retest / re-certify the entire image, because it's possible that the differently executing code may have uncovered a latent bug. (Note that the “obvious” solution – freezing the compiler version you are using – is not possible. Linux and its toolset are tightly coupled.)

Naturally, to ensure sanity and control of project timelines you’ll have to standardize on a tool chain revision level and carefully consider any deltas coming from upstream and test each time you move forward. This adds additional IT / build infrastructure to maintaining your repository or baseline of build / development (effectively, your own custom distribution). You’ll need resources in order to monitor and maintain your tool chain – another area which has nothing to do with your product value proposition or product differentiation.

Time to market

A significant issue impacting time-to-market is the availability and quality of device drivers. Ideally, the hardware vendor would create, maintain and support high quality drivers for their hardware. But unfortunately, Linux’s licensing model has a direct (and negative) impact here. Leading-edge hardware is strongly tied to proprietary Intellectual Property (IP) rights – and manufacturers are hesitant to compromise their IP by releasing source code. Some manufacturers will release “binary blobs” – that is, executable / binary images with no source code. Or, they may release a watered-down version of the driver (with source code) that doesn’t compromise the IP they wish to protect. If the provided driver isn’t suitable for your purposes, however, you’ll need to write and maintain your own.   This uses up valuable elapsed time, as well as developer resources, and thus increases your overall cost.

On the other hand, with QNX, there are no legal obligations to disclose source. Because of this, the hardware manufacturers have far less IP exposure when dealing with QNX drivers. Therefore, they are legally less restricted when creating drivers for QNX, or at least in releasing the information under NDA so that you, QNX, or a third party can create and maintain the driver. In addition, because of QNX's modular, microkernel architecture, drivers are stand-alone entities, meaning they are much more isolated from version changes in other components. There is no kernel interlock like there is in Linux.

If the system you are building is for mission critical operations, like those found in medical devices and functional safety systems like industrial controllers and railway safety, the issue of time to market gets exacerbated due to certification challenges. Building a certified system becomes increasingly complex and downright impossible based on the choice of OS, an issue we’ll explore in further detail in part 3 of this whitepaper series.

Conclusion

The Total Cost of Ownership for a project consists of many factors that may not be immediately apparent. A simple decision based on a shallow analysis of the cost of obtaining and licensing the source code is insufficient. Anyone considering using Linux for a project (especially one for a critical system) needs to be cognizant of the costs and considerations articulated in this paper. They must be prepared to make an informed decision about the relative merits of an apparently “free” operating system (with potentially unbounded costs) versus the predictable cost of obtaining and licensing QNX. Not only is QNX a stable, certified platform with a company behind it that provides support, custom engineering services, open source integration and license compliance, but what QNX mainly provides is freedom. Freedom for you to get back to what you do best: developing your own software.

In our 2nd and 3rd paper of the series we will focus on the costs of maintaining Linux and examine the challenges of certifying a Linux-based system.