WHITEPAPER

QNX | BlackBerry Subsidiary

Total cost of ownership: Linux vs. QNX realtime operating system – Part 3

Chris Ault
Senior Product Manager
QNX Software Systems

2/29/16

Total cost of ownership: Linux vs. QNX realtime operating system – Part 3

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 the runtime licensing cost.

In the 1st whitepaper of this series we examined:

- *Upfront costs*
- *Time to market*
- *Applicability to markets and systems*

In the 2nd whitepaper, we focused on the costs of developing and maintaining Linux and delved into:

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

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

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

Compatibility and Standards

Linux, and the open source community in general, are known for extending things in non-standard ways. For example, the Linux kernel uses GCC extensions such as type discovery, ranges within case statements, and a host of other non-standard features[1]. There are several impacts to TCO due to these extensions: they lock you in to that platform, they may be subject to change without notice, they don't necessarily work with other, industry standard tools (e.g., code transformation), and they can cause confusion for developers who are otherwise knowledgeable about standards-compliant implementations. QNX, on the other hand, conforms strongly to the POSIX set of standards. These standards are well vetted, stable, and provide an excellent base on which to build products, test cases and certification suites.

This is more than just a compatibility issue, however; various standards for critical software make a distinction based on what can be loosely be called the "intent" of the software. IEC 62304, for example, specifically calls out Software Of Unknown Provenance (SOUP) as code that doesn't have formal documentation, or evidence as to the controls on the development process. While Linux epitomizes this approach, QNX on the other hand takes a much more formalized path, having gone so far as to have developed a patented source code traceability system. This means that QNX is much better suited to safety critical devices. Whereas QNX has a roadmap, formalized test, development and support processes, and various certifications (both at the corporate level as well as the product level), Linux tends to be run like the wild west. Even the founder of Linux, Linus Torvalds, has stated that Linux evolves as it needs to, and doesn't have a formal roadmap ("Linux is evolution, not intelligent design"[23]). This approach is fine for hobbyists, researchers, and perhaps even non-safety critical devices, but does not provide any form of traceability for critical systems unless you take on that burden and cost yourself.

The Challenges of Certifying a System

The diagram below shows the growth of the Linux kernel over time. Starting with 2.6.20 (circa February 2007) at 8.1MLOC (Million Lines of Code) through to the latest 4.3 kernel at 21MLOC. Given approximately 100kLOC in the QNX kernel and Process Manager, this shows a growth rate of ***several QNX kernels per release***!

Think of the extra cost of having to certify and review the code.

---

[1] See http://www.ibm.com/developerworks/library/l-gcc-hacks/

[2] *"Perpetual Development: A Model of the Linux Kernel Life Cycle"* (http://www.cs.huji.ac.il/~feit/papers/LinuxDev12JSS.pdf)

[3] *"Linux Evolution"* (http://www.sprg.uniroma2.it/kernelhacking2008/lectures/lkhc08-01b.pdf)

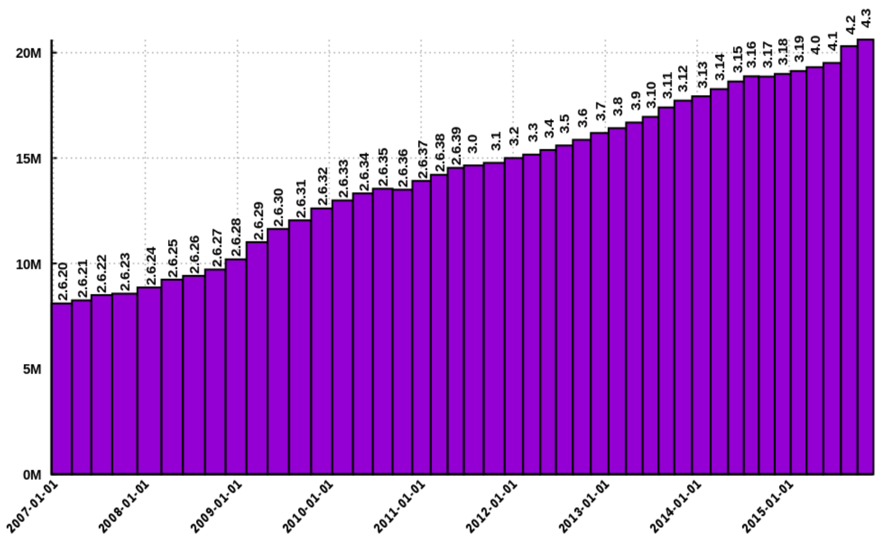

Figure 1, Lines of Code, Linux Kernel[4]

Dead code

The issue of dead code has far-reaching consequences. Dead code is defined as code that cannot be reached during execution, and is not traceable to a requirement. Consider a condition that tests for out of memory. In a device with lots of memory, the code might never be reached during execution (so would seem to be dead code). But the code stems from a requirement: the device needs to respond to an out-of-memory condition by performing some action, and hence is not considered dead code. On the other hand, a condition such as testing to see if a number is positive, and then, within the body of that condition, testing to see if it's less than zero, results in dead code. There's no way that a number is both positive and less than zero, and hence no way to execute the code. Dead code is expressly discouraged by IEC 61508.

When you remove dead code, you are, in effect, creating a private branch with your changes. When you now apply new patches, they may or may not integrate smoothly into your private branch. That's because the automated patching systems depend on context in order to determine where to patch the code. If code has been removed (or even just moved), the context has changed, and thus patches may fail to apply properly – even for live code. What was once dead code (and which you removed) may now become live code based on a new feature or a bug fix. In the example above, the test against the number being positive may have been a bug; the fix may be to test if it's less than 50 – in which case, it's now valid to test further if the number is negative. In this case, you end up doing twice as much work. Even more challenging, you will need to address bug fixes to dead code (that is, patches to the dead code area), which may replace previously removed code. This results in a cascade effect of changes (and, of course, testing). Over and above this, the very task of analyzing code to see which parts of it are "dead" is itself a non-trivial task.

[4] "*Lines of Code Linux Kernel*" by Stefan Pohl - Own work. Licensed under CC0 via Commons https://commons.wikimedia.org/wiki/File:Lines_of_Code_Linux_Kernel.svg#/media/File:Lines_of_Code_Linux_Kernel.svg

Testing issues

Compounding the problem, the Linux architecture is the exact wrong architecture to try and accommodate the rate of source code change (as shown above). Had Linux adopted a microkernel, modular approach, each component could be developed and maintained in isolation, with final integration testing happening on well-defined boundaries. As it stands, the monolithic approach taken with the Linux kernel means that the entire unit has to be tested as one, very complex and integrated whole. It's readily apparent that testing tens of millions of lines of code (Linux kernel), versus approximately 100 thousand lines of code (QNX kernel and process manager) is a much more significant undertaking.

On the other hand, using the QNX operating system (and libraries, tools, etc.), ensures that you are using a cohesive system where each configuration has been holistically tested and certified. This is enhanced by QNX's modular, microkernel architecture – now there are many small, well-defined and independent pieces to test, rather than one large whole with many interdependent parts. The net effect to you (and thus the TCO of the QNX solution) is that you can focus your test and development effort on just the (much smaller) parts that relate to your area of expertise – your product.

Quality and Certification

The aspects of quality and certification need to be addressed as well. Quality is defined as the degree to which a set of characteristics meets requirements. Certification is the confirmation of characteristics (effectively, a confirmation of the quality), usually by an accredited organization.

Putting this into perspective for software, certification is required for certain classes of devices (e.g., medical devices, safety critical devices, etc.) Various certification bodies are responsible for ensuring that the software meets certification requirements.

At the heart of a safety critical certification, such as IEC 62304, is the concept of traceability. This covers key aspects of the software's development cycle: how it's designed, how the requirements are mapped, how the test procedures map to the requirements, and so on.

Ultimately, it must be shown that the software does what it's designed to do.

Formally, this is achieved by showing that each component (be it a line of code, a function, a module, etc), derives from some kind of requirement, and that the requirement can be (and is) tested. In this manner, a complete traceability matrix is arrived at, which demonstrates that every high level requirement flows into a lower level requirement, and so on, until eventually every operation of the software is accounted for.

This is, as you can imagine, an onerous task.

As part of your design, you will have a requirement for an operating environment (the "operating system" component). This is the base of the platform on which you are building your system. If the operating system that you are using is already certified, then you don't need to do that work yourself – you simply state a high level requirement along the lines of "the project shall use a certified, realtime operating system with the following characteristics" and you list aspects that are critical for your design (e.g., response time, size, supported devices, etc). With QNX, your work is done – QNX is already certified in the areas of medical, industrial, and automotive domains.

As far as the concept of SOUP goes, there are two components to it:

(a) Was the software developed for the given purpose (e.g., being incorporated into a medical device)?

(b) Are adequate records of the development process available?

For software that's not specifically developed for the given purpose, the second component can be used to group the SOUP into two types – clear (for which records exist) and opaque (for which records don't exist).

QNX falls into the clear category, whereas Linux falls into the opaque one. Clear SOUP is much easier to certify, because the hard part (the documentation) is present and available. Due to its very nature, Linux doesn't provide records of the development processes, and is thus considered opaque SOUP. As such, there are no requirements, or traceability matrices, or development plans to back up the design of the software. This means that the onus is on you to provide proof that the software does what it's supposed to do. Given the size of the Linux kernel (tens of millions of lines), and the constant change going on in the source base, this is a daunting task, and adds a significant cost to the total cost of ownership.

That's not to say that you can't use open source software – the certifications don't preclude the use of open source, they just impose traceability requirements. Certain open source software is fine to use, because either the development community has developed it specifically with certification in mind, or someone else has gone through the effort of certifying it.

GPL Concerns

In the second whitepaper of the series, we mentioned that Linux was its own worst enemy for two reasons (the first was version rigidity due to the kernel ABI interlock). The second place where Linux works against itself is on the GPL front. While there is much controversy over the GPL and intellectual property (IP) rights in general, this issue comes to bear exactly on device drivers. One of the terms of the GPL license is that any source code combined with GPL-licensed code also falls under GPL. This means that the source code must be made available on demand, and, more interestingly, limited patent rights need to be granted to anyone using said source code (for example, you can't give away the source code and then claim patent infringement when someone uses it). Hardware manufacturers are reluctant to have their source fall under the terms of the GPL (that is, release the source code, let alone give away any other IP rights they may have). At best, this means that the manufacturers will generally release binary blobs (i.e., "no source") as drivers (and even that's taking a chance as far as the GPL goes). GPLv3 takes things even further with the 'anti-tivo' clauses which require people to not only provide source to software mixing GPL software but to also provide a means to update that open source software. Can you imagine, you could be required to provide not only the source to the kernel you use but a means in which a customer can update the software to a different release? Ouch!

A further consideration of the GPLv3 license is that many projects are changing their license from GPLv2 to GPLv3. You may find that GPLv3 is not acceptable in your project, and thus you may find yourself stranded with the source code version that was last licensed to GPLv2. QT is a good example of this – your alternative (if you can't use GPLv3) is to get a commercial license. But now you're paying for source, which isn't what you started out to do by using "free" software.

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 limited analysis of the cost of obtaining and licensing the source code is insufficient, especially when considering large cost items such as development, maintenance, certification, and the overall software team size and dynamic. 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 the ability for you to get back to what you do best: developing your own software.

In summary, the big-ticket TCO contributors for Linux are:

- Certification of non-core assets
- Development and Post-Deployment Costs
  - Patch & Merge
  - Testing
  - Non-core / collateral development (e.g. drivers)
- IP exposure risk
- You end up being in the operating system business

Linux is indeed free … like a free puppy.