WHITEPAPER

QNX | BlackBerry Subsidiary

# Seven fatal mistakes to avoid when choosing an embedded OS

Malte Mundt
Senior Field Application Engineer

QNX Software Systems

1/11/16

# Seven fatal mistakes to avoid when choosing an embedded OS

## Abstract

*How would you go about choosing an embedded Operating System (OS) for the next generation of your product? As long as you do not have to actually do it, you may think that the choice should be based on the objective evaluation of various criteria, such as features, cost, support options, etc. In reality, however, there are seven completely different aspects that push executives, directors, managers and engineers towards fatal traps. Just like quicksand, you do not notice that you stepped into one of them until you get into trouble. Typically, you realize this towards the end of your project, in the middle of your product's lifecycle, or at the beginning of the next project when trying to add advanced features. Thus, in this whitepaper we will make you aware of those pitfalls and shallows you ought to avoid when choosing an embedded OS, without drifting off into a lot of technical details.*

## Mistake #1 – Not choosing at all

Your reading this means that you won't make the worst mistake of them all: not *consciously* choosing your embedded OS. However, often decision makers are so consumed with their application that they do not give much thought to making a conscious OS decision. They assume that what is shipped with the hardware is probably "good enough". This will impact the success of your company more than expected, because as soon as you start development on a particular OS, it is the beginning of building a stack of software that will later become *legacy* – either yours to handle five to seven years from now, or your successor's.

The sad aspect about this is that managers making this mistake will never know how their life could have been different. So what's the issue with a general-purpose OS like Windows, as long as it does what it is supposed to do? The problems start when it doesn't, and you quickly find out how limited your control over such an OS is. Fixing problems that should never have occurred in the first place costs a lot of money. We will discuss why making a conscious OS decision will largely reduce the risk of increased project costs and missed deadlines.

## Mistake #2 – Doing what "all the others" do

It is deeply embedded – pardon the pun – in the human psyche: If not sure, just do what everyone else does, and you will be fine. Eons ago, this was important for the survival of our species: using the tried and tested paths could save your life.

In today's modern world however, constant change is part of our lives, and innovation is at the heart of the embedded industry. Hence, when it comes to deciding for an embedded OS, it can be very dangerous just to rely on what people do on the left and the right. Remember the city of Pompeii that was completely destroyed by a volcano eruption? Why did the people of Pompeii build their city so close to a volcano? Well, someone thought it was a nice place to build a house, a few others followed, and for everyone coming afterwards the decision was easy: "Hmmm, the volcano is close, but hey, it must be okay, all the others have built their homes here, as well." We can even imagine that someone choosing to settle further away possibly might have been subject to remarks along the lines of being too anxious or overcautious. And at

this point, another strong mechanism in the psychological decision-making process kicks in: Better stay in line, don't be too *different.*

Not much has changed regarding the way people make decisions – only those who overcome those ancient mechanisms when choosing their Embedded OS can prevail. Do not allow yourself to get distracted by statements like "in this company, we always use X," or "the hardware vendor recommends Y," or "at my university my professor said Z is the best choice." You, not them, will be held responsible for all kinds of issues. After the destruction of Pompeii, the general response to the disaster was: "It was fate." In embedded, when projects fail, it's usually accounted to "mismanagement", "overly aggressive timelines", "failed outsourcing partners" – you name it. What usually doesn't come to mind is that a lot of time (during development, testing, and problem fixing) could have been saved by leveraging a proper embedded OS and tool set.

Make your own decision, based on real, current, and future needs – and risks. Be prepared to be frowned upon when you suggest an OS that has not been deployed yet inside your department or company. Remember: companies in low-cost countries often compete by employing very large development teams on a low budget; companies in high-cost countries have to stay focused on innovation – doing things in a different, smarter way.

## Mistake #3 - Continue using the OS you have "always" used

This is a tough one. You have been in the business for years, you have successfully completed a few projects, and your company is doing well. So why not just go on with the OS that is currently used in-house?

The safest route often seems to be the one we have taken in the past. This is why marketing experts say that with growing age, it becomes increasingly difficult to convince campaign target audiences to switch brands. Research shows that over time, people tend to become more and more rigid in what products they prefer. With increasing age, the desire to avoid risks sometimes grows so immensely that *everything* new is avoided.

Decision makers selecting an embedded OS must be aware that they likely face a resistance to change. To find out if you are affected, verify what you think about some of the newer inventions, e.g. do you think social networks are cool, or are you trying to stay as far away as possible? Do you prefer printed books or do you own an eBook reader? Are you interested in electric cars or do you think they don't make any sense? We are not suggesting one or the other, and of course not everything new is great and should be embraced automatically. However, these and similar things can serve as a *gauge* to measure personal animosity to new things in general, and this way, you can improve your level of objectivity. Changing the OS from an existing solution to a new, "unknown" one involves a significant investment. On the other hand, staying with what you have can be extremely limiting, and in many cases can even be the higher cost option overall.

A smaller company, maybe even a startup, may just be reinventing your device, without the burden of legacy concepts. Embedded systems are becoming very complex, and it is likely that an OS strategy that was defined years ago cannot be applied any more. Remember: Standing still means losing ground.

## Mistake #4 – Avoiding spending money at all cost

When software is being made available free of charge, why does no one usually ask where it comes from, who programmed it, or what the motivation behind offering it for free really is? It is human nature to take advantage of free offerings: Who would refuse a free, cold beer, even if it's of questionable quality? "Heck, it's free!" While consumers are slowly learning that free software and services come at a price not measured in dollars (e.g., you have to accept ads, you are willing to be tracked), this does not seem to apply for *free* operating systems like Linux. So where are the downsides there?

Increasing cost pressure and wide availability drive some decision makers towards free OS solutions such as Linux, which is usually chosen in good faith. The perilous gut feeling is: "Surely, it is going to work fine. If not, we will fix it, and this fixing will cost us much less than obtaining a commercial solution." Surprisingly, although embedded Linux is *free*, there are many "OS experts" making money with Linux – draw your own conclusions. Support for Linux is also provided through a "community" – large at first sight, however the embedded portion is not so large. For instance, the Linux "Real Time patch", which is required to make it at least a little more deterministic, is maintained by a very small number of people. Given that they could decide to stop their activities (and in fact, this has already been considered several times), the future of this package is questionable. It becomes clear very quickly that a seemingly *free* OS solution just moves the cost elsewhere, into areas difficult to estimate, measure and track.

In our youth, every one of us had idols. We wanted to be like them, because they were very successful and popular; they had something that made them outstandingly different from the average. Usually, the longing for idols disappears when we have grown up. This is somewhat unfortunate, as it can be very worthwhile to find out what those who are *extremely* successful did to get there, to learn from them. Let's look at some of the largest and most successful corporations – car makers, for example. Mercedes-Benz, Ford, and Audi deploy the QNX embedded OS for their on-board infotainment and instrument cluster systems. And that is surely not just because they can afford it: the choice of the embedded OS for devices installed in millions of cars was certainly not taken lightly. So why do car makers go for a commercial OS like QNX instead of a Linux variant?


***Figure 1 - In multi-million dollar projects, the choice of the embedded OS is not taken lightly.***

- Hard deadlines: When a new car model is being announced, the software has to be finished in time. Delays of weeks, or even months, to fix major issues, are not an option.

- Rock-solid technology: A commercial OS like QNX was not developed just for fun, it was, from the ground up, *designed* to be deployed in devices that *must* be stable. Car

vendors cannot afford negative customer experience and publicity when their in-vehicle systems run amok.

- Clear accountability: In those multi-million dollar projects it needs to be absolutely clear who is responsible for each component, including the OS, both from a technical and legal perspective. This is very difficult with “free” Open Source software.

- First-class support: Many little, and some bigger, companies offer consulting services for Linux, but only a vendor that has written their OS themselves can offer the timely and high-quality support that is needed to be successful.

Give commercial embedded OS offerings a try – they exist for good reasons. As with most, if not all things in life, the same adage applies, “You get what you pay for.”

## Mistake #5 - Going with a desktop OS instead of an embedded OS

In 2010, Stuxnet hit the factories, due to vulnerabilities in the Windows OS. Since 2014, security holes in Windows XP are no longer being fixed. Both of these had huge cost impacts for device makers and should never have been a problem. Those two major disasters made it perfectly clear: an OS coming from a desktop context is the wrong choice for industrial control, medical devices or building automation applications. There is a strong need for a purpose-built embedded OS, one designed with security in mind, one that can be deployed for decades.

But why was a desktop OS deployed in embedded applications in the first place? In the past, if one of the system's main requirements was advanced graphical representation of data (visualization), Windows was selected because it was often seen as an OS with a strong focus on graphics, while an embedded OS was in a *black box*, in charge of control tasks. Over the last few years though, two major factors have changed:

- Embedded operating systems are no longer as limited in graphical features as they used to be. Modern offerings like the QNX OS even provide smartphone-grade user experience, dismissing the need for Windows-based systems for visualization.

- Security issues in desktop (and mobile) OSs have been on the rise. In today’s totally networked world, basing your device on an OS requiring multiple security patches every month poses a high risk, and great expense to both the vendor and the end-customer.

The big difference is that a desktop OS was designed for *users*, while an embedded OS was designed for *developers*, for being at the heart of your device. Typically, the corresponding vendors care about what you are doing with it, and can participate in the development with you, by providing consulting and engineering services, which allows you to draw from an expert knowledge pool. And if existing software is needed in your project and cannot be easily

***Figure 2 - Attacks on embedded systems have been on the rise, and security patches are not the answer.***

ported, a *hypervisor* allows your legacy OS to run together with your new one, side by side and makes sure that mission critical tasks keep running even when Windows has to reboot.

## Mistake #6 – Trying to find the fastest OS

***Figure 3 - Fast cars are fun, but are they suitable for everyday tasks?***

Those who are trying to make a conscious OS choice are clearly ahead of those that are not, however the challenge is finding the right criteria for your decision. Thus, it is obvious to jump at some that are easily measurable. Memory usage is no longer as critical today, but speed, or *performance*, is. Many dream of driving a Ferrari, but, if we could afford it, we might quickly find out that it doesn't carry our shopping bags too well, and is very expensive to maintain. So, how does this issue impact an Embedded OS?

Typically, software engineers tasked with *benchmarking* an embedded OS are looking at, e.g.:

- how fast can data be sent over the network
- how many complex mathematical computations can be done in a given time
- how much time it takes to react to an event (latency)
- speed of inter-process communication

The usual way of doing those kinds of tests is writing little programs that exercise the various OS functionality *constantly*, in a number of variations, for a certain time. The big problem with these measurements is that often they don't have a lot in common with how the OS functionality is used in real applications. The OS gives you, for instance, the possibility of having software components in your system communicate with each other. Are they communicating constantly, or even most of the time? Or are they doing what they have been designed to do, and communicate only now and then?

Let's take a look at a standard networking benchmark that constantly sends (and/or receives) data. There are some important discoveries to be made beyond the data throughput:

To keep software components cleanly isolated from each other, an OS that is architected for safety will treat the benchmarking program as an encapsulated *process*, and the network stack as another process. With this approach, networking operation can take a bit more time, as the data the benchmarking program wants to send needs to be moved (copied) over to the network process by the OS core (kernel). In an OS designed for maximum performance however, the network stack is *linked* into the OS kernel, and so is allowed to intrude into the memory space of the benchmarking program, and snatch the data to be sent directly from there. While this is

faster, the big downside is that problems caused by faulty networking packets or programming errors can compromise the whole OS kernel.

The question is: would the difference in speed actually be noticeable in the real application? What percentage of time does your software really spend inside OS functions? There is no easy answer, unless you conduct an in-depth analysis. Experience, however, shows that it in many cases it is a small percentage. Ideally you *know* what speed you actually need, but even if not, one thing is for sure: The *maximum* is not the right number to look for, because as illustrated above, OS speed always comes at a sacrifice: less stability, and more complexity, leading to reduced safety and security for your product.

Experience has shown that a clean application design is the driving factor behind a well performing system. No speedy OS can counterbalance wrong application design decisions but a good OS can make your software more reliable, secure, and easier to maintain.

The implication is that the speed of your hardware is much more important than the speed of your OS. The hardware runs the OS but more importantly, it runs your applications, which will consume the lion's share of hardware resources. Selecting a hardware platform with a bit of *headroom* might cost a little more, but will save you precious development time, help avoiding performance headaches, and leave power for future functionality.

## Mistake #7 – Not planning ahead, aka the "we don't need" syndrome

When looking at an embedded OS, people often come across QNX, and find that it offers:

- very sophisticated real-time capabilities for deterministic behavior
- versions with special safety certifications for industrial, automotive, and medical
- isolation of all software components from each other, and from the kernel, for maximum reliability of the system, including the components you add, or buy from 3rd parties
- a comprehensive suite of development tools that allow in-depth system profiling and analysis

For the less experienced decision makers, the list reads like a lot of nice to haves - in other words, features that are perceived as not really *needed*. A key feature often mentioned in the context of embedded operating systems is "real time". Contrary to popular belief, this term doesn't mean "real fast", instead it stands for being capable of providing deterministic behavior: The system does what it was designed to do, always within the appropriate timeframe. For example, pressing a key shall lead to a certain reaction in a certain time. Sometimes when typing something in the address field of the web browser, it takes a moment to appear. This moment may be brief but can be deadly if it's about cutting the fuel to a malfunctioning jet engine in an emergency. The crucial difference on the OS level is that some OSs are *built* to be real time capable, while others have been "extended" (*hacked*) to provide a degree of real time capability. Both call themselves "real time capable", however the difference in reaction time can be huge. But even if you could tolerate this in your device, never say you "don't *need* a real time OS". A true real time OS is deployed in mission-critical devices, where low reaction time is required, and if not met, can mean catastrophic failure, injury of people, environmental hazards, or production of damaged goods. Beyond that, in those systems other appealing properties are required as well: stability, reliability, availability, certifiability, and maintainability over at least a decade, and much more. And we are sure that you will *need at least* one of those, if not several.

That's why an embedded OS designed for real time is a good thing to look for – designed for real time means designed for the most demanding application scenarios.

All of us try to make our lives simple by comparing and matching unknown things with things we do know. When buying a car, everyone knows that they usually have four wheels, an engine, etc., and all of them will bring you from A to B and the crucial point: all of them will do this *successfully* in a very similar timeframe. An embedded OS is not a car though. An embedded OS, the way many think of it, contains a task scheduler, drivers for various hardware components, synchronization mechanisms, "whatever". The big difference, however, is the definition of *successful:*


***Figure 4 - They may have thought "We don't need a real-time OS", but a more reliable OS would have been a good choice.***

- completion of your project in a given timeframe and budget
- customer satisfaction, low warranty costs
- a scalable solution for the next product

Depending on the embedded OS selected, the cost and timeframe needed to get to your goals can vary greatly. That's because the advantages of a well-designed embedded OS are not like add-ons to a car (e.g., park distance control), but instead they are like the core measures of safety – think seat belts, airbags, crush-zones, etc. We hope you never really *need* one of those, but would you ever drive without them?

## Conclusion

We have discussed why you should choose your embedded OS consciously, based on your own analysis and current project needs, but also possible future requirements. You are now more aware that a change from the seemingly proven path of the past may be necessary, and that it is unlikely that this will come for free. And finally, you learned that speed is not everything, but additional OS features supporting reliability and security absolutely are. Even if not all of them are *needed* at the moment, they will serve as a great insurance for future challenges.

If you are interested in learning more about the selection criteria for an embedded OS, we recommend reading the following:

*Choosing an RTOS for Remote-care Medical Devices*, by Justin Moon et al.: Deep-dive into aspects to watch out for when choosing an embedded OS, from a technical point of view. As you can imagine, for medical devices the OS must meet very high standards – so if you are

building something less life-critical, there's no way to go wrong when using an OS proven in medical devices like eye laser control or blood pump systems.

http://www.qnx.com/download/feature.html?programid=22012

*The Joy of Scheduling*, by Jeff Schaffer and Steve Reid: The scheduler is at the heart of the operating system, it governs when everything runs — system services, applications, and so on. If the designer doesn't have complete control of scheduling, unpredictable and unwanted system behavior can and will occur.

http://www.qnx.com/download/feature.html?programid=21959

*An Introduction to QNX Transparent Distributed Processing*, by Yi Zheng: Everyone knows that files and data can be easily shared in a network. However, a sophisticated embedded OS like QNX enables sharing of any hardware resources in a network, without complex software programming. This brings major advantages such as excellent scalability and a huge cost saving potential.

http://www.qnx.com/download/feature.html?programid=22908

## 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**