MOVING FROM AN RTOS TO LINUX? (PRACTICAL INSIGHTS NOBODY'S TELLING YOU)

KATHY TUFTO, SENIOR PRODUCT MANAGER

Mentor®
A Siemens Business

E M B E D D E D   S Y S T E M S

W H I T E P A P E R

INTRODUCTION

Many challenges arise when moving to embedded Linux® from a real-time operating system (RTOS). Obviously there are trade-offs between what you get with an RTOS and what you get with open source software (OSS). Factors such as scalability and performance must be taken into consideration. Also, what if you want to reuse your existing IP without turning your IP into open source software?

Open source software is often perceived to be free, but if you're not careful unintended costs can quickly add up. There's a great deal of effort that goes into managing a Linux distribution. However, a successful migration from an RTOS to embedded Linux can bring numerous advantages to an organization. Top advantages include reduced licensing costs and greater control over maintainability.

There is much written about considerations of moving from RTOS to Linux for embedded projects. Based on pragmatic experience of helping customers through the decision-making process and the actual transition, this paper provides practical information, so developers can be fully aware of the trade-offs of moving to OSS and the often unmentioned hidden costs of managing a Linux distribution.

WHEN IT MAKES SENSE TO STAY WITH AN RTOS

There are many advantages to moving from an RTOS to open source. However, in some situations it might make sense to stay with an RTOS. A few of the design requirements that might influence your decision include:

- What are the real-time requirements? Just how *real* is real time?
- Is there a specific licensing agreement you must consider or follow?
- Do any of your devices include safety-critical functionality which might require deterministic performance? Can your current operating system be safety-certified? If the device does need to be safety-certified, then to what level?
- How important is it for you to reuse current or existing IP?
- How much memory do you have? How much space can you allocate for the OS? Memory footprint is without question, a key consideration.
- What about support for a multicore processor? Traditionally, RTOSes have only supported single-core, but this is no longer the case as more RTOSes run on multicore.
- What development tools are available with the current operating system? How easy will it be to procure and learn the necessary tools for a Linux deployment?
- What is the cost of development?
- Who's going to provide support should you need it?

These are just a few of the factors that come into play when considering a move from an RTOS to Linux. As you can see, a great deal of forethought and pre-planning is required.

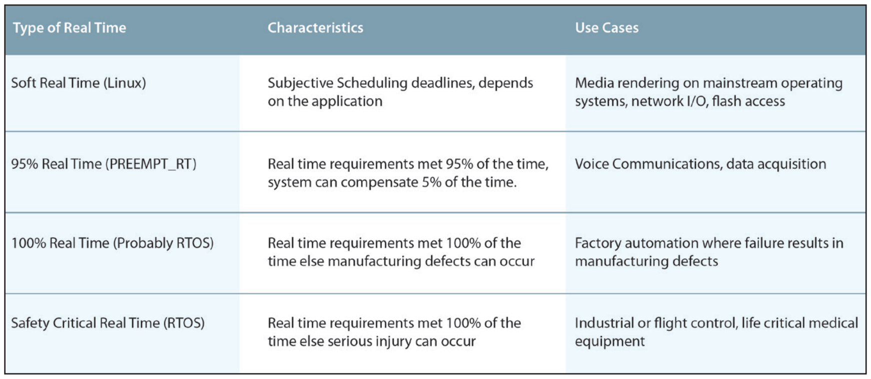
A LOOK AT THE HARDWARE

Performance requirements are critical in the decision as to whether a move to Linux. One of the first points to consider is whether the intended design will fit. Oftentimes, Linux developers will say, "If you want to run Linux, you'll need 128 MB RAM and 5GB for memory." This is not always true because when building a Linux application, you can make it as big or as small as you want. With Yocto™ Project-based Linux, for example, it is quite easy to scale it down to a fully functional OS at 64 MB using 512 megabytes of RAM. You can even have it smaller. But if you start to scale it down to say 16 MB, it's just not going to have a lot of remaining functionality.

So before any decision is made, it's highly recommended that you take a look at your end application and determine whether or not Linux is a good fit. Maybe an RTOS is more appropriate. It's not rocket science to scale Linux, but again, as you go smaller, it becomes more challenging to scale it down with the required functionality.

WHAT ARE THE REAL-TIME REQUIREMENTS?

The question is how *fast* is your real-time? Figure 1 breaks down various types of real-time. What does real-time mean anyway? It shouldn't be any surprise to learn that Linux is used in systems that have real-time requirements. There's a class of system called "soft real-time" where basically, you have subjective deadlines. That is to say, if you miss a deadline here or there, it's not going to cripple functionality. Nothing's going to break such as updating the UI or connecting to a non-critical network, etc.



Type of Real Time	Characteristics	Use Cases
Soft Real Time (Linux)	Subjective Scheduling deadlines, depends on the application	Media rendering on mainstream operating systems, network I/O, flash access
95% Real Time (PREEMPT_RT)	Real time requirements met 95% of the time, system can compensate 5% of the time.	Voice Communications, data acquisition
100% Real Time (Probably RTOS)	Real time requirements met 100% of the time else manufacturing defects can occur	Factory automation where failure results in manufacturing defects
Safety Critical Real Time (RTOS)	Real time requirements met 100% of the time else serious injury can occur	Industrial or flight control, life critical medical equipment

Figure 1: *Embedded industry's classification of real time.*

When you look at the Linux kernel itself, it has about 80 percent of the real-time patches. So Linux is already equipped to handle soft real-time requirements. But what if 80 percent isn't good enough? Say you need your requirements to be met 95 percent of the time with something like a voice communications device or a data acquisition device. This is where you may need something a little bit more than just off-the-shelf Linux, and so you might do something like enable the PREEMPT_RT patch. The PREEMPT_RT patch makes interrupts run as threads. You have the ability to modify priorities of the interrupts and have user space tasks run at an even higher priority than interrupts. Having interrupts run as threads allows you to prioritize the interrupt handlers even when the hardware does not support it. If you apply PREEMPT_RT it's possible to decrease your Interrupt Latency to 10us.

What if you need to be 100 percent real-time 100 percent of the time? An example of this might be a tool used in a fabrication plant that's outputting chips, and if you have a defect in the chip, you stand to lose production time and cost. It's questionable whether you might use Linux in this type of situation. It depends on how fast is real time? How much latency is allowed?

MEETING SAFETY-CRITICAL, REAL-TIME REQUIREMENTS

Safety-critical, real-time performance might also be a consideration. In this scenario, requirements must be met 100 percent of the time or there could be risk of bodily injury. For example, if you're in an industrial setting and you have a robotic device equipped with a laser-based optical safety gate and if somebody crosses that laser, the robot needs to shut off or there will be potentially dire consequences. Or say you have a medical device that, again, if it fails, the user is placed in serious jeopardy. In these types of settings, Linux is probably not your best option.

By default, the Linux kernel is not deterministic or pre-emptible. PREEMPT_RT adds preemption to the Linux kernel. It basically makes your interrupts run as threads in user land, and then you can actually set priorities and do the types of things you want to be able to do in a real-time system. When you apply the PREEMPT_RT patches to Linux, it's going to decrease your worst-case interrupt latency and make your system more deterministic. But this comes at a cost. It adds thousands of lines of code to the kernel source code. The PREEMPT_RT patch is outside the mainline Linux kernel, so you now have to take these thousands of lines of code and maintain them for the life of your product. There's a cost in doing this as well. But it may be worth it, if you need this type of performance.

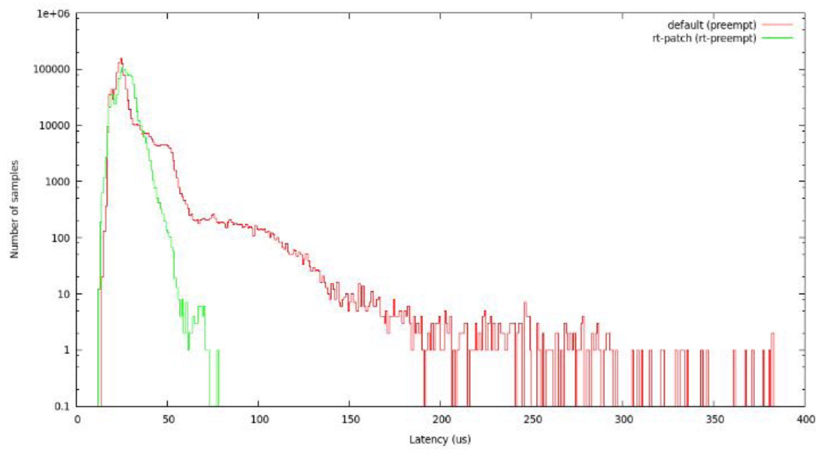


Figure 2: *The histogram shows latency comparison between the default Linux kernel (in Red) and the kernel with preempt_RT patch (green) on a Raspberry Pi. (Source: Emlid.com).*

As you can see in Figure 2, the PREEMPT_RT patch provides reduced latency over the standard Linux kernel. Of course, an RTOS can provide even more reduced latency, but depending on your real-time requirements, you may be able to meet your performance needs with Linux. The take away here is if you can live with higher interrupt latencies, Linux might meet your requirements in a system.

OTHER PERFORMANCE ISSUES: HOW FAST DOES YOUR SYSTEM NEED TO BOOT?

A myriad of activities happen in boot, which means there are plenty of areas developers can optimize. In most situations, developers are able to boot an optimized embedded Linux in less than three seconds. Boot time depends on what you're measuring, and what needs to be turned on at the end of that time. Most Linux developers can reduce boot time by 15 to 25 seconds. This type of optimization is typically not that difficult to accomplish. Of course, if you have a headless Linux it gets even easier because the overhead to initialize the graphics system and render graphics to a display is relative to just booting the kernel and applications. But if you need to have Linux booting in less than 500 milliseconds that would be a significant challenge. It is possible to optimize Linux to meet those goals. Generally speaking however, if you want something that fast an RTOS is probably going to perform a little bit better.

CONSIDERATIONS FOR REUSING EXISTING IP

Once you've decided that Linux is going to meet your application's performance requirements, the next step is to leverage your existing IP. A majority of operating systems today comply with POSIX. The good news is Linux is actually POSIX compliant, but not conformant, which means if you're porting IP from a POSIX environment you might be surprised how well it works and runs with very little effort.

Further, Linux might have an API that looks the same as the API from your RTOS, but what the API actually does from an OS perspective might be quite different. Therefore, you must validate that everything is working as expected. The key here is to be able to port some of the APIs so you can reuse more of your application without having to rewrite it.

The bigger question, however, might be how do you do this without turning your IP into open source?

PROTECTING YOUR PROPRIETARY IP FROM OPEN SOURCE

It’s always a good idea to consult with your legal team when considering a move from an RTOS to open source. And what exactly does it mean for software to be defined as open source? Well, your Linux design must be freely redistributable, and the source code must be made available. It must allow other Linux users to derive code from your Linux implementation. The key point to remember here is that when you derive something from it, depending on the license type, it may require you to license your code under an open source license, depending on how you're doing things. So this process can be tricky if you’re not absolutely sure how open source works (Figure 3). When working in open source you cannot discriminate against people, fields, applications, etc. You must have the right to redistribute the license, so on and so forth.

For more details on these matters, it is highly recommended that you visit opensource.org.

LICENSE TYPES

What's perhaps most interesting to note are the different classifications of open source licenses. There are **permissive licenses** where you can make modifications to things that are using these types of licenses while still keeping your IP proprietary. Permissive licensing gives you more flexibility to do different things. Examples include the BSD license and the Apache license.

There are also **restrictive licenses** which are often referred to as "copy left" or "reciprocal" licenses. This is a very different category of licensing. Restrictive licenses require that when you make changes or enhancements to something that's licensed under one of these agreements, you are required to share these changes with the Linux community. Restrictive licensing requires that you contribute back to the community.

Depending on the license type, there are restrictions around what you can do and how you would link the open source code from your code in order to keep your code from becoming open source. It’s highly recommended that you work with your company’s lawyers to determine which types of licenses are allowed in your applications. You can then use open source tools to verify that you are following your own rules.

It's important to have a legal department that understands open source and can work with you to decide which types of licenses are allowed in your system. You do not want to unintentionally turn your IP into something you have to give back to the community. There are tools available to help you prevent this from happening. Fossology.org will do a scan so you have your distribution with your application. It basically performs a scan and says here are all the licenses that you're using, or here's what everything is going to end up being licensed as.

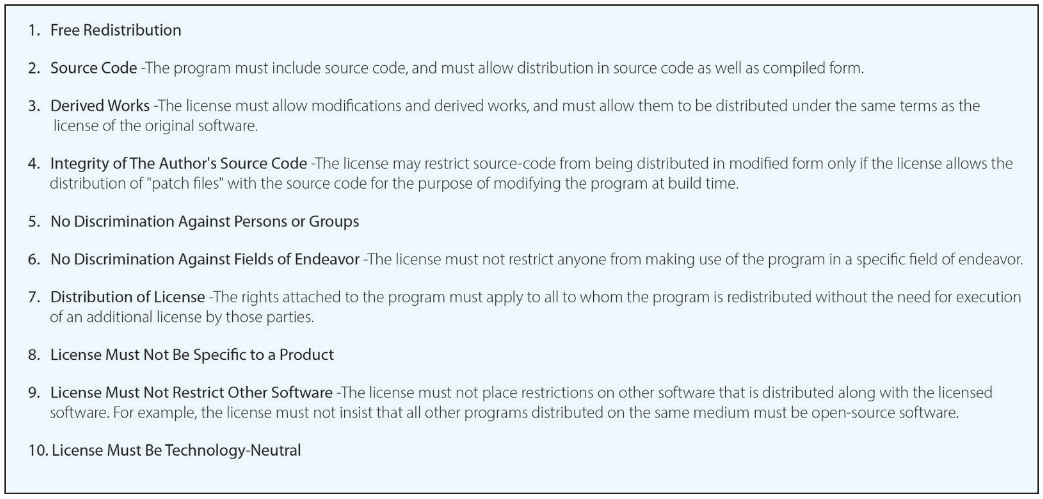
FlexNet is another commercial tool that might be able to assist you along these lines.

More about boot time

The more you're doing at boot, theoretically, the slower the boot-up process is going to be. And then, of course, you have performance user interface (UI) issues. The first thing a user sees when a device starts up is how long it takes for the UI to appear. If a user clicks the start button and it takes two minutes before anything appears, you're probably going to have a very dissatisfied user.

Boot time and application performance are key when it comes to keeping the end user satisfied. One of the challenges for any system is validating that system requirements have been met. Most commercial real-time operating systems are available with tools that provide the ability for developers to look at the performance of the operating system. When you move to embedded Linux, there are open source tools that provide the very same functionality. **Linux Trace Toolkit next generation** (LTTng) is an open source software toolkit which you can use to simultaneously trace the Linux kernel, user applications, and user libraries.

Further, open source visualization tools allows you to visually look at that data inside the system, and then write unique analysis routines to perform activities such as CPU utilization or performance during boot.



1. Free Redistribution
2. Source Code -The program must include source code, and must allow distribution in source code as well as compiled form.
3. Derived Works -The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
4. Integrity of The Author's Source Code -The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time.
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavor -The license must not restrict anyone from making use of the program in a specific field of endeavor.
7. Distribution of License -The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
8. License Must Not Be Specific to a Product
9. License Must Not Restrict Other Software -The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.
10. License Must Be Technology-Neutral

Figure 3: *An open source software license must meet this definition to be considered OSS.*

MANAGING A LINUX DISTRUBUTION

One of the biggest challenges of moving to an open source operating system is managing it. You need to decide: which distribution you will use; which tools you will use to build, customize, validate and deploy your OS; and what processes your developers must follow. At the beginning of your project, make sure your legal team understands what open source is and is not. Your legal team might be surprised to learn that not everything you're planning to build is going to be your company's IP. You need to understand which types of licenses your legal team will allow you to use in your product so you don't end up redoing things at the end of your project.

Another key consideration is how to support your Linux OS for the full life cycle of the product. These are things that are taken for granted with most proprietary OSes, including commercial RTOSes, because commercial offerings typically include tools, documentation, and support. Support often includes software updates, patches, "how-to" help, and answers to general or specific usage questions.

One of the first questions you might ask is "Which tools am I going to use for my Linux development?" Are you are going to need a toolchain? Are you going to need a way to connect to your target and debug your applications? Will you need a way to validate to ensure your performance requirements are satisfied? There are many options along these lines including Buildroot, The Yocto Project, and Debian Linux. One of the more popular choices is The Yocto Project, which provides developers with a build system, application development tools, as well as a Linux distribution. It's considered relatively easy to customize. Many of the commercial Linux distributions, including Mentor® Embedded Linux, are based on The Yocto Project build.

The takeaway here is when you move to embedded Linux, you'll need to know the tools you're going to use. Not only that, but will these tools be available at the time you plan to transition, build, deploy, and maintain your Linux product?

CUSTOMIZING YOUR LINUX BUILD

Once you decide on tools, you're going to need to customize your Linux OS. So you need to ask the question, "How am I going to manage and support my solution?" Much of this depends on whether you're an individual developer or part of a larger team or company. If you're going to do this yourself, businesses typically discover that using open source tools that are freely available are in practice not truly free. Customizing Linux is going to involve managing millions of lines of code, development tools, toolchains, and more, which will require the real costs of a dedicated team whose responsibility is to manage all this "free" technology.

GAINING COMMUNITY SUPPORT

It's important to realize that if you're the sole developer, you might get community support, but not always. If the community decides that what you're doing has far-reaching implications they will help manage and build your code. They will also help fix any bugs or problems. But if your project is an not aligned with the core direction of the OSS technology as determined by the project maintainer, you might not get a lot of help from the community. It also means, and this is extremely important, that ten years down the road, when you realize there's a bug in your system, you're probably on your own trying to troubleshoot and fix it.

The community will typically support a kernel for one to two years. Beginning the third year, you're pretty much on your own. It is essential that you have a plan in place to maintain your Linux build for the entire life cycle of your product. It's imperative that you have a system in place to test your Linux distribution. The question now is more about how do you plan to support your Linux for the next ten years (at least). Further, how are you going to keep your developers from going rogue and creating 100 different branches? Once your Linux build and associated tool chains are branched, it can be extremely difficult to merge it back into one.

IP REUSE

Even after you've ported your RTOS applications to Linux, you still need to consider your strategy for reusing your Linux IP in your Linux-based product line and beyond. If you're not careful, these costs can escalate quickly – especially if you're doing this on your own. It's easy to end up with hundreds, even thousands of branches.

DEVELOPMENT HOST

Don't assume that everybody who's been working in RTOS development knows the Linux development environment. Make sure you have an internal support plan in place for your development team.

SECURITY ISSUES AND RISKS

How are you going to respond to and who's going to monitor the community for security vulnerabilities?

Will your Linux device become a security risk? If it's a connected device for the Internet of Things (IoT) or simply connected to a local VPN, or Wi-Fi network, it has the potential for a security breach of some kind. In the past, if you were using a commercial RTOS and a critical bug was discovered, the company would notify you and send you a patch or fix. If you're using free open source Linux, nobody is looking out for your best interests.

GRAPHICS

One of the primary reasons developers are transitioning to Linux is because of its extensive graphics capabilities. Graphical multimedia support is a strength of Linux. There are many options available including the Qt® framework, Crank, HTML5, and others. Graphics packages are typically not enabled by default, so you need to configure your build to enable them to work. Further, you may need optimizations in order to achieve your specific performance requirements because when you enable these packages they are not optimized for a specific application out of the box.

Often, when working with Linux, graphics developers will develop in multi-OS environments. Linux is designed and operates under the premise that it owns and manages all of the available hardware on a board. For example, if your hardware has a GPU, and you're running Linux on one core and an RTOS on another core, and you're enabling graphics on Linux, Linux is going to want to steal that GPU and not allow the RTOS application to use it. There are solutions available which will enable Linux to share the GPU, but it's something to keep in mind as you plan your transition.

SECURITY

What are the differences in Linux security when compared to an RTOS? Security is all about risk. What could go wrong if the device is breached? You can spend a lot of time and money on security, but what you need to think about is what are the security risks for your devices and then, based on that risk assessment, devise a specific plan for the security architecture of your device.

By its nature, when compared to Linux, an RTOS is feature-constrained and comprised of a fraction of the number of lines of code. Further, an RTOS is developed by a small number of developers, generally not from a globally distributed open source community. Therefore, security for an RTOS is more easily controlled and maintained when compared to Linux.

With Linux, the best deterrent against security threats is to configure *everything to be secure*. How is this accomplished? Essentially, you need to develop with a secure boot strategy which means you want to validate what boots is exactly what should be booting and that nothing's been changed or altered. Developers must also enable available security at the hardware level. You also might want to disable fuses and the JTAG core, so nefarious types can't just open up the box and connect to your system and hack it.

With Linux, one of the keys to security is protecting data. A brief checklist includes:

- Implement a secure boot mechanism
- Disable all unnecessary communication interfaces and ports
- Eliminate all non-essential services and software
- Periodic auditing of installed software
- Monitor and install software updates for the system regularly
- Two-factor authentication for accessing the system
- Implement mandatory access controls (MAC) and auditing: SELinux or similar MAC system with access control lists and regular review of audit logs

Once a Linux device is deployed, developers need to have a plan in place to address vulnerabilities or threats. If your Linux framework will be used in safety-certified devices, all pertinent guidelines must be followed.

Security vulnerabilities

As embedded devices are increasingly more connected, having a process in place to manage and address security vulnerabilities for the full life cycle of your product is becoming extremely important.

The recent **Meltdown** and **Spectre** security bugs exposed Linux in nearly all x86-based devices and many Arm®-based devices. Luckily there are software patches for Meltdown and Spectre for newer Linux kernels, but back porting these patches to older kernels is significant work and the community will decide if they are going to do this work or it they will leave it to software providers and customers to do the work themselves.

Meltdown breaks the most fundamental isolation between user applications and the operating system. This attack allows a program to access the memory, and thus, also the secrets, of other programs and the operating system. If a computer has a vulnerable processor and runs an unpatched operating system, it is not safe to work with sensitive information. This applies both to personal computers as well as the cloud infrastructure.

Spectre breaks the isolation between different applications. It allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets. In fact, the safety checks of said best practices actually increase the attack surface and may make applications more susceptible to Spectre.

For additional information on Meltdown and Spectre, please visit: meltdownattack.com

INDUSTRY SAFETY STANDARDS

There are many safety standards to follow these days. For instance, IEC 61508 is the functional safety standard for industrial automation. Another popular standard is IEC 62304 for safety-certified medical devices.

The IEC 62304 standard is a type of classification based on the risk of death or serious injury in a given system. For example, if you have a system where death or serious injury is possible and the system fails, it would be classified as a Class C system.

Normally, you will not find Linux operating in these types of environments, at least not in a safety function. A Class B system is where non-serious injury is possible. For example, a digital x-ray machine. For these types of applications embedded Linux as an ideal match. If the system fails, in the worst case scenario, a patient might receive a little extra dose of radiation, but the malfunction is not life threatening. Other common examples include some types of infusion pumps. With Class A, no injury or damage to health is possible. An example might include a tablet which acts as a bedside monitor, displaying information but does not directly expose the patient to any type of harm.

For more detailed information on medical device certification, see this industry white paper by Mentor Chief Safety Officer, Robert Bates.

MENTOR EMBEDDED LINUX

Throughout this paper, embedded Linux has been referenced. As far as a commercial quality Linux solution is concerned, Mentor® Embedded Linux is one of the industry's top embedded Linux solutions (Figure 4). Based on the Yocto Project, Mentor Embedded Linux provides developers with a very smooth out-of-box user experience. Instead of having to grab various bits and bytes, software developers receive an installer from Mentor with easy set-up. Further, Mentor supports a broad range of commercial hardware and can support your custom hardware via its dedicated services department.

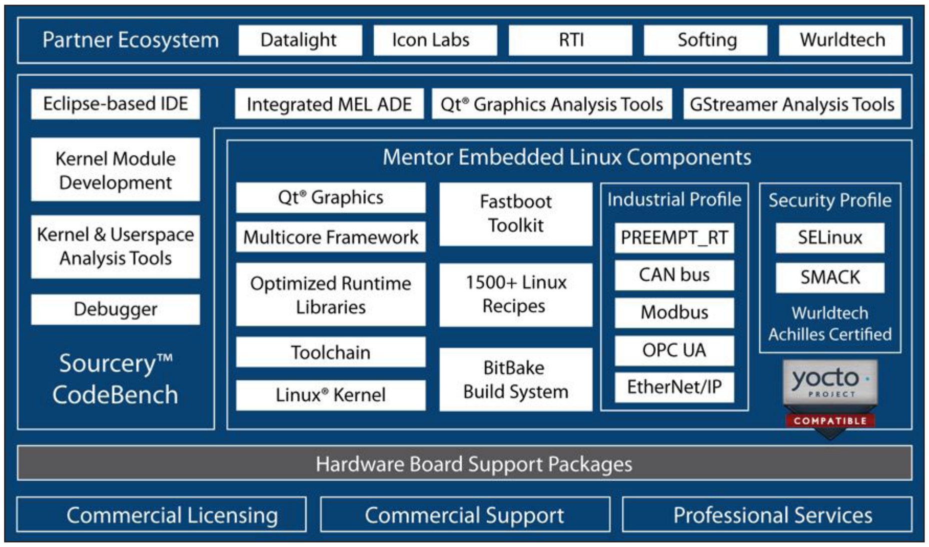


Figure 4 *Mentor's commercially available out-of-the box Linux platform and ecosystem*

Mentor Embedded Linux highlights:

- Yocto™ Project-based Linux platform
- Broad hardware support: Arm®, AMD x86, Intel x86, and PowerPC
- Customizable to meet specific product requirements
- Security team and update process to address critical security defects
- Development tools:
    - Integrated with Mentor Embedded Sourcery™ CodeBench to include Mentor Embedded Linux ADE and Qt® Graphics and GStreamer analysis tools
    - Performance-optimized compiler (GCC) and runtimes
    - Application and kernel development and debug tools including advanced software insight analysis
- Industry leading quality infrastructure and process
- Extensively tested

CONCLUSION

The number one reason why software developers move to open source Linux is control and perceived cost savings.

But as this paper describes, in order to save on development and maintenance costs, a well thought out plan is required. It shouldn't come as any surprise that software developers today are transitioning to open source more now than at any other time in the history of the Linux platform. The Linux community is an invaluable resource to developers, but developers must understand what the community can and cannot do. With a large Linux community of developers there are numerous developers who can assist and develop Linux applications.

There is a lot of expertise out there – ready to assist you.

For more detailed information, please visit the Mentor Embedded Linux product page, or for general Linux information, please visit The Linux Foundation.

Author biography:

Kathy Tufto is the senior product manager at Mentor's Embedded Systems Division responsible for Mentor Embedded Linux, Android Services, and Mentor Embedded Sourcery CodeBench. Before arriving at Mentor, Kathy worked at The MathWorks as a senior training engineer and senior course developer where she taught and developed courses in the area of multi-domain simulation, model-based design, and embedded code generation for dynamic and embedded systems. Kathy holds an EE degree from Boston University.