Switching from Windows Embedded to Embedded Linux

By Sean D. Liming and John R. Malin
Annabooks

August 2013

Desktop Operating Systems to Embedded

With the advent of microcomputers, operating system development started following two paths. One path was the operating systems specifically designed for embedded systems, like VRTX which eventually become VxWorks. The other path was the operating systems for general purpose microcomputers like UNIX inspired Linux and the transition from MS-DOS to Windows. With the enormous popularity of microcomputers, came a surge of development effort. The sophistication of the operating systems grew quickly as did the user interfaces. Operating system services just bloomed providing wired and wireless network services, plug-and-play devices, audio processing, graphics and video processing, and intuitive graphic user interfaces became the norm. It didn't take long for the embedded system developers to start looking at all the features and services that were available in the desktop microcomputers and want to incorporate them in their embedded systems. It also did not take long for the embedded system developers to realize that they did not have the time nor the budget to replicate these features and services in their own proprietary software. Thus, there came a desire to take the desktop systems and adapt them to embedded systems. Embedded Linux and Windows Embedded grew out of that effort.

Windows being a closed source, proprietary operating system and Linux being an open source, openly distributed operating system led to different embedded methodologies and build tools. Because of the investment in time and effort to learn either of the systems, embedded developers tend to focus on one embedded system or the other. Therefore, there are a large number of Windows Embedded developers who know very little about embedded Linux and vice versa.

As long time Windows Embedded developers, we have seen Microsoft's embedded offerings go from selling MS-DOS as an afterthought to a full-fledged Windows Embedded OS. Our company offers a unique approach to learning Windows Embedded through articles, books, training classes, and consulting services. Over the years, we have had many different developers with a wide range of experiences come to our training classes. Over the past 8 years, there have been few Linux developers come to our classes. The reasons given for going from embedded Linux to Windows Embedded varied, such as Java being stale, corporate or customer pressure to the use Windows, device driver support, or just kicking the tires. Being Windows developers for so long, we didn't have enough background to address the differences between the two operating systems. Topics like upgrading and controlling the user interface came up regularly. Through our consulting services, there have been a few customers who had strong Linux backgrounds. One customer focused on developing the application in Java and just let us build the Windows Embedded operating system since no one on their team wanted to touch Windows. Another customer was building an internet appliance device that had no GUI display. The device took multimedia content off the network for processing. Why they chose Windows Embedded rather than Linux was puzzling. Even as Windows developers, we would have recommended Linux for the project.

These experiences with Linux developers were adding up, and with the success of Linux in the embedded market, we knew at some point we would have to spend some time learning Linux to be able to speak intelligently to our customers. In 2008, Microsoft announce product name changes and the desire to focus on vertical markets, which left the general embedded market open to everyone else, and everyone else was Linux. For several years after the announcement, Microsoft didn't update Windows CE and only produced Windows Embedded Standard 7, thus learning Linux became a priority. Since we had Linux developers trying to learn Windows, the obvious questions was "how does one go from Windows Embedded to embedded Linux?" A journey into embedded Linux took us through several distributions, discussions with a few colleagues and customers, meeting new contacts, trying different tools, some luck, and eventually a couple of Linux Foundation training classes. Having some exposure to other operating systems,

it was not too difficult to learn the terminology and understanding of the tools. One end-result was a short book that covers firmware, embedded Linux cross-compiler tools, and application development. The most important result was the knowledge that allows us to have a better constructive discourse with future customers.

Learning about Linux takes some work and practice. Open source projects are constantly under development, and there are some pitfalls to avoid when getting started. This paper is for Windows Embedded developers looking to move to embedded Linux. The paper explores the rationale, comparisons, and tools and training options to make the switch.

The Reasons to Switch to Embedded Linux

Why you want to switch is an important question to ask. Microsoft has some strong product offerings, but they don't reach every processor or device category. Microsoft has gone through some turmoil with organization and executive management changes. The focus on embedded products has been spotty with some technologies being hyped only to be withdrawn. Anyone building an embedded device has to ask the hard questions about using Windows Embedded:

- Is Microsoft abandoning the embedded market?
- Will investing in new technology for Windows be available in the future?
- Will technical support be available to fix problems to Windows Embedded closed / proprietary source code?
- Will there be future product and marketing changes in licensing and/or activation that prevent using Windows Embedded in the future?

Embedded Linux offers relief from an unstable and unfocused business model, and there are also strong reasons to consider using Linux:

- Complete source code availability makes it easier to fix items yourself rather than be dependent on black boxes that are under some else's control.
- Development tools to support different processors are available as free download or part of a year payment support offering.
- Not dependent on the whims of a single company. There are many Linux distributions available with companies and organizations to support them. Companies from large Fortune 500 to smaller organizations provide support for Linux.
- A company will have access to a worldwide pool of developers and resources, and there will be no lack of engineers who can build a custom Linux operating system. The Linux community is strong, and their developers are willing to help provide answers. We even talked to a key player in the Linux community, kernel maintainer Greg Kroah-Hartman, during our investigation into embedded Linux.
- Many universities offer Linux courses, thus future engineers will be available with the basic knowledge.
- Availability of different programming environments such as Java, Qt, and .NET also means a company can port existing Windows applications over the embedded Linux.
- BSP support is available for a variety of processors: ARM, x86, PowerPC, MIPS, etc. as well as driver support for many peripherals. Linux drivers are as readily available as Windows drivers.

Common Concerns about Linux

A move to Linux brings up the concerns about protecting intellectual property, multiple open source license agreements, and development tool stability. These were some of the same concerns we had with open source when embedded Linux started appearing in the late 90's. As we found, many of these concerns are brought about by misunderstanding, competitive spin, and some fear of the unknown.

Open source software means that you have to address open source licensing. There are multiple open source license agreements that can go into a Linux distribution. Windows Embedded does have an advantage with only one license agreement. If you develop a product with Windows Embedded, a license agreement is signed for the right to put Windows Embedded in your

product, and you must pay a per unit royalty. If a company writes an application in Visual Studio and only uses the APIs provided by Microsoft, then the company owns the application outright, and source code doesn't have to be provided. When it comes to Linux, the biggest concern is that a company has to give up the source code to their product when using open source.:

For Embedded Linux, there are many different open source license agreements such as GPL, LGPL, MIT, Apache, etc. To assist with the complexity and act as a neutral organization, the Linux Foundation has developed a set of open source tools, training curricula and a self-administered assessment checklist that will allow companies to ensure compliance in a cost-effective and efficient manner. The Open Compliance Program also includes a new data exchange standard so companies and their suppliers can easily report software information in a standard way.

The other common concern is that open source development tools are unstable. The stability concern comes from history and a misunderstanding of how open software works. Linux is constantly under development. The early days of Linux saw many revisions of the tools and of the distributions, as independent programmers contributed their time and effort. As time has progressed and Linux has grown, many large technology companies are investing in several of these projects, as well as, creating working groups to promote specifications and working groups. Today, software tools have matured and many working groups are managing orderly releases and long-term support.

Comparison of Windows Embedded and Embedded Linux

A direct comparison between Windows Embedded and embedded Linux is not feasible. Embedded Linux is scalable from small devices without an MMU to large-scale server systems. Windows Embedded is not one product, but several products, each designed to address a specific market segment or device. The best comparison is done by looking at each Windows Embedded product in-turn.

Windows Embedded Compact

Windows Embedded Compact (formally known as Windows CE) was developed in the 1990's as a small ROMable Windows operating system for PDA and smart phones. Windows Embedded Compact has been replaced by Windows 8 for ARM as the current Windows Phone operating system. Windows Embedded Compact is not a serious part of Microsoft Corporation's future strategy, but it is still part of the Windows Embedded product line up. At the time of this writing, the latest release is Windows Embedded Compact 2013.

Attribute	Windows CE 5.0	Windows Embedded CE 6.0	Windows Embedded Compact 7	Windows Embedded Compact 2013	Embedded Linux
Processors	X86, ARM, SH, MIPS	X86, ARM, SH, MIPS	X86, ARM, MIPS	X86, ARM	X86, ARM, PowerPC, MIPs
Real-Time support	Yes	Yes	Yes	Yes	Real-Time support option available in the kernel
Tools	Platform Builder, Windows Embedded CE Test Kit (CETK)	Visual Studio with Platform Builder Plug-In, Windows Embedded CE Test Kit (CETK)	Visual Studio with Platform Builder Plug-In, Compact Test Kit (CTK)	Visual Studio with Platform Builder Plug-In, Compact Test Kit (CTK)	crosstool-ng, PTXdist, Buildroot, Yocto Project

IDE	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Eclipse, NetBeans, Anjuta, Aptana Studio, Bloodshed Dev, Code::Blocks, Geany, Qt Creator, IntelliJ IDEA
Kernel Programming Languages	C/C++	C/C++	C/C++	C/C++	C
Application Programming Languages	C/C++, C#, VB .NET, Java	C/C++, C#, VB .NET, Java	C/C++, C#, VB .NET, Java	C/C++, C#, VB .NET, Java	C/C++, C#, Java
API's	Win32, MFC, .NET Compact Framework	Win32, MFC, .NET Compact Framework	Win32, MFC, .NET Compact Framework	Win32, MFC, .NET Compact Framework	ANSI standard API's, POSIX, Mono (.NET Framework)
Host Development System	Windows	Windows	Windows	Windows	Windows, Linux
Runtime Royalties	Yes	Yes	Yes	Yes	None
Cost of Development Seat	$995	$995	$995	~$1,000 (estimated)	Free
Lifecycle Support End Date	10/14/2014	4/10/2018	4/13/2021	2023 (estimated)	None

Windows Embedded Standard

Windows Embedded Standard (WES) is the Windows desktop designed for the embedded market space. WES includes additional lock down features to support systems that are in harsher environments. The WES roadmap stretches all the way back to Windows NT with the following product releases: Windows NT Embedded, Windows XP Embedded (now called WES2009 or simply Windows Embedded Standard), Windows Embedded Standard 7, and Windows Embedded 8 Standard. Since WES is built from the same Windows desktop code, all the application and driver support for desktop Windows is available for WES.

WES has also attracted a spinoff called Windows Embedded Industrial (formally Windows embedded POSReady), which addresses the needs for the point of sale market.

Attribute	**Windows Embedded Standard 2009**	**Windows Embedded Standard 7**	**Windows Embedded 8 Standard**	**Windows Embedded Industry 8.x**	**Embedded Linux**
Processor	x86 only	x86 and x64 only	x86 and x64 only	x86 and x64 only	X86, ARM, PowerPC, MIPs
Real-time	Via 3rd party extensions	Via 3rd party extensions	Via 3rd party extensions	Via 3rd party extensions	Real-Time support option available in the kernel
Minimum Image Size	500 MB Boot Media	500 MB (32-bit)	2 GB (32-bit) 4 GB (64-bit)	2 GB (32-bit)	5 MB image size can be

		1 GB (64-bit)		4 GB (64-bit)	achieved for the simplest applications, but typical size for x86 is 256MB,
Tools	Target Designer, Component Designer, Database Manager, Target Analyzer, Sysprep	Image Builder Wizard, Image Configuration Editor, Windows Embedded Developer Update, Deployment Image Servicing and Management, ImageX, Target Analyzer, Sysprep	Image Builder Wizard, Image Configuration Editor, Module Designer, Windows Embedded Developer Update, Catalog Manager, Deployment Image Servicing and Management, ImageX, Target Analyzer, Sysprep	Optional: System Image Manager	crosstool-ng PTXdist Buildroot Yocto Project
IDE	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Visual Studio, Visual Studio Express	Eclipse, NetBeans, Anjuta, Aptana Studio, Bloodshed Dev, Code::Blocks, Geany, Qt Creator, IntelliJ IDEA
Kernel Programming Languages	C/C++,	C/C++	C/C++	C/C++	C
Application Programming Languages	C/C++, C#, VB .NET, Java	C#, VB .NET, Java	C#, VB .NET, Java	C#, VB .NET, Java	C/C++, C#, Java
API's	Win32, MFC, .NET Compact Framework, .NET Framework 2.0 through 4.5	Win32, MFC, .NET Compact Framework, .NET Framework 2.0 through 4.5	Win32, MFC, .NET Compact Framework, .NET Framework 2.0 through 4.5	Win32, MFC, .NET Compact Framework, .NET Framework 2.0 through 4.5	ANSI standard API's, POSIX, Mono (.NET Framework)
Host Development System	Windows	Windows	Windows	Windows	Windows, Linux
Runtime Royalties	Yes	Yes	Yes	Yes	None

Cost of Development Seat	~$1,000	~$1,000	Free	Free with signed license agreement	Free
Lifecycle Support End Date	1/14/2014	10/13/2020	2023 (estimated)	2023 (estimated)	None

Embedded Linux Options

For Windows Embedded, all the development tools to build the operating system and applications come from a single vendor. The only exception is the .NET Micro Framework, which requires external tools to build the firmware. In contrast, there are many development tool options available for embedded Linux, thus the most important step in switching to embedded Linux is the selection of development tool chain and/or distribution. The Linux Foundation offers a publication that dives into evaluating cross development tool chains.

The tasks for an embedded Linux project are as follows:

- Choosing the development tools to build the kernel – The cross compiler for the kernel might be different from the compiler for the applications, since the kernel cannot be linked to user space libraries.
- Choosing the development tools to build the user space applications.
- Building the boot loader.
- Configuring and porting the kernel.
- Creating the file system.

A company can choose to build a distribution from scratch, but the amount of resources and cost to maintain the distribution and tools may be a negating factor. Rather than build custom tools, there are several auto-tool chains available that help build the kernel, file system, or both. Some tool chains are freely available, and they are sold with additional development tools. Some popular examples are, Cross Linux from Scratch, crosstool-ng, PTXdist, Buildroot, and the Yocto Project. Alternatively, a company could choose a distribution that is already available. There are over 500+ Linux distributions to choose from. Some are very popular, and some are very obscure. Some distributions focus on embedded, while others are targeted for specific uses.

The choice of CPU also plays into the selection process since CPU compiler optimization and device driver support are important factors. A highly optimized compiler will contribute to better system performance, and the more drivers supported means less work that has to be done at the kernel level. Many development boards come with different distributions. Selecting the right tools is important not just for the short-term project, but for system performance and the longer-term life cycle of the product line. As developers move to other projects, it is important to consider the longevity and popularity of the tool chain so future resources are available.

The Yocto Project

The wrong choice of development tools could lead to project delays due to code issues or kernel problems. The problem is there are too many choices, and trying to pick the right one might not work for the next project. A solution is needed to provide a cohesive, standard development platform for building Linux kernels for the target hardware that allows both diversity and choice, but provides committed support. Seeing a need in the embedded market, various companies and Linux developers came together to create such as solution called the Yocto Project.

The Yocto Project™ is a Linux Foundation workgroup and open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture. It was founded in 2010 as a collaboration among many hardware manufacturers, open-source operating systems vendors, and electronics companies to bring some order to the chaos of embedded Linux development.

The Yocto Project is a complete embedded Linux development environment with tools, meta-data, and documentation. The Yocto Project allows the developer to create a custom Linux distribution using upstream projects.

The Yocto Project relieves the developer from having to specifically identify and download the source of each of the packages that make up a Linux distribution, setting up a system with the correct set of tools to build these packages, and finally assembling them into an embedded Linux image. The Yocto Project builds upon and extends a series of upstream, open-source projects that form its toolkit. With each Yocto Project release, updated versions of these projects with new features and bug fixes are included. Every new release also provides updated recipes and templates to track changes, features, and bug fixes for the source packages that make up a Linux distribution.

In short, the Yocto Project delivers a complete solution to build the operating system and applications just like Windows Embedded. The difference is that the source code is pulled from different upstream projects from the Internet. The interesting thing about the Yocto project is how it parallels the operating system development ideas from Windows Embedded. Recipes like components are used to define a single selectable feature of the operating system. You can pick and choose the features that go into an embedded Linux distribution just like picking features for Windows Embedded operating system.

Yocto Project Structure

At the center of the Yocto Project is Poky, a platform independent, cross-compiling layer that utilizes the OpenEmbedded core. Poky provides the mechanism to download, patch, build and combine thousands of distributed open-source projects to form a complete and coherent but fully customizable Linux software stack.

>>>>> Figure 1 goes here

Figure 2 depicts the Yocto Project workflow based on OpenEmebdded. User Configuration, Metadata, Hardware Configuration, and Policy Configuration determine which Upstream Projects, Local Projects on the developer’s system, and what sources from optional Source Change Management system to include, as well as, to control the individual process steps of the workflow. A Yocto Project release already contains configuration information and metadata that will produce a working Linux system image, thereby requiring minimal adaption to the developer’s local build environment.

>>>>>> Figure 2 goes here

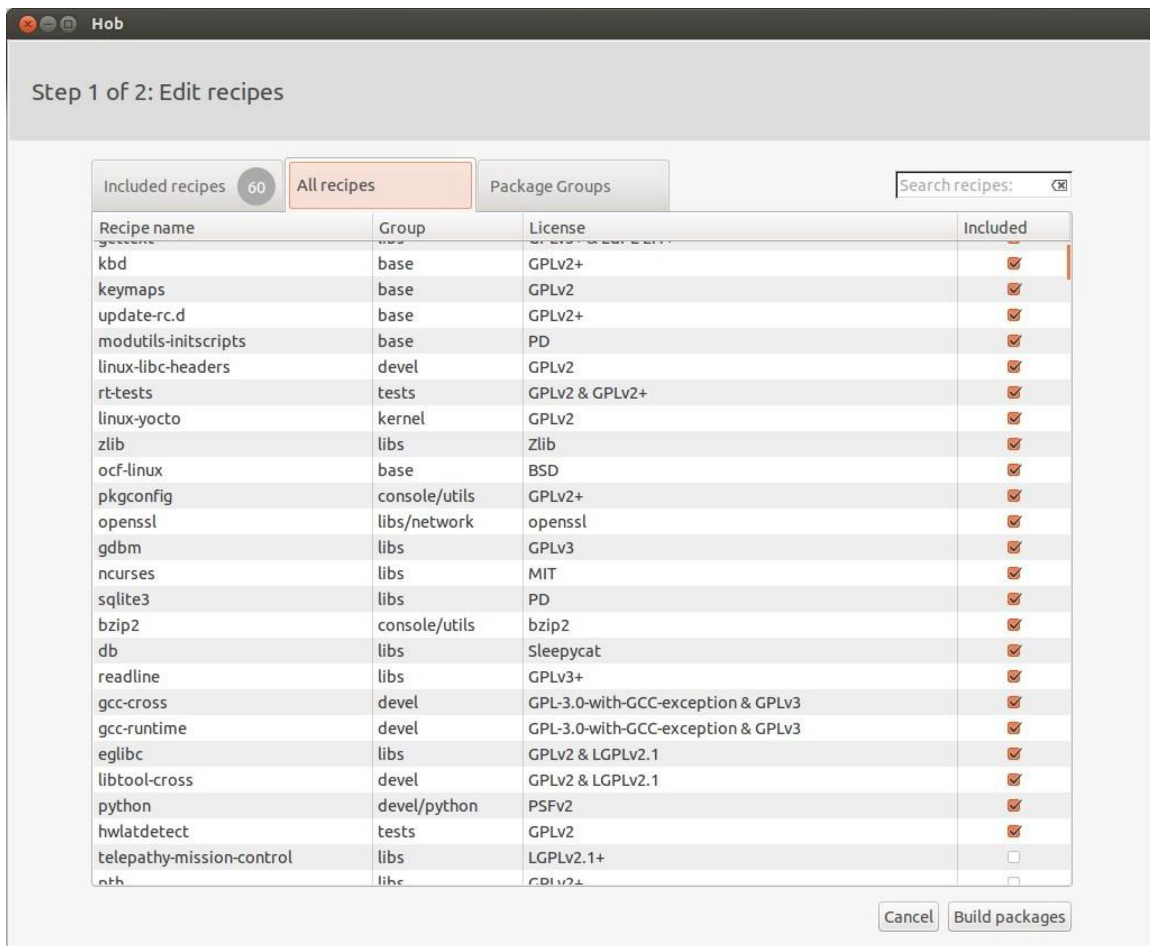
Figure 2 also shows the linear workflow process for creating a Linux image and application toolkit. The steps consist of: Source Fetching, Patch Application, Configure/Compile, Output Analysis for Packaging, Package Creation and QA Test. These steps are in fact repeated for each source package before all sources or Package Feeds have been created and can be combined into an image. The Yocto Project supports multi-processor and multi-core build systems by automatically calculating dependencies and executing process steps in parallel, managing changes, and only rebuilding package feeds who’s input, source, metadata, dependencies, or other parameters have changed, which greatly accelerates the entire build process.

To further simplify the process for a novice, as well as experienced developers, the Yocto Project provides a series of different profiles that automatically create images for common applications:

- core-image-minimal – A small image just capable of allowing a device to boot
- core-image-base – A console-only image that fully supports the target device hardware
- core-image-core – An X11 image with simple applications, such as terminal, editor, and file manager.

- core-image-lsb – An image suitable for implementations that need to conform to Linux Standard Base (LSB).
- core-image-sato – An X11 image with example GUI implementations using Matchbox Windows manager, Sato theme, and Pimlico Applications. The image also contains terminal, editor, and file manager.
- core-image-sato-dev – An X11 image similar to core-image-sato but also includes a native toolchain and libraries needed to build applications on the device itself. Includes testing and profiling tools, as well as, debug symbols.

The above list represents a few of the available profiles. Many more specialized profiles are available, and the developer has the ability to create their own. To assist with developing custom distributions, the Yocto Project comes with a GUI development tool called Hob, which is a graphical interface to bitbake. With Hob, the developer adds the BSP and other support layers, and then they can pick and choose the different features or recipes to include in the build. Hob automatically performs a dependency check to make sure the features selected have their required support libraries included. Hob parallels some of the Windows Embedded configuration editing tools.



Yocto Project Components & Tools
Yocto Project features include:

- Application Development Toolkit installer and Application Development Guide
- Eclipse IDE Application development plug-in
- Board Support packages for Atheros RouterStatationPro, FreeScale's MPC8315E, Intel's Atom processor, Texas Instruments' OMAP, and others.
- GCC toolchain

- Poky and an improved OpenEmbedded bitbake
- System toolchain bootstrapping and machine specific sysroot.
- X Windows System

Platforms

Yocto Project supports popular hardware including:

- ARM
- MIPS
- PowerPC
- X86 architectures

Yocto Project Collaborators

The Yocto Project is actively supported by embedded and embedded Linux industry leaders in embedded design, development, tools, and products, including:

Hosted by The Linux Foundation
The Angstrom Distribution
ARM
Cisco
Dell
ENEA AB
Freescale Semiconductor
Fujitsu
Gumstix
Hewlett-Packard
Hitachi
Huawei
IBM
Intel
Juniper Networks
LSI Corporation
Mentor Graphics
Montavista
Motorola
NEC
OpenEmbedded
Oracle
Qualcomm
Sony
SuSe
Texas Instruments
Timesys
Wind River
And many more…

Writing and Porting Applications to Embedded Linux

With so many options, finding a Linux distribution that will accommodate your hardware platform will not be difficult. The real challenge will be to port existing Windows Embedded applications to Linux. The cost and effort will vary depending on what programming language the current application is written in and what API and support libraries were used. Windows Embedded applications are written in many different language/API combinations. The most popular are C#/.NET Framework and C++/MFC. Java, Adobe Flash, Adobe Air, and HTML are also used. The good news is that all these same languages are available for Linux. Popular API sets for Linux are QT and X-Windows. There is even support for .NET Framework through the Mono Project.

Porting a Windows Embedded application to Linux will take some effort. Even if you have written the application in a managed code language like .NET Framework or Java, some adjustments must be made to interface to the underlying hardware. In the case of .NET Framework, the Mono project currently supports everything in .NET Framework 4.0 except for WPF, WCF, and WWF. There is also some limited .NET Framework 4.5 support for C# 5.0 async support and Async Base Class Library support. The Mono Project has a tool, the Mono Migration Analyzer (MoMA), to help analyze an application to see if the application uses something not supported by Mono. If something is not supported, it is best to re-write the application to achieve the best optimized solution.

Real-time support is available for Linux. The most challenging applications to port are deterministic or real-time applications. These applications will take more development effort to re-write since real-time API's, kernel architecture, and timing are different. Some silicon-vendors offer tools to measure and track real-time response for Linux applications.

What Windows Embedded Developers Need To Know About Linux

Going from one operating system to another can be a fun challenge. With the journey into Embedded Linux, we have a better understanding of the differences between the two operating systems. There are some key points Windows Embedded Developers need to understand when developing with Linux

The Obvious Differences

Windows and Linux architecture and setup are completely different. The thinking and approach to Linux developments tools takes some getting used to for the Windows Embedded developer. Windows Embedded provides all the tools and cross compilers already built. The developer just focusses on developing the applications, drivers, and final image. Windows comes from a single company, and updates to the tools are based on what the single company has planned. Linux or GNU/Linux is comprised of the kernel, tools, and various projects that can make up the root file systems that come from different sources. Most importantly, you have to build the tools first; and then you can focus on applications, drivers, and image development. Linux has an advantage that it is constantly being updated and popular features give way to new approaches. Solutions like the Yocto Project attempt to provide all the tools for you.

The best way to get started is to learn Linux by installing and getting familiar with one of the popular distributions. Learning little things like "\" in Windows versus "/" used in Linux, home directory versus root directory, shell commands, case sensitivity, avoiding spaces in file names and directories since scripts can get tripped-up, and scripting take some getting used too.

Image Development Approach

The real difference is in the approach to building an OS image. Windows Embedded provides the means to setup user accounts, passwords, computer name, system shell, startup services, device drivers, and applications before building the image so the provisioning is completed during OS installation. There is some provisioning for Windows Embedded after installation. Embedded Linux allows you to include drivers and applications, but all provisioning and tuning must happen after the image is installed. Post-install provisioning is a minor change in thinking but a very important one. Scripts must be created to automate the build process.

Starting Linux

The whole methodology for designating and launching a shell in Linux is very different from Windows Embedded. Every Windows OS requires a shell and what shell is launched is defined in the registry. The Windows registry was created as a single place to configure the Windows operating system. Windows EC and Windows have a structured start sequence that involves the registry. A registry doesn't exist in Linux; so as a Windows developer, you will have to get comfortable with scripts. Also Linux has multiple terminals (TTY) that can run different sessions so there is no single shell. After the Linux kernel loads, it looks to start its first process, which then brings up all the services and most importantly user space. The older popular initialization for Unix/Linux is called System V, which has a structured mechanism to call scripts at different

runtime levels. The scripts themselves have a structure to handle startup and shutdown. In System V, the scripts run in a linear chronological ordering. This System V has been around since 1969, and has been found to be too slow by today's standards. To provide parallel processing and faster boot time, newer distributions are moving to systemd. Systemd is a system daemon that manages other daemons. Rather than a sequential start where one service waits for the other, systemd uses Unix domain sockets for inter-process communication. All sockets for all daemons are created and then all daemons. Any client requests for daemons that are not yet running are cached in the socket buffer and then fulfilled when the daemon is running. This allows for a lot of parallel processing during the boot sequence. The idea is to maximize CPU and memory resources to achieve a faster boot time. When moving to embedded Linux, it is important to learn about systemd..

Launching GUI applications requires knowledge of how to create scripts for X-Windows server. System V or systemd will launch X-Windows, and X-Windows, in turn, runs through several initialization scripts to launch a GUI interface.

Drivers and Kernel Tuning

Finally, there is device driver support. Most driver types are supported in the kernel. Drivers can be built into the kernel or provided as loadable modules. Tuning the Linux kernel is an important part of the development processes, and there are a couple of books that discuss configuring the kernel. If you have been working with Windows CE/EC, then building drivers into the kernel image is nothing new.

Learning all these different specifications and implementations takes time, but training is available to help get developers through the learning curve faster.

Embedded Linux Training

Training is a key step to consider when making the switch to embedded Linux. Information is scattered across the web, and there are books that provide self-help information. Instructor-led training courses provide the best starting point to get familiar with embedded Linux. The Linux Foundation offers a variety of instructor-led training course to meet different project requirements. The Linux Foundation training offers Corporate Linux Training that includes custom and on-site options. From our experience, these classes were invaluable to learning Linux quickly.

Linux Foundation Courses

LF411 - Embedded Linux Development	LF262 - Developing with Git
LF211 - Introduction to Linux for Developers	LF410 - Embedded Linux Development Crash Course
LF205 - How to Participate with the Linux Community	LF404 - Building Embedded Linux with the Yocto Project Crash Course
LF312 - Developing Applications For Linux	LF488 - Implementation and Management of Open Source Compliance
LF320 - Linux Kernel Internals and Debugging	LF384 - Overview of Open Source Compliance End-to-End Process
LF331 - Developing Linux Device Drivers	LF281 - Executive Review of Open Source Compliance
LF405 - Building Embedded Linux with the Yocto Project	LF272 - Open Source Compliance Programs - What You Must Know
LF432 - Optimizing Linux Device Drivers for Power Efficiency	LF271 - Practical Guide to the Open Source Development Model
LF329 - Introduction to Android	LF315 - Inside Android: An Intro to Android Internals
LF308 - Introduction to Embedded Android Development	LF295 - Android Bootcamp
LF363 - Portable Application Development for Tizen Devices	LF273 - Tizen Training For Non-Developers

Conclusion

Whether considering a new project or a new iteration of an existing project, it is important to ask the fundamental questions of support and longevity. Windows Embedded and embedded Linux have been around for about the same length of time. A single company, Microsoft, controls Windows Embedded and the development tools that go with it. Market conditions change and company focus can shift. Having a dependency on a single source that can change direction or terminate technologies can delay or even kill a project. Even if Microsoft does not change its embedded strategy, it promises 10-year support for its embedded releases, and that means 10 years of support from when the product is initially released, not from when you chose to put it into your product. Embedded Linux provides vendor independence. Embedded Linux solutions come from several vendors and range greatly in offerings and prices, making it easier to fit to various needs and budgets. The Linux community provides a source of virtually indefinite support.

BSP and device driver development can be costly. Support for processors and chipsets is a very important consideration when choosing an embedded OS. Embedded Linux offers all the generic driver support and many off-the-shelf applications. Most importantly, embedded Linux offers the scalability to support small, low-power devices all the way to large enterprise systems. Windows Embedded targets a more narrow range of hardware platform architectures that limit the practical market segments that can be addressed.

Finally, support is a critical consideration. Embedded Linux has a large development community, so getting support and finding resources is not a problem. The Linux Foundation is a valuable resource with tools like the Yocto Project, training, and other support resources to help make the transition from Windows Embedded to embedded Linux a smooth one.

References

https://www.linux.com/learn/tutorials/524577-here-we-go-again-another-linux-init-intro-to-systemd

http://en.wikipedia.org/wiki/Systemd

http://en.wikipedia.org/wiki/Init

https://en.wikipedia.org/wiki/UNIX_System_V

http://www.ibm.com/developerworks/library/l-linux-shells/

Embedded Systems Conference 2008 - Kevin Dallas keynote - http://www.slideshare.net/msftweb/windows-embedded-esc-silicon-valley-keynote-address

Windows Embedded Product Life Cycles - https://www.microsoft.com/windowsembedded/en-us/product-lifecycles.aspx

Starting Linux - http://www.databook.bz/?page_id=787