Embedded Linux with the Yocto Project

Abstract

When developing embedded products, creating a custom Linux operating system can be a daunting task. The challenge is choosing a single starting point: a multitude of Linux distributions are available to developers and users, but finding a standardized position from which to build is difficult. Enter the Yocto Project, a collaborative effort between the Linux Foundation and independent developers to gather standardized templates, tools, and methods for creating custom Linux systems. The open source tools are coupled with Poky, a versatile build system, to form a flexible, robust architecture for developers. This article gives an introduction to the Yocto Project's utility when creating your own Linux-based systems and offers a specific exercise to walk readers through development system setup using Yocto Project 1.3.

Embedded Linux with the Yocto Project

Now that we have covered the firmware to set up the system to boot the operating system, our next step up the stack is the open source operating system, Linux. If you ask the question in Google, "How many Linux distributions are there?" the answers will lead you to 100s, 1,000s, a Wikipedia page that covers all the known distributions, and a site that watches distributions. The problem with Linux is that there are too many choices. The issue facing the embedded developer is which distribution to choose. Pick something too proprietary, and it might be costly to support in the future. Pick something too popular, and you might end up with features you don't need. On top of all that, you have to track all of the licenses that you are using, and provide a list to the customer when you ship your device. Someone new to Embedded Linux could get lost in all of the development options. To help solve this dilemma, some embedded Linux engineers came together and developed the Yocto Project.

The Rise of Linux

In the 1980s and early 1990s, the Embedded Market was fragmented with several operating system and kernel offerings. The commercial offerings included pSOS, VxWorks, QNX, OS9, LynxOS, IBM DOS, and MS-DOS. Many companies had their own proprietary homegrown kernels. Homegrown meant that there was one or two people that knew all about how the kernel worked. In the early part of the decade, Linus Torvalds set out to create a free, open source kernel that ran on the 80386 processor. Using GNU tools he created Linux, which was a UNIX-like operating system. Since it was open source, the new project started getting some help.

The 1990s was a dynamic period with the growth of the Internet and dynamic changes in the Embedded Market. WindRiver acquired pSOS and immediately killed it in favor of their VxWorks. Microsoft introduced Windows CE to target handheld devices, but embedded developers also showed great interest. Linux was taking shape and different distributions started to appear. The focus for Linux was servers and some attempts at desktop systems. Windows NT Embedded officially arrived at the end of the decade to round out Microsoft's two prong strategy for embedded.

To save on costs, companies began to discard the old homegrown kernels in favor of commercial or open source solutions. Linux kept gaining traction with each newer kernel update. Real-time versions from a few vendors started to appear. Slowly Linux began a takeover much of the embedded market.

Even WindRiver had to add a Linux solution. In the 2000s, operating system choices came down to two operating systems families: Windows Embedded and Linux. When Apple introduced the iPhone and Google introduced Android, Linux has risen to the top of the operating system world. Today, Linux and UNIX devices are found almost everywhere. Some of these devices are everyday items like flat screen TVs, DVD players, set-top boxes, phones, and wireless routers.

Linux and Windows Kernel

The kernel is at the heart of an operating system. It's the brains or conductor that manages the tasks the processor(s) will perform. Traditionally, there are two types of kernel architectures: monolithic kernel (mono-kernel) and microkernel. A microkernel provides the basic set of core functionality such as thread management, synchronization functions, low-level address space management, and inter-process communications. The other functions of the operating system run in user mode space such as file systems, networking stacks, and device drivers. Two good examples of a microkernel are QNX® Neutrino® RTOS and MINIX 3. A microkernel depends heavily on task switching to pass messages around to each sub system. Microkernels are portable and reliable, but because of the context switching overhead, don't perform as well as mono-kernels.

In contrast, a mono-kernel has all of the OS functions within the kernel. A mono-kernel is just one big program. Linux is the best example of a mono-kernel. A mono-kernel performs very fast, since it doesn't have as much context switch overhead. If the kernel crashes, the whole system goes down. There is a nice kernel map available on Wikipedia showing all the individual modules that make up the Linux kernel.

The Windows NT kernel is more of a hybrid that a true microkernel. There were some specific design goals behind the Windows NT kernel: extensibility, portability, reliability, compatibility, and performance. There is a good book that covers the original design goals listed at the end of the chapter. For the user mode space, all applications interact with subsystems such as Win32 or POSIX. The subsystem interface to the kernel's exposed system services. The kernel itself has a layered approach for the kernel mode layer. Above the kernel are a few managers such as memory manager, process manager, security monitor, plug-n-play manager, graphics manager, and I/O manager. Below the kernel is the hardware abstraction layer or HAL. Portability between platforms is the main purpose of the HAL. All device drivers must go through the HAL layer to interact with hardware. Linux doesn't restrict applications' direct access to hardware, but Windows will not allow direct access. The next chapter will demonstrate an application in Linux directly accessing hardware.

Windows NT started out on the DEC Alpha processor, and early versions were ported to MIPS, PowerPC, and x86. Overtime, x86 became the only supported processor until it was recently ported to ARM processors. The ARM version is called Windows RT. The only subsystem that is supported on ARM is the new WinRT subsystem.

All this discussion about kernels is mundane to most developers. One of the biggest differences between Linux and Windows is how the OS configuration parameters are stored. Linux stores the OS parameters in various configuration files, which adds some complexity to those new to Linux. Windows stores all the parameters in a database called the registry. Some Linux developers who have come to my classes have commented or questioned the concept of the Windows registry. The original Windows 3.1 for MS-DOS did use individual configuration files (INI). For security and programmability, the registry was developed to store the data. When going from Windows Embedded to Linux Embedded, you will have to get use to configuring the kernel using different parameter files.

Linux Development

Windows Embedded Standard (WES), which is the Windows desktop embedded release, allows the Windows OS to be built from about 100 selectable packages. There is no compiling of the operating system. Basically, you are installing the binaries as they come in Windows desktop. Windows CE (Windows Embedded Compact) is a completely different operating system, but like Linux it is built from source code.

Linux is available in what is known as a distribution. Ubuntu and Fedora are two popular distributions. There are many more. Unlike WES, Linux is built from source code. There are different build systems available to create a custom distribution. Since Linux supports different processor architectures, cross compiler tools are needed. The basic development process for Linux goes as follows:

1. Download and compile the tool chain
2. Download, configure, and build the kernel
3. Build a boot loader
4. Build a file system.

You can create a Linux distribution from scratch, but most developers prefer to use a standard tool chain. There are many development tools available such as crosstool-ng, PTXdist, and Buildroot. If you are looking to go from Windows Embedded to Linux, you will come across the Yocto Project.

What is the Yocto Project?

If you are familiar with Windows Embedded, you probably have used Platform Builder, Target Designer, or Image Configuration Editor. All of these tools allow you to build custom Windows Embedded images by choosing from a set of components or modules. The Yocto Project is a custom build engine for Embedded Linux. The Yocto Project contains the Poky build system to build a custom distribution from different packages.

Managed by the Linux Foundation, the Yocto Project gets contributions from various Embedded Linux developers. You have access to many of the Linux library and application projects that have been up-streamed for use on Linux. Images can be targeted for x86, ARM, MIPS, and PowerPC.

More details can be found in the quick start user guides and development manuals found on their website at https://www.yoctoproject.org/. This chapter covers setting up the development system and looks at the different ways to build a distribution. Rather than spend the time getting into too much explanation, let's jump right in and get the system set up and start building a distribution. The explanation will follow.

Exercise: Development System Setup

If you are a just getting started with Embedded Linux and the Yocto Project, the Yocto Project Quick Start guide is a very good starting point. The steps below are for Yocto Project 1.3. I am going to expand the information and focus on developing under Ubuntu.

Development System Requirements

The most important requirement is choosing the right hardware for development. A faster processor with multiple cores, lots of memory, and a fast Ethernet connection is the best for Yocto. A system with an i5 with 2 physical cores and 4 GB RAM took several hours to build the OS. A system with an i7 Ivy Bridge that has 4 physical cores and 16 GB RAM brings the build time down to around an hour and 30 minutes.

Disk space is also important since the largest project can consume 50GB space. 250GB to 512GB disk is recommended.

Ubuntu - Development System Setup

The Yocto website calls out specific Linux distributions that have been validated to work with the Yocto Project. These distributions include CentOS, Fedora, OpenSUSE, and Ubuntu. For these instructions, we will use Ubuntu 12.04 LTS 64-bit.

1. Assuming that you have a clean system to set up and another system to download Ubuntu, download Ubuntu from their website at http://www.ubuntu.com/.

2. Create a bootable disk using the ISO that you just downloaded. You can create a bootable USB flash disk from Ubuntu or use a special Windows utility to create a bootable USB flash dish under Windows.

3. Boot up the development machine and install Ubuntu. As you install the OS, you will be asked for a user name. I recommend keeping the name short and one word with no spaces. A lot of the work is performed at the command line and spaces can introduce issues. You will also be asked for a password. Make the password strong but short since you will be using the password to boot the system a lot during development.

4. Once completed, you will want to download the latest updates for Ubuntu.

Once the updates have been completed, you may want to look into the following:

- Firewall – the firewall might be disabled. You can check the status in the terminal:

```
$ sudo ufw status
```

  To enable the firewall:

```
$ sudo ufw enable
```

- Check the hard drive access performance, and make tuning changes using the hdparm utility.
- VIM – Gedit is the standard editor for Ubuntu. For Windows developers, Gedit is analogous to Notepad, but offers a few more features. If you want the full Linux experience and keep your hands on the keyboard, you can download VIM or just us vi.

Yocto Project Setup

The Yocto Project comes as a single tar ball file, and there are separate BSP packages for different processor / platform solutions. You can download them as files through the website or via command line. Here are the basic steps:

1. Make sure that the system is connected to the Internet.

2. Create a main project folder called Yocto1.3 under your home directory.

3. Open a terminal; you can click on the dash icon and type terminal. You should lock the terminal to the launcher by right-clicking on the icon in the launcher and selecting lock to launcher from context menu.

4. Ubuntu has the dash shell as the default shell /bin/sh. Yocto will not work correctly with the dash shell so you will need to change the shell. Run the following in the terminal:

```
$ sudo dpkg-reconfigure dash
```

Using he arrow keys select “No”. Check to be sure the update occurred:

```
$ ls –l /bin/bash
```

The result should come back with /bin/sh -> bash, which means bash is now the shell.

5. Now, we need to download the tools and libraries needed for Poky. The Yocto Project Quick Start guide provides a list of items. These items can change over the different releases, so please review the current release to see if there are any additions. You can enter a single command line:

```
$ sudo apt-get install sed wget subversion git-core coreutils unzip texi2html
texinfo libsdl1.2-dev docbook-utils fop gawk python-pysqlite2 diffstat make
gcc build-essential xsltproc g++ desktop-file-utils chrpath libgl1-mesa-dev
libglu1-mesa-dev autoconf automake groff libtool xterm libxml-parser-perl
```

or put the command in a script file:

```
#!/bin/sh
sudo apt-get install sed wget subversion git-core coreutils \
unzip texi2html texinfo libsdl1.2-dev docbook-utils fop gawk \
python-pysqlite2 diffstat make gcc build-essential xsltproc \
g++ desktop-file-utils chrpath libgl1-mesa-dev libglu1-mesa-dev \
autoconf automake groff libtool xterm libxml-parser-perl
```

Use the chmod +x to make the script file executable.

Downloading, unpacking, and installing the tools and libraries takes a few minutes.

6. In the terminal, change directory to the Yocto1.3 project folder.

7. The next step is to download Poky. You can go to the download page (http://www.yoctoproject.org/download) to save off the tar file in the project directory. You can also use the command line:

```
$ wget http://downloads.yoctoproject.org/releases/yocto/yocto-1.3/poky-danny-8.0.tar.bz2
```

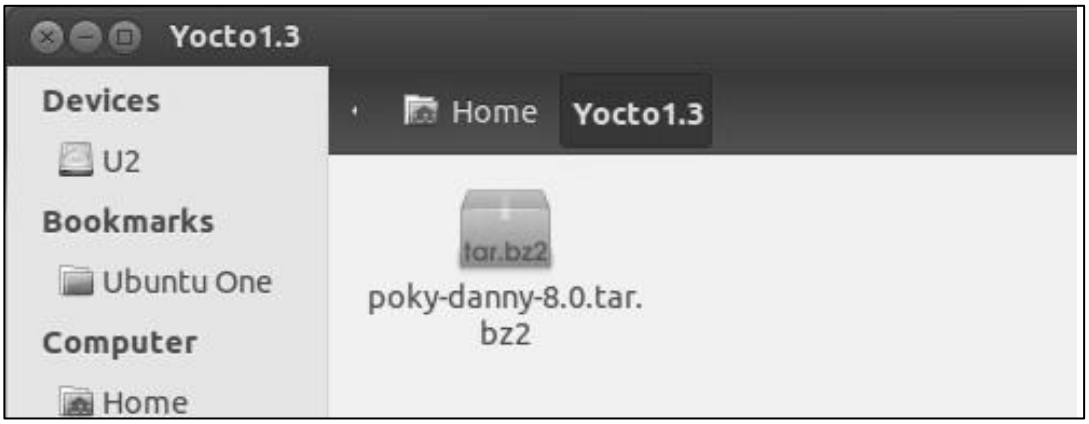

Figure 1: Yocto Project 1.3 Download

8. You can either double click on the tar file to open the Extraction tool and click Extract, or extract the tar files at the command line:

```
$ tar xjf poky-danny-8.0.tar.bz2
```

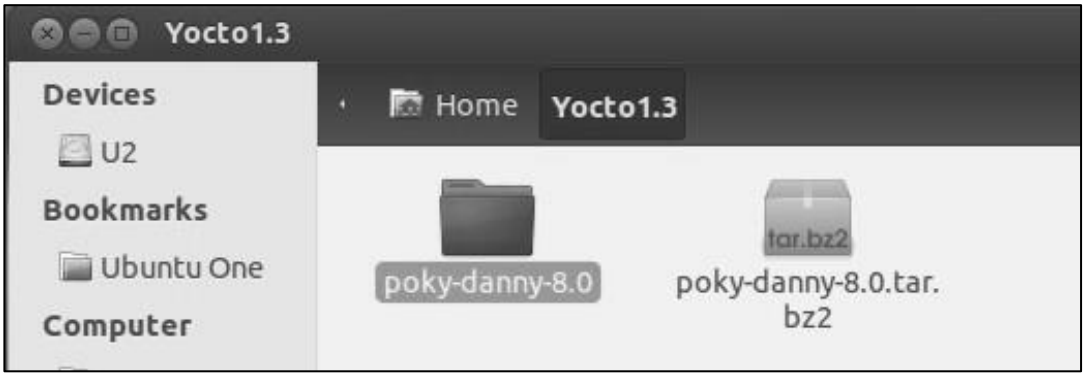

Figure 2: Poky Extracted

Poky comes with an emulator to test project builds, but building to real hardware is the main goal. There are a number of BSPs available from the download site: https://www.yoctoproject.org/downloads. We will download a few BSPs to test different projects.

9. Some target systems don't have a video display and only communicate over Ethernet or serial ports. In the case of serial ports, you will want to download the serial terminal application:

```
$ sudo apt-get install minicom
```

10. Minicom has a text-based menu system that can be accessed using CTRL-A + Z. To set up minicom, type in the following:

```
$ sudo minicom -s
```

You can configure the serial port with the baud rate and other parameters.

11. Download and install GParted utility from the Ubuntu Software Center. This utility will help with partitioning disks.

With the system setup completed, the next step is building images. We will build three images for three different platforms. The first images will be built from the command line, the second will be built using the GUI tool, Hob, and the third image will tie together UEFI + BLDK booting of the operating system.

Conclusion

The next phase of system development, image building, is described in the book's next exercise. The open source templates, tools, and build system provided by the Yocto Project whose setup is described above offer an efficient, optimized design environment for developers. These tools provide unparalleled utility for new developers or those looking to migrate from Windows systems to Linux systems. Following the above steps will guide developers toward a solid starting point for creating cutting-edge, marketable, customized Linux systems for embedded products.

Open Software Stack for the Intel® Atom™ Processor *covers software development for embedded systems from the BIOS level up through operating systems, device drivers, and sets of tools, the latter of which are optimized for Intel's Atom line of processors. Liming and Malin introduce developers to a wide range of open source tool kits and building environments. The book will be invaluable for helping developers new to Linux set up a system, create builds, and run tests, laying the foundation for innovative development of embedded system solutions.*

This article is based on material found in the book *Open Software Stack for the Intel® Atom™ Processor* by Sean D. Liming and John R. Malin. Visit the Intel Press website to learn more about this book:

http://noggin.intel.com/intelpress/categories/books/open-software-stack-for-the-intel-atom-processor.

Also see our Recommended Reading List for similar topics:

http://noggin.intel.com/rr.

About the Author

Sean D. Liming has been involved with embedded systems for over 15 years. He started with Microsoft's first embedded distributor Annasoft (Annabooks Software) with products such as MS-DOS, Windows 3.1x, and Windows 9x before the introductions of Windows CE and Windows NT Embedded. Sean has authored over 35 articles and eight books including the popular Windows XP Embedded Advanced and Professional's Guide To Windows® Embedded Standard 7. He has traveled around the world as a featured speaker at Microsoft embedded conferences. Besides the books and articles, he has created the first classes on Windows CE, NT Embedded, .NET Micro Framework, and POS for .NET. He has also created advanced classes for XP Embedded and Windows Embedded Standard 7. Sean has worked at Intel focusing on XScale™ for telematics, and he was the head of engineering at Annasoft Systems. Sean is currently the owner of SJJ Embedded Micro Solutions, LLC. and subsidiary Annabooks. In 2002, he became a Microsoft MVP. He received his BSEE from California State Polytechnic University in Pomona, California; focusing on computer architecture and design.

John R. Malin was an early pioneer in using IBM-PC's to develop embedded software for x86-based embedded devices in the mid 80's. Over the past 20 years John has worked with a number of embedded operating systems starting with VRTX, Nucleus, PharLap, ThreadX, Windows Embedded OSes, and .NET Micro Framework. He has also co-authored a number of articles, white papers covering embedded development, and the book Real-Time Development from Theory to Practice. John is a cofounder of Annabooks, LLC and has a BS and MS in Solid State Physics from Case Western Reserve University.
