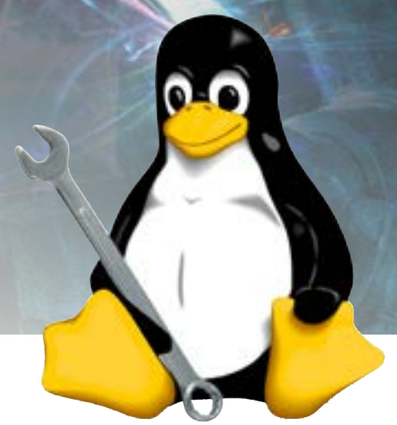
EMBEDDED IN THIN SLICES

Linux System Configuration (Part 3)

Build Your Linux System

This is the final installment in a multipart article series dealing with configuring Linux for embedded systems. In this article we will look at some of the ways available to you to build the embedded Linux system you need.

By Bob Japenga (US)



Fall is in the air and it is the time of the year I love best—especially in New England. Contrast that with building a new version of embedded Linux for a project. As a manager, I cringe every time one of my teammates tells me that they need to build (for the first time) a new version of Linux.

There were times when we designed a board from a reference design and built everything from a ready-made distribution from the chip maker. That was easy. A couple of quick cookbook-style steps here and there. Perhaps configuring a new flash memory chip or RAM chip. Compile. Build. Done. Simple. But then, at some point later, we discovered that our version of Linux didn't have the driver we needed for some piece of hardware or it didn't support some software functionality. This recently happened to one of our customers who delayed implementing a software function until late in the project. When we were asked to implement it, they already had special drivers that only worked with their version of Linux to interface to their proprietary hardware. But they needed a streaming video USB gadget that wasn't in their version of Linux. We had a dilemma. We had to either bring all of the proprietary code over to a new version of Linux or back-port the code from a later version of Linux into the older version. It was time for the antacid. There was no easy answer. We knew cost overrun and schedule slips were imminent.

This can happen, and it is painful. Very often, the chip manufacturers don't mainstream their changes (i.e., put them into a stable release at Kernel.org). That can be understandable in the fast-paced world of hardware development. If asked, they will say that they haven't done it *yet*. If you have to bring that code into a new version of the kernel, you will need to bring all of their changes that they made to get the reference design running into the new kernel. One time we used a reference design and very late in the life cycle discovered a bug in the C library. The manufacturer not only didn't mainstream their changes, they couldn't find the source code they used for the c library. What a nightmare.

These days we are smarter and things are a little easier. Thanks in part to OpenEmbedded, Yocto, and Ångström. This month I'll cover each of these. We have used all three. There are other options, like Linux From Scratch, but since we are only examining it in thin slices, I am only going to cover the slices we have actually used.

OPENEMBEDDED

In Part 2 of this article series, we discussed the concept of a Linux distribution. These are things like Ubunto, Fedora or, in the embedded world, Ångström. A distribution is a complete set of tools, the kernel, the packages, and the build instructions to create a version of Linux. OpenEmbedded provides you the framework to enable you to create your own distribution. From the name, you will gather that it specializes in creating an embedded version of Linux, which is very different than starting with a distribution like Ubunto.

The problem for us as designers is how to create a distribution for our project. Over the years, we have seen a number of attempts to help us. The OpenEmbedded project was a big step forward in this process. I won't bore you with the history of how this evolved, but the major contribution that the OpenEmbedded organization provided to our community was the development of BitBake and the structure for metadata used to define your system. Let's look briefly at each of these.

BITBAKE

In the past, developers have used makefiles to build the Linux kernel and the associated applications. The kernel was configured as discussed in my June 2014 article, in which I described using script files to tie the whole process together. We used this method on many projects early in our deployment of Linux. Once in place, it worked okay. The problem came when we had to make changes to the version—of either the kernel or its associated libraries or the applications.

Another problem had to do with the tool chain. The tool chain needs to be tied to any development life cycle or when you start a new project. Only once have we used the same build process from one Linux project to another. And that was because the hardware was identical. This is a significant problem. The OpenEmbedded organization was formed to help us with that. What OpenEmbedded brought to the table was a new tool and specified a new language to define the architecture. That tool was BitBake.

Think of BitBake as make on steroids. The inputs to BitBake are called "Recipes." BitBake takes recipes and metadata to create a complete Linux distribution. It is not necessary to have a build script (i.e., write code) to create most distributions. The language is powerful enough to do everything you need. And if it has shortcomings (they all do), it does allow you to use Python to create more complex operations. It is flexible enough to handle multiple hardware architectures and helps you to manage and create multiple releases for those targets.

METADATA

So what is this metadata that BitBake uses? Think about what you need to know to create the Linux kernel. You need a configuration file. You need to know where your tool chain is to compile and build the kernel. You need to know where the source files are located. Ideally, they should be available on the Internet. You need to know complier options. And you need to know a series of dependencies (i.e., if you change this file, other files need to change). Building a package has all of the same requirements. All of this gets wrapped up in what is called the metadata.

RECIPES

Okay, you are saying: "This is sounding a little cheesy." Baking. Recipes. What are these recipes? If BitBake is make on steroids, recipes are makefiles on steroids. A given recipe tells BitBake exactly the who, what, where, and how for building a particular package. Who is the name of the package. What is the type of package. Where is the location. And then How to build it. The end result is a file system image and a kernel image that you can put on your target system.

YOCTO

Wow! That sounds great. But what's lacking? Why did Yocto need to come along? Basically, Yocto picks up where OpenEmbedded left off. More heavily funded, it accomplishes all of the same goals that OpenEmbedded set out to attain. But Yocto relies heavily on two things from OpenEmbedded: BitBake and the metadata. OpenEmbedded remains a separate organization from Yocto and is concentrating on making BitBake better and providing metadata for a host of architectures. Yocto is concentrating more on the tools and the board support packages (BSP) for a wide variety of architectures and processor boards. For example, OpenEmbedded maintains the metadata for something like the BeagleBoard and the Raspberry Pi boards, but the Yocto project is the place to get the BSP.

Another way to look at Yocto is to think of it as a place where silicon providers like Atmel, Texas Instruments, and Freescale can create a common framework for the release of Linux for their new processor or core. For us as users, that is a win-win situation.

Another benefit for developers, is that if, for example, we choose to use a commercially supported release of Linux (Bob-Are words missing in sentence?). Perhaps we need to meet the FDA requirements for Software of Unknown Provence (or Pedigree) (SOUP). Yocto helps because all of the major vendors are members of Yocto. So, if you do all of your development on a particular Yocto version and then need the kind of support

ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

that these commercial packages provide, you can seamlessly switch. This is not true if you use just any old distribution. We found this on one project when we evaluated going to a commercially supported version of Linux from Wind River. We were evaluating this because of the processes and documentation that they provided for FDA approval of a medical product. Going with the commercially supported release was going to save us considerable time and money for documentation. The problem was that we used a version of the kernel from Texas Instruments that came with our development kit and that version was not supported by Yocto. Creating a new distribution for the closest version supported by Wind River would probably cost more than the benefit received by having the documentation and support provided by Wind River.

ÅNGSTRÖM

If all of these names haven't gotten your head spinning, let's just talk about one more. Ångström heavily relies on OpenEmbedded and Yocto to create a distribution specifically targeted for embedded systems. It is highly scalable. It can run on a device with as little as 4 MB up or on a system with several terabytes. But of course the question is, what can it do with only 4 MB? There are a host of off-the-shelf boards that it supports including the popular BeagleBoard and Beagle Bone open source hardware platforms as well as the Raspberry Pi. So, if you are using a reference design or an SBC that is supported by Ångström, you will probably have the most tools and flexibility for maintaining your design throughout its lifecycle.

STICK WITH LINUX

Where are we to begin as system designers? We are offered a plethora of choices. Let's consider them in order of decreasing complexity resulting in decreased cost and time: we can certainly build Linux and the subsequent applications from Kernel.org; we can use a tool like "Build Linux from Scratch"; we can use a board-support package provided by a hardware vendor that is not part of Open Embedded, or from one that is part of Open Embedded; we can build our design using a project already supported by Yocto; or we can start with an existing distribution (Ångström, or Ubunto, or Debian) and go on from there. Our experience is that the cost savings across the life cycle are significantly larger the later we go in that list of options. A few years back, hardware venders of x86 SBCs could tout simplicity by providing an SD card or Compact flash with Ubunto or Debian on it. Everything was there for you. You could concentrate on your killer application.

Today, with Ångström, you can have that for a wide range of processors and for specifically an embedded system. If you are rolling your own hardware based on a reference design, make sure the silicon provider is using OpenEmbedded and ideally has created a Yocto project.

We have come a long way in developing easier ways to create, modify, and maintain embedded Linux systems. We still have a long way to go to achieve the full potential that Linux offers. Sometimes, the very issues I covered in this article make me want to stop using Linux. But then where else would I go? Hopefully, this article series has helped you better understand how to configure the kernel, select and build the applications, and finally put together a distribution for your next embedded Linux project. Of course, only in thin slices.

circuitcellar.com/ccmaterials

RESOURCES

Ångström, www.angstrom-distribution.org
Fedora, https://fedoraproject.org
Ubuntu, www.ubuntu.com
Open Embedded, www.openembedded.org
Linux from Scratch, www.linuxfromscratch.org
Wind River Linux, www.windriver.com/products/linux.html
Yocto, www.yoctoproject.org
BeagleBone, http://beagleboard.org/bone
BeagleBoard, http://beagleboard.org
Raspberry Pi, www.raspberrypi.org