by Bob Japenga (USA)

"Embedded in Thin Slices" is a new bimonthly column devoted to helping professionals hone their embedded programming and design skills. Topics such as embedded Linux and how to deal with concurrency issues will be covered.

Getting Started with Embedded Linux (Part 1)

When to Choose Linux for Your Embedded Design

This three-part series introduces the topic of embedded Linux. Why should you choose Linux? To begin with, it can easily add features, devices, and functionality—many of which are already built into the operating system. Read on to find out more.

In Malcolm Gladwell's bestseller *Blink* he talks about the way we make decisions. In particular, he describes a process our brains go through called "thin slicing." Thin slicing describes our ability to find out what is really important with limited data. I have been designing embedded systems for 38 years, yet I find I am always a beginner. I always have limited data and a limited vision of what is coming next. The field is so broad and the applications so varied, that no one can fully understand even a small portion of the field. In this and subsequent columns, I'll write about some thin slices of my experience in the field of embedded systems design.

For the next three columns I'll share my experience with embedded Linux. This month I focus on choosing Linux for embedded applications. The next two columns will discuss choosing a platform, a tool chain, and licensing issues.

RTOS FOR EMBEDDED SYSTEMS

When do you need a real-time operating system (RTOS) for our embedded system? You always need to be careful with the tools on your tool belt. When you have a great hammer, it's amazing how many things start looking like nails. Using Linux in embedded systems can be that way. It's so powerful, once you have invested the time to learn how to deploy it (and that investment is not cheap if you port your own version of the kernel—more on that next time) and know how to do it quickly, everything starts looking like a Linux target.

Perhaps a broader question needs to be answered first: When do you need an RTOS in your embedded design? The two thinly sliced answers I have found are: when you have more than two or three concurrency requirements. Or, when you have (or even may have) a number of complex generic devices that require device drivers or when you have one or more complex generic functional requirements.

CONCURRENCY REQUIREMENTS

Basically, concurrency means that you have to do two things at the same time—concurrently. These can creep up on you unawares. Certainly, we can all recognize the apparent ones. For example, a system that has two or more independent control loops is the most obvious one. But anytime you talk to interfaces that have variable response rates with significant delays you have added a concurrency requirement. For example, let's imagine a simple device with a simple TCP/IP stack. The single-thread user interface makes a call to "ping" a device to see if it's there. The stack may take anywhere from 1 ms to 20 s to return. How many of us use inane user interfaces daily that force us to wait the 20 s before the network times out—even to abort the call? That kind of interface puts a concurrency requirement on

your system.

Our company does a lot of projects with only a few concurrency requirements. These are nicely handled with a foreground/background scheme or a single main loop, using a few interrupt handlers for the concurrency. No RTOS is needed. But I have found that once you have a user interface and one or more slow interfaces, some sort of threading/multitasking is essential to keeping the design and user interface clean.

> "When you have a great hammer, it's amazing how many things start looking like nails.Using Linux in embedded systems can be that way. It's so powerful, once you have invested the time to learn how to deploy it, everything starts looking like a Linux target."

DEVICES & FUNCTIONALITY

One of the other things that drives us to choose an RTOS is the need to interface with a generic complex device. By generic, I am not talking about devices that are unique to our proprietary system. I'm talking about devices that have general utility, such as USB sticks, cameras, cell modems, wear-leveling flash memory, CODECs, PHYs, or Wi-Fi devices, for which device drivers have already been written. We simply cannot take the time to design the software drivers for the vast array of devices that some of our embedded systems need to talk to. Many RTOSes provide a wide assortment of these generic drivers to simplify the task of interfacing to them.

Similarly, sometimes our requirements contain requests for wheels that have already been invented, such as encryption and compression algorithms, mesh networks, remote access, and remote control. We design a lot of systems that often start with a small device and function complement. But user demand forces us to add devices and functionality we never dreamed of long after the design is complete. This is where a full-fledged RTOS like Linux pays for itself over and over again. These already invented devices and functionality are added for low-entry cost and we can concentrate on the problem domain we are trying to address.

Allow me to list a few of the devices and functionality that got added on to some of our designs long after the first release. These were features for which we had not given a minute of forethought to in the original design.

The first feature was a full keyboard. A customer wanted to expand his user interface from our keypad to a full QWERTY keyboard. Solution with Linux: No change. Plug a USB keyboard in. No drivers to write. No queues to change.

The second feature was e-mail capability. A customer wanted to alert maintenance people with an e-mail rather than a fax. Solution with Linux: Add an existing open-source mail program and a simple interface to add the e-mail addresses.

The third feature was remote display and control. A customer wanted to be able to remotely control a custom QVGA display with a custom keypad from his PC. Solution with Linux: An open-source Linux virtual network computing (VNC) library enabled the customer to remotely control and monitor this custom user interface with little effort on our part.

The fourth feature was a cell modem. A customer wanted to provide Internet connectivity in remote sites that did not have Internet service. An open-source Linux ppp daemon was easily configured to talk to a large number of cell modems and networks.

A fifth feature was proxy settings. Networking is a big reason we choose Linux. Anytime a system requires flexible, secure, and expandable Internet connectivity, Linux is our choice. With Linux, we get everything we need (and usually more) for safe and secure networking. We had fully deployed a system when, late in the game, our customer discovered that they needed proxy capability for some of their customers. With Linux, the answer to our customer was: "No problem." It's built in. We didn't think of it—but we didn't need to.

Without question, it costs us less time to add these features (as well as many others) to our customers' products because we chose Linux as our RTOS.

If we have these needs, why choose Linux? Certainly the concurrency requirements can be met with a wide range of available RTOSes. One of the hottest items of late is the FreeRTOS project, which offers an RTOS ported for more than 27 architectures. If you require concurrency but not complex generic devices or complex generic functionality, then Linux is probably more complex than you need.

A LINUX ROTS?

Before we leave the topic of concurrency, I need to address the real-time nature of the operating system (OS). I suspect that some of you may be biting your tongue when I describe Linux as a real-time operating system. If by that you mean an OS that must meet hard deadlines in the sub-millisecond range that can never miss a deadline without extreme consequences, then you are correct. Various successful attempts have been made to add real-time extensions to the Linux kernel. But out of the box, Linux does not meet that strict requirement. Basically, the two terms of interest are deterministic scheduling and low and deterministic latency. If there is interest, I can talk further about this in another column.

An example of a simple hard deadline real-time requirement is a generic SPI or I^2C slave device (assuming that your processor doesn't have built in support for it). In the latest Linux kernel, you will be hard pressed to find a driver to support this seemingly simple interface. The reason is that Linux can't meet the timing requirements the host would demand of such a

device. Although the master can slow the interface down, the device is a slave to the master's clock. If you don't control the master, your driver will sometimes fail to deliver.

We get around this in our designs by putting simple microcontrollers (less than $.50) in front of Linux to meet any hard deadlines. It's not worth it to try and bend the OS and to use real-time extensions to make it meet such rigid demands. For the most part, doing it in hardware is inexpensive and, more important, easily verifiable.

That said, we had a requirement for Linux to process interrupts with a latency of less than 10 to 20 µs. Our initial design had a PIC microcontroller in front to handle this. However, since the requirement wasn't "hard" and we could live with a retry, missing one in every 200 of these events wasn't a big deal. So, in production, we eliminated the PIC front end and are handling it just fine in Linux—most of the time. This is why I call Linux a real-time operating system. You can design your application and your drivers with very low latency. Determinism is the problem.

> "My thin slice tells me that Linux offers more drivers for more devices and more complex generic functionality than any OS on the planet with the exception of Windows."

So, in that sense, Linux can handle some pretty rigorous real-time requirements. In addition, the Linux community has done a lot in the last five years to add real-time features into the kernel. But interestingly enough, we have not had a need to utilize most of these features as of yet.

AVAILABLE OPTIONS

If you need concurrency, you've settled the hard deadline issues (either from a specification point of view or by hardware), and your system has one or more complex drivers or generic functions, what are your options? One solution is to roll your own. Certainly this is an option, but I cannot think of one reason to justify it. A second solution is to use a commercial off-the-shelf (COTS) OS. Certainly many of the fine COTS OSes offer a plethora of complex device drivers and complex generic functionality as part of their offering. We have used QNX, Microsoft Windows CE, Wind River VxWorks, Green Hills Integrity, and Mentor Graphics Nucleus on a number of systems, and all offer excellent tools, generic drivers, and much generic functionality. The cost (both recurring and nonrecurring) for some of these is staggering. A third solution is to use Linux (we'll talk about some of the flavors next time). The compelling factor for us to choose Linux was not entry cost. There were two driving factors: Availability of complex drivers and complex generic functionality we never dreamed of and availability of the source code when things don't work.

DRIVERS & FUNCTIONALITY

Our customers and their customers are constantly demanding new features. New features require new devices and new devices require new device drivers. It is my thin slice that new Linux drivers for new devices and new functionality are available more quickly than from any other option other than Windows (drivers and functionality for Windows CE is another question). When my customer demands SSH instead of telnet and my RTOS vendor only offers telnet, what are my options? I don't have enough pull to make them change. Do I port SSH to the COTS OS? Do I lose the customer? None of these options are pretty.

Furthermore, if the device requires me to write the driver, Linux again wins hands down in terms of ease of development. Although I would say the documentation is harder to find, there is so much released source code that I can usually pattern my new driver after something similar without completely reinventing the wheel.

Without question, my thin slice tells me that Linux offers more drivers for more devices and more complex generic functionality than any OS on the planet with the exception of Windows. Here is where the issue of entry cost is not the primary concern. How much is it worth to be able to quickly and inexpensively add these features and remain competitive? This is where the real cost savings of Linux works in our environment.

SOURCE CODE AVAILABILITY

Finally, support provided by the COTS is great theoretically. But we have uncovered bugs in four of the five previously mentioned players and gotten stuck. Not one of them was ever fixed by the supplier. These are well-designed OSes, so when we uncover a bug it is usually very obscure and random. How can we get them to duplicate it? They need our hardware and our application. Many times we can't even prove that it's in their code and not ours. Without the source code, we're just plain stuck. We fully understand why, in their business model, they cannot release the source code. But for this reason alone, we choose Linux for the kinds of systems we build.

A THIN SLICE OF LINUX

When our projects require concurrency and flexible and extensible functionality, my thin slice of reality says Linux every time. I would love to hear from you. What has been your experience with embedded Linux or with the other major players?

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 150 [Bob, Is this number correct?] 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.

RESOURCE

The FreeRTOS Project, www.FreeRTOS.org.