Feature Article

Embedded Linux on PC/104

By Eric Finster

There's embedded Linux and there's embedded Linux. The Linux strategy used for a shirt-pocket sized MP3 player won't necessarily fit in the PC/104 world. And guess what? RedHat and other commercial releases lean toward the MP3, PDA, cell phone, etc. side of embedded. So what's a PC/104 system architect to do? This article will look at the Linux beast from a definitely "PC/104 angle." It will ask, then answer, a number of critical questions related to application, development environment, tool chain, drivers, and applications.

In a relatively short amount of time and for a number of different reasons, the Linux operating system has become the first choice of many embedded systems developers. First, it provides a much richer set of features than would typically be available in either a custom software environment or a DOS-based system. Second, since it is open-source and freely available to the public, it can alleviate some of the costs associated with many commercially available real-time operating systems (RTOSs). Finally, its UNIX flavor gives it a standardized operator interface and programming API, making it an excellent mid-range solution for projects that need a fair amount of "software horsepower," but wish to avoid the cost of a third-party RTOS.

The list of features provided by the Linux kernel is impressive and is growing all the time. In addition, with the sheer volume of freely available software for Linux, putting together an embedded system with a remarkable amount of functionality and connectivity can be quick and painless. Features such as embedded Web-servers, GUIs, or remote administration can easily be compiled and installed once the system is up and running. However, many developers lured by the promise of greater functionality in a shorter amount of time, are left wondering just exactly what an embedded Linux system will look like. Should I just install a desktop distribution on the target? What about development tools? How much hardware support will I need to run Linux?

This article will attempt to answer some of these questions and in the process, hopefully, give you a feel for how Linux looks in an embedded application. As you will see, Linux is a remarkably configurable operating system, and taking the time to get the right environment set up can really streamline the development process.

When do I need Linux?

When is the right time to use Linux in a particular embedded application? This is the common question for developers new to Linux.

An application that consists of mostly custom code of low complexity will be unlikely to benefit much from moving to Linux because the overhead of booting the kernel and setting up the system will unnecessarily complicate the application. For example, an application that just responds to incoming serial requests by toggling some digital I/O lines would not be a good candidate for using Linux. On the other hand, if your application requires the use of any major standardized software features, like TCP/IP, GUI support, or multitasking, Linux may save a considerable amount of time and money.

Another important factor is the hardware you intend to use for the application. A typical x86 Linux system will require at least a 386 processor and 8 Mbytes of DRAM, not to mention a place to store the file system. Some commercially available embedded Linux distributions have kernels that will run in more resource-constrained environments, but if the above requirements are too steep for your application, Linux might not be the best fit.

Linux is quite flexible when it comes to user interface requirements. If your hardware has a monitor and keyboard, then Linux will come up right out of the "box" and you will be able to log in to your system and run commands. If you are operating in a headless environment, you can still use Linux over a serial port, or over Ethernet (via telnet). One of the most remarkable and wonderful things about Linux is, to the system, all three of these environments are essentially the same. In fact, the shell program, the program that reads user input and executes commands, is the exact same program no matter which setup you use. The end result is that you can spend your entire development phase with a monitor and keyboard attached, and then reconfigure the system to run over the serial port for production purposes, and the interface over the serial port will be the same. You will see how this works later during discussion of device drivers.

How should I set up my development environment?

Once you've decided to use Linux for your application, the question of how to get the system onto your embedded target quickly arises.

There are two major factors that distinguish between the number of different approaches to loading Linux onto an embedded target. These factors are: where the Linux kernel resides, and where the file system resides. This is because Linux has essentially a two-stage boot process. First the kernel comes up and initializes all the hardware and driver support, and when it has finished, it mounts a file system known as the root filesystem. Once the root filesystem has been mounted, the kernel runs a special program called init, which is responsible for running any initialization scripts, spawning system services, and controlling the system run-level.

This first approach is what I call a standalone system (Figure 1). This is where the embedded target has enough functionality to look like a complete PC, and a standard desktop Linux distribution can be installed on it directly. In this case, both the kernel and the file system reside on the same drive, which is supported by the target, typically an IDE disk drive. However, there can be some drawbacks to this approach. First of all, most desktop distributions are pre-configured for quite a bit more functionality than a typical embedded system will require. This can lead to slow boot-up times and sometimes even slow system performance since loading unused drivers and services consumes system memory. Also, desktop distributions distribute kernels, which are config-

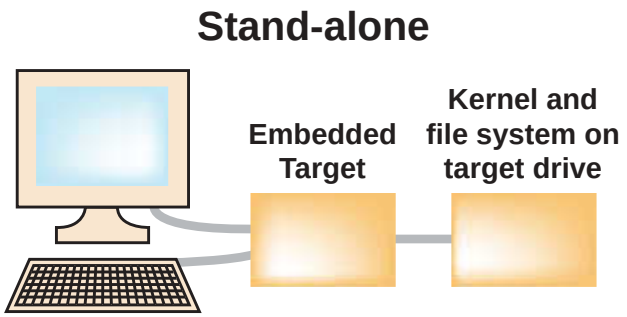

Figure 1

ured to support every possible device since they are meant to be compatible with a wide variety of desktop hardware. Eliminating these unused drivers and services can save a considerable amount of system memory.

There are a number of commercially available embedded Linux distributions (Hard Hat Linux, Blue Cat Linux, and so on), which are stripped down for embedded use, and, most of them also support standalone configurations where the system is installed directly on the target. They may also support one or more of the advanced booting mechanisms listed below, which address the problem of diskless and headless targets.

The next approach is the network-boot approach (Figure 2). In this configuration, a network boot ROM is used to load a system image off the network when the board is powered on. Notice that in this case both the kernel and the file system reside on a remote machine, and whatever hardware mechanisms are in place for

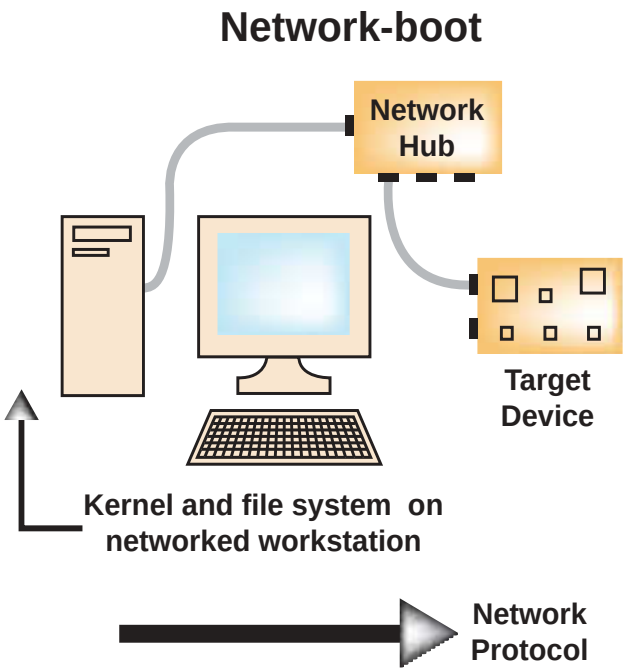

Figure 2

booting from the network take care of transferring the kernel to the board. This approach will also make use of the Network File System (NFS), which is described below. Many of the commercially available embedded Linux systems have support for generating custom boot ROMs designed for various embedded targets and are tailored to work with their particular setup.

Finally, there is what I call the hybrid approach (Figure 3). In this case a Linux kernel is booted directly by the target hardware, and then the kernel seeks out its file system over the net-

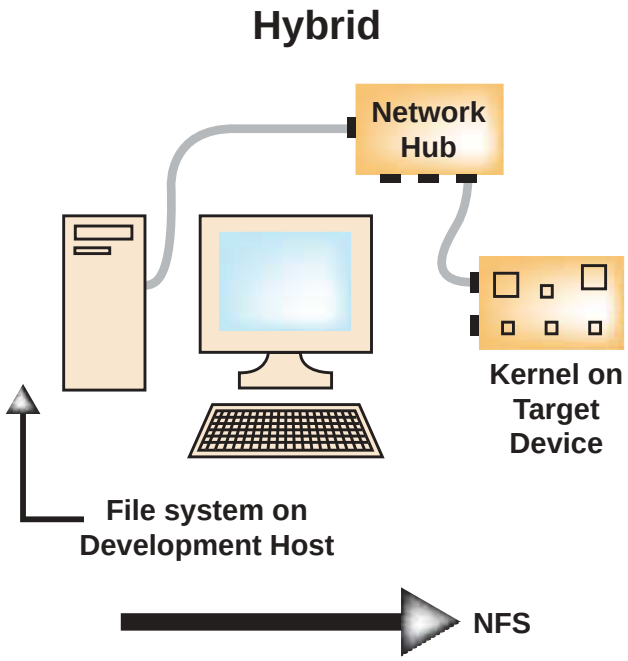

Figure 3

work using NFS. The kernel can be booted on the target board by either using a boot floppy, or, if the board supports it, by putting the kernel directly in Flash or some other permanent storage device. If you get your Linux system as part of a package from a PC/104 board-level manufacturer, this is a common approach, since the manufacturer can pre-configure a kernel that supports the board's features. Micro/sys, for example, makes many of its boards available with a Linux kernel pre-installed in Flash so that you can boot the target immediately into the Linux environment (Figure 4).

Figure 4

The advantage of the hybrid approach is this: the target board's file system will appear as a subdirectory on your desktop machine. This means you can manipulate the configuration of the target just by editing the files in this directory, with all the comforts of your desktop machine. Additionally, if the target system uses the same compiler as your desktop system, you can develop your application, compile it, and just copy it to its final location. This setup provides the most flexible development environment for embedded targets, since it scales down to the smallest devices and up to full-scale systems.

When the development phase is complete, configuring the board for production use can be as simple as issuing a copy command that transfers all the files from the desktop system to some permanent media device mounted on the target. Micro/sys provides

Embedded Linux on PC/104 cont.

an automated script that will copy all the files to a Disk-on-Chip and make any necessary configuration changes to boot that device.

What development tools will I use?

Though it can be a bit intimidating at first, the GNU C-compiler (gcc) and associated tool kit are one of Linux's greatest strengths. The compiler produces fast, efficient code and can be configured as a cross-compiler for a wide variety of target systems. In most cases, the embedded Linux distribution will include some version of gcc already configured for the correct target platform. The GNU C-compiler also has an associated debugger called gdb, which can be configured for remote use, rounding out a complete set of embedded application development tools.

There are many freely available and proprietary Integrated Development Environments (IDEs) available for Linux and gcc as well. These programs will let you write and build (and sometimes debug) applications in a comfortable graphical environment, which can be a great help if you are coming from a Windows or DOS background. On the other hand, as many Linux gurus will tell you, learning one of the more common Linux text editors such as vi or emacs will be well worth the effort if you plan to get deeper into customizing your Linux system.

What kind of drivers will I need?

Since Linux is a multiuser, multitasking environment, the kernel cannot allow arbitrary access to hardware devices. Thus, all hardware requests are sent through Linux's device driver layer. Devices are represented by files in a special directory /dev. When an application program opens a device file, it is like opening a channel to the hardware device that owns that file. By reading or writing data to the file (using the standard fread() or fwrite() functions, or any other I/O functions for that matter), the application program can communicate with the hardware device.

Routing all hardware functions through these device files allows the Linux kernel to handle sequencing and simultaneous access. It also adds an added dimension of abstraction to the system as a whole. As I mentioned above, the same shell program can be run over the virtual console (that is, monitor and keyboard), the serial port, or the Ethernet port via a telnet session. This is because the shell only has to know how to read and write from a device file and doesn't care what kind of device file it is using. Though it might seem counter-intuitive at first to have all the hardware accesses go through a file, you will find, after playing with the system for a while, that it simplifies many programming tasks immensely.

Each device file is created in a way that allows it to talk to a specific hardware driver inside the kernel. Drivers exist for many of the hardware features you will find in most desktop systems (for example, keyboard, video, serial ports, and disk drives) but they may not exist for some of the more exotic features found in embedded systems, such as digital I/O or specific data acquisition cards. If you find this is the case for some hardware feature you wish to use, you might consider writing a driver yourself. There are many freely available guides on the Internet that show you how to do this. Although, if the I/O task is simple enough, you can use the ioperm()/iopl() functions to allow your program to access I/O ports directly. You should be able to find information on these functions in the main pages of any Linux desktop system.

How will I get my application going?

Linux was originally designed to be a multiuser operating system. This kind of configuration can sometimes be overkill in an embedded system where only a single application is meant to run when the system starts. As I outlined above, after the kernel has finished booting up, it transfers control to an executable program on the file system called init. In a desktop system, this program will go through a set of initialization scripts to configure the system and launch any services that are required, and then provide the user with a login prompt.

For an embedded system, you will probably need to configure these scripts (typically stored in /etc/rc.d) to launch your application. Some embedded distributions may automate this process for you by allowing you to specify a single executable file as your application program, and then generating the appropriate configuration scripts accordingly. Also, you may wish to read about the init program and its configuration file /etc/inittab, since for your final production environment, you will need to design a startup mechanism that handles launching any services needed by your application (system logging, for example), launching your application itself, and handling error conditions, such as when your application exits abnormally.

What does it all mean?

Linux can be a terrific solution for embedded applications that depend on a high degree of software support or that make use of standardized software protocols or packages. Like anything else in life, Linux is not a free lunch, as it requires a considerable investment in designing and implementing a system that will work for your application. If you are a newcomer to Linux, it is highly recommended to first get your hands on one of the many desktop Linux distributions, install it, and play with the system for a while before you embark on an embedded application. Make sure you are first comfortable with the various tools and utilities that make up the system.

Also, remember that when you move to a more robust operating system like Linux, you are trading time spent developing application code for time spent configuring the system as a whole, of which the application code is just one part. I have tried to outline some of the configuration challenges you may face when moving Linux into the embedded space, but the success of the project will depend upon viewing Linux as the sum of its parts: your application, the drivers, the kernel, and the files and utilities in the file system. When seen in this light, Linux is one of the most flexible and robust systems available today.

Eric Finster *is a Software Engineer with Micro/sys. Micro/sys has been deeply involved in the board-level OEM computer market since 1978. Applications assistance on Linux, VxWorks, DOS, and other operating systems – especially their impact on PC/104 systems – is a key area of customer support. Micro/sys is located in Montrose, CA, just north of Los Angeles. Eric can be reached at: efinster@embeddedsys.com*

Micro/sys, Inc.
3730 Park Place
Montrose, CA 91020
Tel: 818-244-4600
Fax: 818-244-4246
E-mail: info@embeddedsys.com
Web site: www.embeddedsys.com

Resources

For further information about embedded Linux visit:

LinuxDevices Web Portal for Embedded Linux
http://www.linuxdevices.com

Monta Vista Embedded Linux Distribution
http://www.mvista.com

LynuxWorks Embedded Linux Distribution
http://www.lynuxworks.com

Red Hat (Embedded) Linux Distribution
http://www.redhat.com

Micro/sys Single Board Computers
http://www.embeddedsys.com

Reprinted from PC/104 Embedded Solutions / Summer 2002