

# most significant bits

News and Information For High-Tech Professionals

Published by Pocket Protector Press™

A division of Stout Systems



Summer 2006

## In This Issue

### Employment Agreements

What is typical? What is excessive? A quick overview of the most common practices.

### Overview of Embedded Linux, Part II, A Short HOWTO

Walk through the development of a specific system, paying special attention to tools, development issues and the steps for taking a general distribution of Linux and configuring it for a target systems.

### Recent News

Stout Systems welcomes new employees Rick Pluth and Bill Whelan (Software Engineers).

### Job Openings

Contract opportunities still strong and permanent employment openings are surging. See our current openings at our Web site [StoutSystems.com](http://StoutSystems.com).

### Subscribe To Our Newsletter

If you would like to receive this newsletter directly at your desk or in-box, send a note to [info@stoutsystems.com](mailto:info@stoutsystems.com) providing your subscription info (e-mail or ground mail address).

**Call us:**  
(734) 663-0877

Read and understand before you sign

## Employment Agreements

by John Stout

I read an interesting article in which the author discussed his experiences as a consultant in the IT industry. In particular, he detailed the “terrifying provisions” that one consulting company attempted to get him to agree to.

I thought it would be worthwhile to give a brief summary of what is now more-or-less standard in the industry as far as employment agreements are concerned.

**“At-will” employment:** This means that employer or employee can terminate employment at any time for any reason, with or without notice. This is the typical employment relationship, with no employment contracts or guarantees.

**Background checks:** Extensive checks are common, including checking criminal records, driving records, terrorist databases, doing drug screenings and, in some cases, even credit checks. There should be some limiter in the agreement as to who will be allowed to look at the results of background checks—typically the firm doing the screening and the firm’s customer who requested the screening.

**Non-disclosure:** Firms want to protect confidential information. It is important that you understand what your employer considers confidential. It may include proprietary technologies, customer information, pricing, payroll, plans and internal processes. There should be limits on the time and circumstances of the disclosure of such knowledge.

**Non-compete:** Former employees who



have knowledge of the firm’s customers, vendors, proprietary technology and other information could use that knowledge competitively against the company. A non-compete specifies that an employee or consultant may not work for one of the company’s competitors for some period of time after leaving. Non-competes for consultants also exclude them for working directly for the firm’s customers. In both cases, the duration of the non-compete is typically one to two years, after which the firm has to yield to right-to-work employment statutes, which can vary from state-to-state.

Individual companies may have other clauses and provisions in their employment agreements, so if you encounter anything that seems excessive consult your attorney before signing.

**John W. Stout** is the founder and president of Stout Systems Development. He has twenty-five year’s experience in the software industry. He is also sought after as a technology speaker, presenting sessions at developer conferences and user groups.

## Overview of Embedded Linux, Part II A Short HOWTO

by David B. Rein

*Editor's Note: In the Spring 2006 edition of our newsletter, we featured an article by Dave titled Overview of Embedded Linux. If you missed the article and would like to read it, it is available online at [www.stoutsystems.com/newsletter-spring06.htm](http://www.stoutsystems.com/newsletter-spring06.htm).*

**I**n the Spring 2006 newsletter I explored several of the advantages and issues of using Linux for embedded systems. I thought it might be helpful to walk through how I developed a specific embedded Linux system, paying special attention to tools, development issues and the steps for taking a general distribution of Linux and configuring it for a target system.

For purposes of illustration, I'll discuss a target system that automated the control of building lighting. Large buildings frequently use automated lighting systems to turn on and off the lights in various zones. Think of how tedious it would be to run around flipping switches every morning and then again every evening. Now think of how convenient it would be to program lights to turn on at 6AM, to turn up at 6PM and to turn off at midnight.

The lighting control system used an inexpensive, small-footprint X86 motherboard with on-board Ethernet, USB, RS-232 serial, ATA disk interface, PS/2 keyboard and mouse ports and video. The video and keyboard were used only for development and debug.

The development path for this project took advantage of the fact that the motherboard had the features of a normal PC system. The tools that I selected worked in both Windows and Linux. This allowed the development steps to be:

- Step One: Start application development in Windows.
- Step Two: Develop device drivers in Linux.
- Step Three: Port application to Linux.
- Step Four: Port application to target hardware.
- Step Five: Create embedded version of Linux.

The choices that we made at the beginning of the project—the use of Linux, a full-featured operating system, the use of an X86 motherboard, and the use of tools that run on both Windows and Linux—made it possible for us to design, develop and review an application on Windows long before we had to worry about the final embedded target.

Step One, application development, was done on a

Windows box using the Cygwin port of the GNU tools. Application development was completed long before we worried about anything having to do with hardware issues, operating system or device drivers. In this step I was able to test module function and inter-module communications. The interfaces to device drivers were stubbed out. The design partitioned major functionality into separate programs and used the TCP/IP stack for inter-process communications.

Step Two ran concurrently with Step One. Here I developed the USB serial port driver on a Linux box. The manufacturer of the serial chip provided open source device drivers for Linux. I just needed to rework the drivers to provide the functions need for this project. The driver development was significantly eased by the fact that the whole Linux kernel is open source, making it, in effect, the DDK (driver development kit). I was able to examine all modules of the USB driver stack and gain insight into the needs of my driver.

An important feature of Linux is that most device drivers are separate modules that are dynamically loaded and unloaded. The kernel does not have to be changed or rebooted just to modify a driver. A note here for those familiar with the Windows Blue Screen of Death (BSOD): the first bad memory access by my driver resulted in all the information from the BSOD being dumped to the screen while the system continued to run. All I had to do was unload the driver, modify, rebuild and then reload the driver. In the course of driver development I only had to reboot about a dozen times due to the driver hanging, but never due to memory access.

Step Three was to port the application to Linux. This required setting up a development environment on a Linux box, building the code and testing it for correct operation. Since the source code was already built with the GNU tools, there was very little editing needed to build in the Linux environment. Most of the effort was modifying makefiles. I was able at this point to test the input and output processes which used my custom device drivers. This procedure was very similar to Windows development. I used Visual SlickEdit on both Windows and Linux, which provided a consistent IDE environment.

Step Four was porting to the target system. This was done in two parts. First, I installed an unmodified version of Linux onto the target system, which included a disk drive and CDROM. With this configuration I was able to test the application and device driver, especially checking for performance and memory issues. The second part was to reduce the footprint of the storage and memory usage.

Finally, Step Five. I created a read-only FLASH based system.

It is at this point that one is faced with the sculptor's dilemma: the block of marble holds the work of art, giving the sculptor the problem of removing only the unnecessary stone. With Linux there is tremendous choice of kernel modules and utilities. Often there are multiple packages to accomplish the same function, so it is important to understand the packages and choose the best for the system requirements. One path was to use a Linux distribution configured for a minimal footprint such as the Linux Router Project or Damn Small Linux. I found several weaknesses in the minimal pre-configured systems; they used older kernels, didn't include some features needed for the target system, and were not compatible with many pre-built packages. The other path was to start with a full distribution, such as Debian or Fedora, and then whittle it down to meet the storage requirements. I chose the latter path using the Debian distribution since it is widely supported, well tested, and provides the latest kernel and a vast number of pre-built packages. It is also what I was using for my development system so it simplified the port to the target.

The two main tools needed to configure the target system are menuconfig and a package manager. Menuconfig is the part of the kernel build process and allows configuring the kernel and device drivers. For Debian based distributions the utility, Aptitude, manages the installation and removal of utility packages. Using these tools brought the storage footprint down from 1GB to under 400MB. Further removal of unnecessary files and drivers completed the sizing to under our target of 256MB.

The next step was to make the system read-only. This

required the following:

- identify those files and directories that needed to be writable
- snapshot the files and directories, placing them in a FLASH directory
- change these files and directories to symbolic links pointing to the ramdisk locations
- on boot, copy the snapshot from the FLASH directory to ramdisk

As the project matured, the Linux box built the code and downloaded it to the target to run and test. With the monitor and keyboard attached, the target system allowed console access to each process and control of the system. Without the keyboard and monitor, which was the normal target configuration, Telnet provided much of the same debug access into the system and processes. I only ran the code on the development system when I needed to use the debugger.

Using Linux in an embedded system provides excellent tools for development and testing and the flexibility that allowed compatible operation from a full featured environment to a limited target. The end result is a development process that can separate application development from the embedded target issue, often removing operating system configuration and driver development from the critical path and allowing the port to the target be done in smaller and more controlled steps.

**David Rein** is a software engineer and the founder of DBR Consulting, LLC. He has been designing embedded and real-time systems for over thirty years. He can be reached at [drein@dbrco.com](mailto:drein@dbrco.com).

## References

### O'Reilly Books

"Building Embedded Linux Systems" by Karim Yaghmour	Great resource for understanding the configuration and issues related to embedded Linux. Only covers the 2.4 kernel.
"Linux Device Drivers" by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman	Complete details on Linux device driver development for the 2.4 and 2.6 kernels.

### News Groups

Comp.os.linux.embedded	Focused and low flame, the title says it all.
Comp.os.arch.embedded	More general news group but very informative. Lots of hardware discussion.
Comp.os.linux.announce	Announcements of what's new in the Linux world.
Comp.realtime	More software oriented than comp.os.arch.embedded.

*continued on page 4*

continued from page 3

### Web Sites

<a href="http://www.linuxdevices.com">www.linuxdevices.com</a>	Great forums on embedded Linux issues.
<a href="http://www.damnsmalllinux.org">www.damnsmalllinux.org</a>	50MB port of Debian Linux, complete with GUI and broad support of PC devices. Put a copy on a USB FLASH disk and you can recover a damaged system.
<a href="http://www.uclinux.org">www.uclinux.org</a>	uCLinux is a port of Linux for very small systems, including processors without memory management. Works well on low end Coldfire systems.
<a href="http://www.rtlinuxfree.com">www.rtlinuxfree.com</a>	RTLinux has support for hard real-time.
<a href="http://www.sourceforge.net">www.sourceforge.net</a>	Repository of many open source projects.
<a href="http://www.lnxw.com">www.lnxw.com</a>	LynuxWorks is a commercial Linux operating system for real-time embedded projects.
<a href="http://www.mvista.com">www.mvista.com</a>	Monta Vista provides real-time enhanced ports of Linux for many different processors and off the shelf boards.
<a href="http://www.timesys.com">www.timesys.com</a>	TimeSys provides Linux development tools and board support packages.



ADDRESS SERVICE REQUESTED

Presorted Standard  
US Postage Paid  
Permit #187  
Ann Arbor MI

### In This Issue:

- Employment Agreements
- Overview of Embedded Linux, Part II, A Short HOWTO