Embedded Linux applications: An overview

From wrist watches to cluster-based supercomputers

Darrick Addison (dtadd95@bellatlantic.net)
Senior Software Engineer/Consultant
ASC Technologies Inc.

01 August 2001

After a survey of Embedded Linux applications and their environments, Darrick Addison gives you step-by-step instructions for setting up a suitable hardware and software environment for developing those applications.

Linux now spans the spectrum of computing applications, including IBM's tiny Linux wrist watch, hand-held devices (PDAs and cell phones), Internet appliances, thin clients, firewalls, industrial robotics, telephony infrastructure equipment, and even cluster-based supercomputers. Let's take a look at what Linux has to offer as an embedded system, and why it's the most attractive option currently available.

Emergence of embedded systems

The computers used to control equipment, otherwise known as *embedded systems*, have been around for about as long as computers themselves. They were first used back in the late 1960s in communications to control electromechanical telephone switches. As the computer industry has moved toward ever smaller systems over the past decade or so, embedded systems have moved along with it, providing more capabilities for these tiny machines. Increasingly, these embedded systems need to be connected to some sort of network, and thus require a networking stack, which increases the complexity level and requires more memory and interfaces, as well as, you guessed it, the services of an operating system.

Off-the-shelf operating systems for embedded systems began to appear in the late 1970s, and today several dozen viable options are available. Out of these, a few major players have emerged, such as VxWorks, pSOS, Neculeus, and Windows CE.

Advantages/disadvantages of using Linux for your embedded system

Although most Linux systems run on PC platforms, Linux can also be a reliable workhorse for embedded systems. The popular "back-to-basics" approach of Linux, which makes it easier and more flexible to install and administer than UNIX, is an added advantage for UNIX gurus who already appreciate the operating system because it has many of the same commands and programming interfaces as traditional UNIX.

The typical shrink-wrapped Linux system has been packaged to run on a PC, with a hard disk and tons of memory, much of which is not needed on an embedded system. A fully featured Linux kernel requires about 1 MB of memory. However, the Linux micro-kernel actually consumes very little of this memory, only 100 K on a Pentium CPU, including virtual memory and all core operating system functions. With the networking stack and basic utilities, a complete Linux system runs quite nicely in 500 K of memory on an Intel 386 microprocessor, with an 8-bit bus (SX). Because the memory required is often dictated by the applications needed, such as a Web server or SNMP agent, a Linux system can actually be adapted to work with as little as 256 KB ROM and 512 KB RAM. So it's a lightweight operating system to bring to the embedded market.

Another benefit of using an open source operating system like Embedded Linux over a traditional real-time operating system (RTOS), is that the Linux development community tends to support new IP and other protocols faster than RTOS vendors do. For example, more device drivers, such as network interface card (NIC) drivers and parallel and serial port drivers, are available for Linux than for commercial operating systems.

Flash memory

Flash RAM is a special kind of memory that most Palm devices use to store the operating system. It has the advantage of allowing operating system upgrades, and it is also used in digital cellular phones, digital cameras, LAN switches, PC cards, digital set top boxes, embedded controllers, and other small devices. An embedded system, like Embedded Linux, does not require a disk drive, although a number of other memory organizations are possible. So if, let's say, Linux runs out of flash memory, it may use part of the flash memory as a read-only file system to store extra programs and static data.

The core Linux operating system itself has a fairly simple micro-kernel architecture. Networking and file systems are layered on top of the micro-kernel in modular fashion. Drivers and other features can be either compiled in or added to the kernel at run-time as loadable modules. This provides a highly modular building-block approach to constructing a custom embeddable system, which typically uses a combination of custom drivers and application programs to provide the added functionality.

An embedded system also often requires generic capabilities, which, in order to avoid re-inventing the wheel, are built with off-the-shelf programs and drivers, many of which are available for common peripherals and applications. Linux can run on most microprocessors with a wide range of peripherals and has a ready inventory of off-the-shelf applications.

Linux is also well-suited for embedded Internet devices, because of its support of multiprocessor systems, which lends it scalability. This capability gives a designer the option of running a real-time

application on a dual processor system, increasing total processing power. So you can run a Linux system on one processor while running a GUI, for example, simultaneously on another processor.

The one disadvantage to running Linux on an embedded system is that the Linux architecture provides real-time performance through the addition of real-time software modules that run in the kernel space, the portion of the operating system that implements the scheduling policy, hardware-interrupts exceptions and program execution. Since these real-time software modules run in the kernel space, a code error can impact the entire system's reliability by crashing the operating system, which can be a very serious vulnerability for real-time applications.

An off-the-shelf RTOS, on the other hand, is designed from the ground up for real-time performance, and provides reliability through allocating certain processes a higher priority than others when launched by a user as opposed to by system-level processes. Processes are identified by the operating system as programs that execute in memory or on the hard drive. They are assigned a process ID or a numerical identifier so that the operating system may keep track of the programs currently executing and of their associated priority levels. Such an approach ensures a higher reliability (predictability) with the RTOS time than Linux is capable of providing. But all-in-all, it's still a more economical choice.

Different types of Embedded Linux systems
There are already many examples of Embedded Linux systems; it's safe to say that some form of Linux can run on just about any computer that executes code. The ELKS (Embeddable Linux Kernel Subset) project, for example, plans to put Linux onto a Palm Pilot. Here are a couple of the more well-known small footprint Embedded Linux versions:

ETLinux -- a complete Linux distribution designed to run on small industrial computers, especially PC/104 modules.

LEM -- a small (<8 MB) multi-user, networked Linux version that runs on 386s.

LOAF -- "Linux On A Floppy" distribution that runs on 386s.

uClinux -- Linux for systems without MMUs. Currently supports Motorola 68K, MCF5206, and MCF5207 ColdFire microprocessors.

uLinux -- tiny Linux distribution that runs on 386s.

ThinLinux -- a minimized Linux distribution for dedicated camera servers, X-10 controllers, MP3 players, and other such embedded applications.

Software and hardware requirements

Several user-interface tools and programs enhance the versatility of the Linux basic kernel. It's helpful to look at Linux as a continuum in this context, ranging from a stripped-down micro-kernel with memory management, task switching and timer services to a full-blown server supporting a complete range of file system and network services.

A minimal Embedded Linux system needs just three essential elements:

- A boot utility
- The Linux micro-kernel, composed of memory management, process management and timing services
- An initialization process

To doing anything useful while remaining minimal, you also need to add:

- Drivers for hardware
- One or more application processes to provide the needed functionality

As additional requirements become necessary, you might also want:

- A file system (perhaps in ROM or RAM)
- TCP/IP network stack
- A disk for storing semi-transient data and swap capability
- A 32-bit internal CPU (required by all complete Linux systems)

Hardware projects of interest

Here are some current embedded hardware projects tailored to the Linux operating system.

PLEB: Pocket Linux Embedded Box with ARM SA-1100 / ArmLinux Ucsimm / Uclinux Flash EPROM.

Linux Lab: The Linux Lab project is intended to help people develop data-collection and process-control software for Linux. It is planned to provide a standardized development environment for a wide variety of applications, from hardware support to application development.

Controller Area Network: Controller Area Network (CAN) bus driver for Linux GPIB; the Linux GPIB Package is a support package for usual GPIB (IEEE 488.1) hardware. The driver supports National Instruments AT-GPIB, TNT488.2, and PCII and PCIIa boards. This package contains a complete development environment with testing and configuration tools, libraries and tcl and python language support.

Hardware platform options

Choosing the best hardware can be complex because of internal company politics, prejudices, legacies of other projects, a lack of complete or accurate information, and cost, which should take into account the total product costs and not just the CPU itself. Sometimes a fast, inexpensive CPU can become expensive once bus logic and the delays necessary to make it work with other peripherals are considered. To calculate the necessary speed of a CPU for any given project, start with a realistic view of how fast the CPU needs to run to accomplish a given task and triple it. Also, determine how fast the bus needs to run. If there are secondary buses, such as a PCI bus, consider them also. A slow bus (that is, one that is saturated with DMA traffic) can significantly slow down a fast CPU. Here are some of the best hardware solutions for Embedded Linux applications.

Bright Star Engineering: Bright Star Engineering's ipEngine-1 is a credit-card sized single-board computer with Embedded Linux support. It utilizes a PowerPC-based CPU and provides an array of on-board peripherals, including Ethernet, LCD/Video Controller, USB, Serial I/O, and a 16K gate user-configurable FPGA. BSE's Embedded Linux configuration allows Linux to be booted from the ipEngine's on-board 4MB flash memory.

Calibri: CalibriTM-133 is a ready-to-use, compact, multipurpose network appliance that uses Embedded Linux as its operating system. It offers a highly efficient and low-cost solution to firewall, VPN, and routing demands.

EmbeddedPlanet: EmbeddedPlanet has created a PostPC-era computer that comes loaded with MontaVista's HardHat Linux. Powered by a PowerPC-based computing engine and matching I/O card, Linux Planet comes in a colorful translucent case with a touchscreen and access to digital and analog I/O.

Eurotech: Eurotech provides embedded PC SBC and sponsors ET-Linux, a complete Linux system designed to run on small industrial computers and based on glibc 2.1.2.

Microprocess Ingenierie: Microprocess develops, produces, and sells standard and customized products for the industrial and embedded market. Microprocess has a global activity in real-time software and great expertise in systems integration. Its products, like the 740 PowerPC compactPCI board, can be ordered with a standard distribution of Linux or an Embedded Linux version.

Moreton Bay: Moreton Bay is releasing their NETtel 2520 and NETtel 2500 range of Linux-based Internet routers. These small, easy-to-connect intelligent router solutions are engineered to offer a simple, secure, and affordable extranet-friendly Virtual Private Network (VPN) for flat networks. The NETtel router family runs an Embedded Linux kernel. A development kit is available; it enables customized code to be stored in flash memory and executed inside the NETtel. The code may contain special encryption or authentication protocols, or some local monitoring script where NETtel is used as a remote control device.

Matrix Orbital: This an optional, but not recommended, addition. Matrix Orbital manufactures a line of serial LCDs and VFDs, which many Linux users are including in their embedded systems. The product line ranges from 8x2 to 40x4 character LCDs, 20x2 and 20x4 VFDs, plus a 240x64 graphic LC (128x128 on the way). Communication with the displays is accomplished via either RS232 or I2C, both of which are standard on all of their modules. A comprehensive command set is included in the modules' BIOS.

Real-time Embedded Linux applications

One of the most important issues with embedded systems is the need for a real-time operating system. The definition of real-time here varies quite a bit. To some people, real-time means responding to an event in the one-microsecond range, to others it is 50 milliseconds. The hardness of real-time also varies quite a bit. Some systems need hard real-time response, with short

deterministic response latencies to events. However, on many systems, when analyzed closely, we see a response time requirement that is actually near real-time. Often the real-time requirement is a tradeoff of time and buffer space. With memory getting cheaper, and CPUs getting faster, near real-time is now more typical than hard real-time and many commercial operating systems that claim to be real-time are far from being hard real-time. Usually, when you get into the detailed design of these systems, there are warnings that the drivers' interrupts and applications must be very carefully designed in order to meet real-time requirements.

RT-Linux (Linux with real-time extensions) contains time critical functions to provide precise control over interrupt handling, through the use of an interrupt manager, and does a good job of making sure that critical interrupts get executed when needed. The hardness of this approach depends mostly on the CPU interrupt structure and context-switch hardware support. This approach is sufficient for a large range of real-time requirements. Even without the real-time extensions, Linux does pretty well at keeping up with multiple streams of events. For example, a Linux PC system on a low end Pentium is able to keep multiple 10BaseT interfaces executing effectively, while simultaneously running character-level serial ports at a full 56KBPS without losing any data.

Some real-time hardware and software Linux APIs to consider are RTLinux, RTAI, EL, and Linux-SRT. RTLinux is a hard real-time Linux API originally developed at the New Mexico Institute of Technology. RTAI (DIAPM) is a spin-off of the RTLinux real-time API that was developed by programmers at the Department of Aerospace Engineering, Polytechnic Politecnico di Milano (DIAPM). EL/IX is a proposed POSIX-based hard real-time Linux API being promoted by Red Hat. And Linux-SRT is a soft real-time alternative to real-time APIs, which provides performance-enhancing capabilities to any Linux program without requiring that the program be modified or recompiled.

See the Resources section later in this article for information on the above and for some Web sites offering different flavors of software extensions, development tools, support, and training courses for the standard Linux operating system.

Short deterministic response latencies
Some real-time embedded systems need to respond quickly to external events in order to accomplish a specific task. A custom microcontroller embedded inside a missile, for example, needs to respond quickly to external events such as moving targets, weather, humans, etc., before instructing the missile to target a specific object in its surrounding environment. Short deterministic response latencies mean that the embedded system can determine the time it will take to respond to an external event.

Configuration procedures

Now let's take a look at how to make LEM, a small, embeddable Linux distribution, which provides both network and X server. You can download this distribution, although it is not essential. You will need a full Linux distribution to build your own Embedded Linux operating system, which will contain everything you need (utilities, sources, compiler, debugger, and documentation). Here is a list of the software that can be used to make LEM:

TinyLogin: TinyLogin is a suite of tiny UNIX utilities for handling logging into, being authenticated by, changing one's password for, and otherwise maintaining users and groups on an embedded system. It also provides shadow password support to enhance system security. TinyLogin is, as the name implies, very small, and makes an excellent complement to BusyBox on an embedded System.

BusyBox: BusyBox is a multicall binary used to provide a minimal subset of POSIX-style commands and specialized functions. It is geared toward the very small, such as boot floppies, embedded systems, etc. Specifically it is used in the Debian Rescue/Install system (which inspired development on the original BusyBox), the Linux Routeur Project, LEM, lineo, and others. Busybox is being maintained by Erik Andersen.

Ash: Ash is a very small Bourne shell.

Sysvinit: Sysvinit is the most used init package for Linux. We will use init and the C version of the start-stop-daemon.

See the Resources section for more information on these items.

Creating a bootdisk

A bootdisk is basically a miniature, self-contained Linux system on a floppy diskette. It can perform many of the same functions that a complete full-size Linux system performs. The following material is based on the Bootdisk-HOWTO (see Resources).

Step 1. Bios

All PC systems start the boot process by executing code in ROM (specifically, the BIOS) to load the sector from sector 0, cylinder 0 of the boot drive. The boot drive is usually the first floppy drive (designated A: in DOS and /dev/fd0 in Linux). The BIOS then tries to execute this sector. On most bootable disks, sector 0, cylinder 0 contains either:

- Code from a boot loader such as LILO, which locates the kernel, loads it, and executes it to start the boot proper
- The start of an operating system kernel, such as Linux

If a Linux kernel has been raw copied to a diskette, a hard drive, or another media, the first sector of the disk will be the first sector of the Linux kernel itself. This first sector will continue the boot process by loading the rest of the kernel from the boot device.

Step 2. The boot loader

You will use a boot loader like LILO to operate the boot process. It allows the development and production platforms to co-exist on the same hardware and permits switching from one to the other just by rebooting. The LILO boot loader is loaded by the bios. It then loads kernels or the boot sectors of other operating systems. It also provides a simple command line interface to interactively select the item to boot with its options. See Resources for more information on LILO.

Step 3. The kernel
The kernel checks the hardware and mounts the root device and then looks for the init program on the root filesystem and executes it.

Step 4. Init
Init is the parent of all other processes that will run on your Linux OS. It will watch its child processes and start, stop, re-launch them if needed. Init takes all information from /etc/inittab.

Step 5. Inittab
The file /etc/inittab/ refers to scripts named /etc/rc... to do the system setup. It also has entries for the getty tool to handle the login process.

Step 6. The login process
There is one getty available in the inittab file for each console allowed for the users. Getty will launch /bin/login to verify the user password.

Step 7. Creating a new partition
From the LFS-HOWTO (see Resources): Before we can build our new Linux system, we need to have an empty Linux partition on which we can build our new system. If you already have a Linux Native partition available, you can skip this step and the following one. Start the fdisk program (or cfdisk if you prefer that program) with the appropriate hard disk as the option (like /dev/hda if you want to create a new partition on the primary master IDE disk). Create a Linux Native partition, write the partition table, and exit the (c)fdisk program. If you get the message that you need to reboot your system to ensure that the partition table is updated, then please reboot your system now before continuing.

Step 8. Creating an ext2 file system on the new partition
From the LFS-HOWTO (see Resources): To create a new ext2 file system we use the mke2fs command. Give $LFS as the only option, and the file system will be created. From now on I'll refer to this newly created partition as $EMBPART. $EMBPART should be substituted with the partition you have created.

Step 9. Mounting the partition
To access the newly created filesystem, you have to mount it. To do this, create an /mnt/hda? directory and type the following at the shell prompt:

```
mkdir /mnt/hda?
mount $EMBPART /mnt/hda?
```

If you created your partition on /dev/hda4 and you mounted it on /mnt/hda4, then you'll need to return to the step where you copied a file to $dollar;EMBPART/usr/sbin, and copy that file to /mnt/hda4/usr/bin. Do this after the last command in Step 14 (Copy the file in $EMBPART/usr/sbin).

Step 10. Populating the filesystem
The root filesystem must contain everything needed to support a full Linux system. We will build a directory structure not that far from the File Hierarchy Standard (see Resources).

Step 11. Directories

The mkdir function in the new mounted filesystem creates the following directories:

/proc
Directory stub required by the proc filesystem
/etc
System configuration file
/sbin
Critical System binaries
/bin
Basic binaries considered part of the system
/lib
Shared Libraries to provide run-time support
/mnt
Mount point for maintenance
/usr
Additional utilities and applications

- cd /mnt/hda?
- mkdir bin dev home proc sbin usr boot etc liv mnt root tmp var
- mkdir -p usr/bin usr/sbin usr/share usr/lib
- mkdir -p etc/config etc/default etc/init.d etc/rc.boot
- mkdir -p etc/rc0.d etc/rc1.d etc/rc2.d etc/rc3.d etc/rc4.d etc/rc5.d etc/rc6.d etc/rcS.d

/dev
The dev directory is the stub required to perform devices input / output. Each file in this directory may be created using the mknod function. You may save time by directly copying the required dev entries from your desktop Linux, using the following instruction:
cp -dpR /dev /mnt

Installing TinyLogin and login dependencies

TinyLogin (see the Resources section to install it) will give us the following tools in less than 35Kb: /bin/addgroup, /bin/adduser, /bin/delgroup, /bin/deluser, /bin/login, /bin/su, /sbin/getty, /sbin/sulogin, /usr/bin/passwd.

Please refer to your main distribution doc or man pages for a full description of those commands.

Step 12. Configuring TinyLogin

From the TinyLogin README: TinyLogin is modularized to help you build only the components you need, thereby reducing binary size. To turn off unwanted TinyLogin components, simply edit the file tinylogin.def.h and comment out the parts you do not want using C++ style (//) comments.

Step 13. Installing TinyLogin

After the build is complete, a tinylogin.links file is generated, which is then used by make install to create symlinks to the tinylogin binary for all compiled-in functions. By default, make install

will place a symlink forest into *pwd* /_install unless you have defined the PREFIX environment variable.

Step 14. Installing Sysvinit and start-stop daemon

After the kernel is done loading, it tries to run the init program to finalize the boot process. Now:

1. Unpack the Sysvinit archive
2. Go to the src directory
3. Copy the init executable in $EMBPART/sbin

The Sysvinit package also offers a C version of the start-stop-daemon in the contrib directory.

1. Compile it
2. Copy the file in $EMBPART/usr/sbin

Step 15. Configuring Sysvinit

Sysvinit needs a configuration file named inittab, which should be placed in $EMBPART/etc. Here is the one used in the LEM distribution:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.6 1997/01/30 15:03:55 miquels Exp $
# Modified for LEM 2/99 by Sebastien HUET
# default rl.
id:2:initdefault:
# first except in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS
# single-user mode.
~~:S:wait:/sbin/sulogin
# /etc/init.d executes the S and K scripts upon change
# 0:halt 1:single-user 2-5:multi-user (5 may be X with xdm or other) 6:reboot.
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
# CTRL-ALT-DEL pressed.
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -r now
# Action on special keypress (ALT-UpArrow).
kb::kbrequest:/bin/echo "Keyboard Request--edit /etc/inittab to let this work."
# /sbin/mingetty invocations for runlevels.
1:2345:respawn:/sbin/getty 9600 tty1
2:23:respawn:/sbin/getty 9600 tty2
#3:23:respawn:/sbin/getty tty3 #you may add console there
#4:23:respawn:/sbin/getty tty4
```

Step 16. Creating initial boot scripts

As seen in the inittab file, Sysvinit needs additional scripts in its own directories.

Step 17. Creating the necessary directories and base files

Use the following command to create the directories:

```
cd $EMBPART/etc
mkdir rc0.d rc1.d rc2.d rc3.d rc4.d rc5.d rc6.d init.d rcS.d rc.boot
```

Go to the unpacked Sysvinit source directory
Copy the debian/etc/init.d/rc to:$EMBART/etc/init.d
Go to the $EMBPART/etc/init.d/
Create a new file rcS like those in LEM:

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel
/etc/default/rcS
export VERBOSE
# Trap CTRL-C  only in this shell so we can interrupt subprocesses.
trap ":" 2 3 20
# Call all parts in order.
for i in /etc/rcS.d/S??*
do
        [ ! -f "$i" ] && continue
        case "$i" in
                *.sh)
                        (
                                trap - 2 3 20
                                . $i start
                        )
                        ;;
                *)
                        $i start
                        ;;
        esac
done
# run the files in /etc/rc.boot
[ -d /etc/rc.boot ] && run-parts /etc/rc.boot
```

Copy run-parts from your distro to $EMBPART/bin.

Step 18. Adding base scripts

A lot of the commands being used here are UNIX/Linux commands that set, export, etc. paths that are embedded inside of a UNIX shell script.

```
<!-reboot---------------------------------------------->
```

Create a new file reboot containing the following:

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
echo -n "Rebooting... "
reboot -d -f -i
<!-halt------------------------------------------------>
```

Create a new file halt containing the following:

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
halt -d -f -i -p
<!-mountfs--------------------------------------------->
```

Summary

The Linux operating system has a very bright future in the area of embedded applications for anything from Internet appliances to dedicated control systems. Roughly 95% of all newly manufactured microcomputer chips are used for embedded applications. The power, reliability, flexibility, and scalability of Linux, combined with its support for a multitude of microprocessor architectures, hardware devices, graphics support, and communications protocols have established Linux as an increasingly popular software platform for a vast array of projects and products.

Because Linux is openly and freely available in source form, many variations and configurations of Linux and its supporting software components have evolved to meet the diverse needs of the markets and applications to which Linux is being adapted. There are small footprint versions and real-time enhanced versions. Despite the origins of Linux as a PC architecture operating system, there are now ports to numerous non-x86 CPUs, with and without memory management units, including PowerPC, ARM, MIPS, 68K, and even microcontrollers. But look out, there's more coming in the near future for many other Information Technology (IT) domains!

Resources

- Get a complete Linux distribution of ETLinux.
- Take a look at the multi-user, networked Linux version LEM.
- ThinLinux is a minimized Linux distribution for dedicated camera servers, X-10 controllers, MP3 players, and other such embedded applications.
- RTLinux is a hard real-time Linux API originally developed at the New Mexico Institute of Technology.
- EL/IX is a POSIX-based hard real-time Linux API.
- TinyLogin is being maintained by Erik Andersen.
- Ash is a very small Bourne shell.
- Sysvinit is the most used init package for Linux. We will use init and the C version of the start-stop-daemon.
- Read the LILO documentation.
- Go to the LFS-HOWTO to create a new partition, and an ext2file system on the new partition.
- Real-Time Linux is an extension of the standard Linux operating system. It provides a simple and streamlined real-time executive that runs the standard Linux kernel as its lowest-priority task, while allowing the insertion of user-defined, higher-priority (real-time) tasks, while still allowing full access to the sophisticated services and features of standard Linux.
- Linux-SRT is an extension to the Linux kernel, which improves the way real-time (RT) applications are run. Standard Linux isn't designed to be a "Media OS" and cannot guarantee that audio, visual, or other critical processes run at a fixed rate.
- Visit IBM's home for embedded Linux.
- Check out IBM's embedded Linux products and developer support.
- Browse more Linux resources on *developerWorks*.
- Browse more Open source resources on *developerWorks*.

About the author

Darrick Addison

Darrick Addison works as a Senior Software Engineer/Consultant for his company, ASC Technologies Inc. He has been designing and developing custom software applications since 1993. He has worked on the design and development of software ranging from database applications, network applications (TCP/IP client/server), GUI applications, and embedded systems applications in various commercial and government environments. He currently holds a BS degree in Computer Science and is pursuing his Master's degree in Computer Science/Telecommunications at Johns Hopkins University. You can contact Darrick at dtadd95@bellatlantic.net. He welcomes your comments and questions.