An Examination of Embedded Linux as a Real Time Operating System

Mark Mahoney
CS550 Fall 2001
November 29, 2001

Abstract: Embedded Linux is a non-real time operating system that can be made to work in the embedded environment. The operating system is rich in features, modular, open source, and free. Using a simple method of interrupt abstraction, embedded Linux can handle hard real time requirements and provide functionality that most real time operating systems cannot.

The Linux operating system is a robust, sophisticated, highly reputable operating system for desktop PC's and servers. The success of Linux in the desktop and server environments is well accepted. The sheer number of PC's and servers running Linux attest to the fact that users are indeed satisfied with Linux as a desktop/server operating system. Since Linux is so well accepted, and free with open source code, it is no wonder that embedded developers have begun to look for ways to incorporate this constantly evolving product into their devices.

One of the great benefits of using any variant of Linux is that there are many developers committed to adding functionality and fixing bugs to the open source tree for this operating system. The history of Linux goes back to 1991 when Linus Torvalds began to work on an operating system for an x86 compatible machine, inspired by Andrew Tannenbaum's Minix operating system. Torvalds eventually posted the work he had done to the internet and asked others to add, modify and enhance his work. Some of the brightest software developers took him up and a revolution began.

Real Time Embedded

Because the benefits of open source development are so numerous, the embedded world is trying to ride the coattails of the operating system that is sweeping the industry. Before an examination of how Linux is made to work in the embedded world we must first distinguish how embedded devices differ from desktop and server machines that Linux was designed to work on.

An embedded device is one in which some form of computing mechanism is embedded into a device that is not a computer, that is, a device that has functions other than raw computing. For example, the purpose of a cellular telephone isn't to compute the sum of rows in a spreadsheet or store files, its purpose is to allow mobile communication. A cellular phone, however, could not provide this functionality if it were not for the embedded processor that controls the transceiver, the user interface and a host of other subsystems.

A microwave oven might be a better example because you don't expect much, if any, processing of data to produce information from a microwave oven. What you expect is hot food. Nevertheless, a microwave oven is highly dependent on an embedded processor controlling the functionality of the many different parts that make up a microwave oven.

Furthermore, a *real time* embedded device is one that can respond to events in a consistent, time bounded manner. The handling of these events makes up the real time requirements. Because embedded processors control embedded devices, and processors are controlled by software, it is the responsibility of software engineers to guarantee that real time requirements are met. The software engineer's greatest tool to accomplish this is the real time operating system.

A real time operating system allows individual tasks of the system to be broken up and run concurrently, with the caveat that the system will always respond to certain events within a specified time limit. Imagine the case where an airline pilot wishes to change its course to avoid hitting a mountain. In this case, it is of the utmost importance that the system responds in a timely manner. The pilot cannot afford to have a variable response time in this situation. A failure of the system, and the loss of many lives, would result if the system didn't respond in a way that was expected from the pilot. There are other tasks that the system would still have to complete, but when a change in direction is required it takes precedence and must be addressed within a fixed time limit.

Having said that, it is important to state that Linux is not a real time operating system, and thus it would appear that it would be a poor choice to handle the real time requirements of an embedded device. Not every embedded device, however, has the burden of protecting human life. There are many embedded devices where missing a deadline set by some real time requirement would be a nuisance, but would not kill anyone. This leads to the distinction between real time requirements.

A hard real time requirement is one in which the success of meeting the requirement is not only dependent on achieving the correct result, but also getting the result within specified time limit. A failure of the system would result if either factor were not met, for example starting to turn an airplane after it hit the mountain. This is not to say that if you fail to meet every hard real time requirement in every system that people will be harmed, but it does mean that the system will fail. There is no recovery from missing a hard real time deadline. The implication is that devices would shut down, or restart upon missing a real time deadline, losing the associated state information of the session.

A soft real time requirement also requires time-bounded results, but if they are occasionally missed the system can recover. It would be nice if we received each and every packet in a streaming audio application, but if we occasionally miss a packet the application could continue with reasonably good quality of audio. It should be said that if enough soft real time requirements were missed with some frequency then it would be considered a failure of the application. For example, if we only received one out of every three streaming audio packets then there would be audio, but the quality would be so bad that it would no longer qualify as meaningful audio. Therefore, there is a limit to how many soft real time requirements can be missed and still maintain an acceptable level of functionality.

The difference between hard and soft real time requirements, then, is that missing a hard real time deadline is an unrecoverable catastrophic error in the application, whereas missing an occasional soft real time requirement may not halt the functionality of the system.

The Current State of the Embedded World

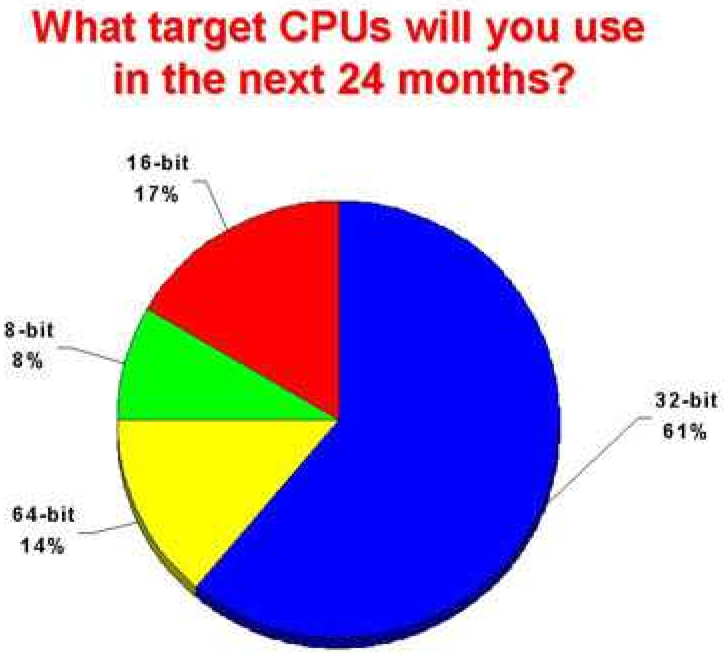
Embedded real time devices proliferate in the market. The devices sold today are smarter, faster and cheaper than entire computer systems of just a few years ago. PDA's, cell phones and internet appliances are being sold in increasing quantities. There is a greater need for high-resolution graphic displays, TCP/IP stacks, and wireless networking devices.

The desire for these products is increasing the demand for state of the art operating systems that can support features that were only available on dedicated desktop machines in the past. Protocol popularity is playing a factor in the development modern operating systems. Customers are now expecting protocol support in their embedded devices. Product designers want support for everything from TCP/IP to Bluetooth to 802.11 built in to the operating system and ready to go. This greatly adds to the complexity of modern operating systems.

Hardware is also evolving quickly, and prices are becoming so low that real time operating system vendors are having trouble keeping pace. Device driver development and operating system upgrades are making it harder to keep up with technology- only the best supported real time operating systems will survive.

It is even harder for companies that have created their own proprietary operating systems for their products. Without the resources of a large operating system development company it is almost impossible for these "home grown" solutions to be developed and tested before becoming obsolete by the next generation protocol or hardware. The goal of all real time operating system developers is to provide advanced features while maintaining the reliability that is associated with the real time operating system of the past.

The good news is that faster processors and memory are making it easier for applications to satisfy real time requirements. Remember, embedded devices of a few years ago were running 8-bit processors, running in single digit megahertz and with memory in the kilobytes. Below are the results from an online survey of embedded developers sponsored by linuxdevices.com (2001)



(Copyright ©2001, CNET Networks, Inc.)

It is apparent that developers will be taking advantage of the latest developments in processors. Time to market and adding features are now much more important then when there were significant hardware limitations. With severe hardware limitations, just getting a semi-functional product out to market was a major accomplishment.

In many cases, when hard time requirements are measured in milliseconds, traditional operating systems can handle them reducing the need for specialized real time operating systems. Writing applications without an operating system is too simplistic for today's requirements. DOS is out of touch, "home grown" kernels require an incredibly steep technology curve, Windows is perceived as unreliable, fat and expensive and UNIX is not very embeddable. There are vendors in the market that are keeping up, but they of course, come with a price tag. Some popular commercial real time operating systems are VxWorks, pSOS and WinCE.

Embedding Linux

Linux is not a real time operating system, it lacks event prioritization and preemption functionality in the kernel. Except for this fact, Linux is an ideal operating system for embedded devices. Linux is powerful and offers sophisticated features, it's free, it comes with source code, it is modular (unwanted features can be removed), it comes with support for many devices and protocol stacks, it is well documented and is constantly improved by a loyal group of developers. A great deal of work has been done to overcome the limitations of Linux in the embedded environment, and several solutions have been proposed.

One of the keys to embedding Linux is reducing the size requirements for the Linux kernel. The full kernel is currently about 40 million lines of source code. The footprint for the full kernel would be too large to fit on most embedded devices. Fortunately, Linux was developed as a modular piece of software. This not only helped reduce the footprint but also made it easy for separate developers to work on it independently. Linux's modularity allows nonessential portions of the kernel to be left out, thus reducing the ROM requirements for embedded devices.

The kernel can be made to fit on a floppy, but this version would include almost none of the rich features that make Linux so desirable. A typical installation with a fully functional kernel and a very impressive set of basic features (networking, shell, built in drivers, etc) can fit on the order of 2-4 MB of ROM with 4-8MB of RAM.

Most Linux variants are similar, the only difference is what is left out for which particular implementation and how the installation process is managed. There are many ready-to-go small footprint versions available, or you can build your own to optimize for your specific application.

In the survey mentioned previously, embedded developers were asked what type of Linux they would need to run their current applications, 44% of the embedded developers polled said that the only thing they needed to run Linux on their embedded devices is a reduced footprint version, another 12% said they could use a standard version of Linux. 42% of developers said they would need Linux with real time extensions. This means that versions of Linux available right now could handle more than half of all products' real time requirements, either due to advances in hardware or the nature of the applications.

For some devices, though, there still is a need to satisfy some real time requirements. The first approach to allow Linux to handle real time requirements is called the preemption improvement approach. In the preemption improvement approach the Linux kernel is modified to reduce the amount of time that the kernel spends in non-preemptible sections of code. The strategy is to reduce the length of the longest sections of non-preemptible code in order to minimize the latency of interrupts or real time task scheduling in the system. The longest section of non-preemptible code is the shortest scheduling latency that can be guaranteed for a hard real time system running Linux.

The primary way to accomplish this is to add additional points in the Linux kernel where the currently executing kernel thread relinquishes control or makes itself available for preemption. This is done, of course, by altering the code throughout the kernel. The kernel is examined and wherever a long segment of code that disables interrupts or prevents other processes for gaining access to the processor is found, it is rewritten. Each long section of code is broken up into several subsections where the state of the kernel is maintained at the end of each subsection and preemption is allowed to take place if it is needed. This technique is referred to as a low latency patch.

There are several disadvantages to this approach, the most significant being that no guarantee can be given about the longest delay in the kernel. The size and complexity of the kernel mandates that an exhaustive examination of every possible path through the code is impossible. Linux is a constantly evolving piece of software, it would be unrealistic to expect every new feature, every new driver and every bug fix to Linux to meet some "longest allowable preemption limit", and even if it were possible there's no guarantee that everyone would agree on the limit. Linux is truly a global standard, when developers change Linux in a way that breaks the standard they are not doing themselves or others any good. Currently, versions of Linux that implement low latency patches (the most famous of which was written by Ingo Molnar, a key contributor to the Linux kernel) give historical measurements and observations about what the longest measured delay was, no hard limit can be guaranteed.

Additional disadvantages include the possible bugs that all this tweaking throughout the kernel will produce, it has the potential to disrupt a well functioning kernel. Also, the code is harder to follow because the additions break up the logical flow of the kernel code. A less obvious drawback is that with the additional preemption points throughput could be reduced by the additional context switches, register spills and cache misses.

For the reasons stated above, it is clear that the preemption improvement mechanism cannot truly satisfy the portion of developers who require real time extensions. Another technique has been developed that is better suited to the remaining population of developers who require hard real time extensions for their applications to function. The interrupt abstraction method handles the transition of Linux to a real time operating system better and more elegantly than the preemption improvement model.

The interrupt abstraction, or dual kernel, approach's basic implementation is to have an additional real time scheduler that runs the Linux kernel as the lowest priority task. All interrupts are initially handled by the real time scheduler in a manner that satisfies the hard real time requirements of the system. Since the Linux task has the lowest priority in the real time scheduler, it runs only when interrupts are not being handled.

An interrupt abstraction layer sits between the real time scheduler and the Linux task that emulates interrupt control hardware. Interrupts are passed to the Linux kernel through this abstraction layer. This guarantees that the Linux kernel will not be the cause of hard real time requirements being missed. When the Linux kernel is in a non-interruptible section of code and a hardware interrupt is generated, the real time scheduler that sits in front of the Linux scheduler will immediately preempt the lower priority Linux kernel to handle the interrupt. The interrupt will be handled, and a message, or virtual interrupt, will be buffered in the interrupt abstraction layer for the Linux kernel to handle at an appropriate time for the non real time operating system. This way time bounded requirements, such as real world event processing can be handled.

The benefit of this approach is that it allows use of the functionality provided in a sophisticated, ever evolving operating system like Linux while adding real time functionality that is lacking in Linux. The Linux kernel task could handle read/write operations, communications/networking, parallel and serial I/O, system initialization, memory management, file systems, process control, device drivers and a host of other features that most real time operating systems don't do as well as Linux, or at all.

Most real time operating systems don't have the benefit of independent developers constantly adding and improving their operating system. The interrupt abstraction version of Linux retains all the benefits of open source software, while gaining the real time benefits that it was previously lacking.

Using this approach, all non-real time functionality is migrated to the Linux portion of the operating system. For example, data acquisition applications using this approach are usually composed of simple polling or interrupt driven real time tasks that pipe data through a queue to a Linux process that takes care of logging and display. In such cases, the I/O buffering and aggregation performed by Linux provides a high level of average case performance while the real time task meets strict worst case limited deadlines.

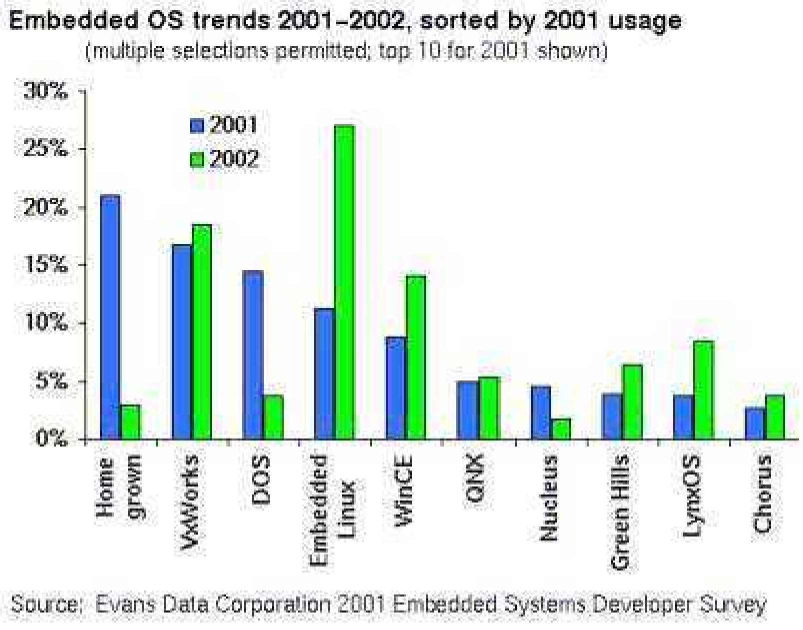
The inventor of this approach, Victor Yodaiken, noted that, “One area where we hope to be able to make particular use of this approach is in Quality of Service (QoS), where it seems reasonable to factor applications into hard real time components that collect or distribute time sensitive data, and Linux processes that monitor data rates, negotiate for process times and adjust algorithms.”

Now that we’ve seen how to embed Linux in devices, lets examine why we would want to do such a thing.

Embedded Linux in Industry

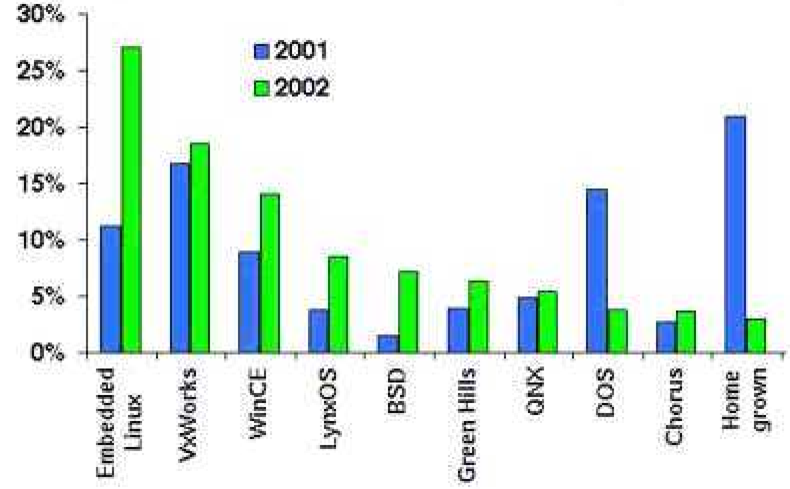
Creating a real time kernel is a major development effort, an even more daunting task is to keep this kernel up to date and bug free with ever evolving hardware improvements. It will become apparent shortly that this method of “rolling you own kernel” is quickly dying. Commercial real time operating system companies are doing their best to keep up with the rapid increases in hardware, and they are doing a reasonably good job at it, what they lack, however, is the enlightened spirit and ingenuity that comes from open source software development. Additionally, these companies profit from the sale and licensing of their product, so developers must spend part of their budget on the fees associated with using a third party operating system, either in up front costs or per unit prices- and sometimes both.

Developers are quickly learning that they can keep their costs down and begin development sooner if they use an open source, off the shelf version of Linux in their devices. Below is a chart of the most used operating systems in embedded devices ordered for the year 2001 from a survey conducted by Evans Data Corporation


Embedded OS trends 2001-2002, sorted by 2001 usage
(multiple selections permitted; top 10 for 2001 shown)

Source: Evans Data Corporation 2001 Embedded Systems Developer Survey

Here you can see that next to home grown kernels and the commercial operating system VxWorks, embedded Linux will be the most used operating system for embedded applications this year. Here is the same chart ordered for what is expected next year

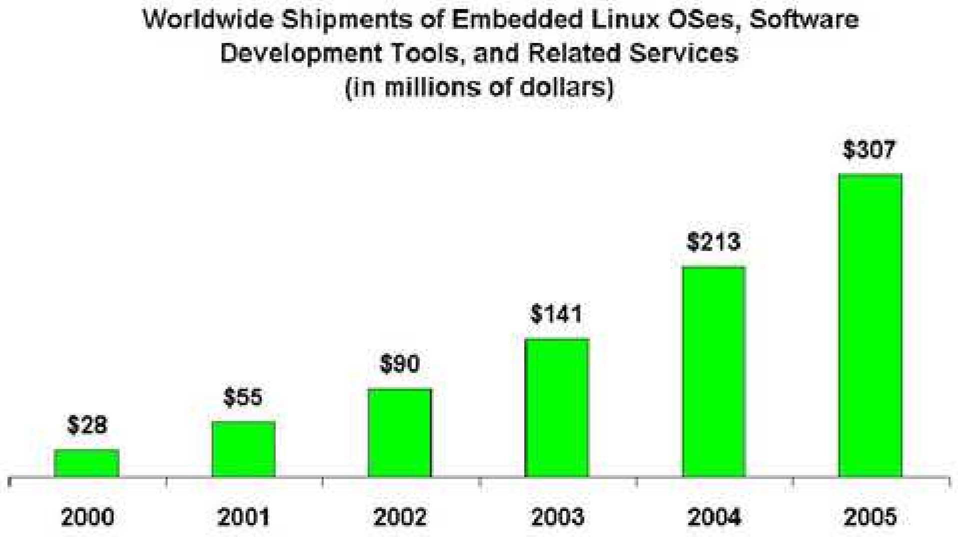


Embedded OS trends 2001-2002, sorted by 2002 expectation
(multiple selections permitted; top 10 for 2002 shown)

Source: Evans Data Corporation 2001 Embedded Systems Developer Survey

The use of homegrown kernels will almost disappear next year, decreasing 633% in a single year. Embedded Linux will pick up the slack to become the most used embedded operating system in 2002 increasing its share by 140%. VxWorks will see slight gains in 2002, but WindRiver, the company that distributes VxWorks, has to be worried. It is clear that rolling your own kernel is too costly and error prone, maybe more importantly it slows down time to market. Developers are clearly choosing embedded Linux as the alternative. Other operating system vendors should be worried as well because once 27% of the total market begins using embedded Linux successfully, like they will next year, there is going to be little incentive to pay the licensing fees associated with particular operating systems. Furthermore, it is going to become less expensive for developers who have been using VxWorks to rewrite their applications using a free version of Linux rather than a licensed version of VxWorks. Because of this fact, the usage of VxWorks and other commercial operating systems should decrease in the years to come.

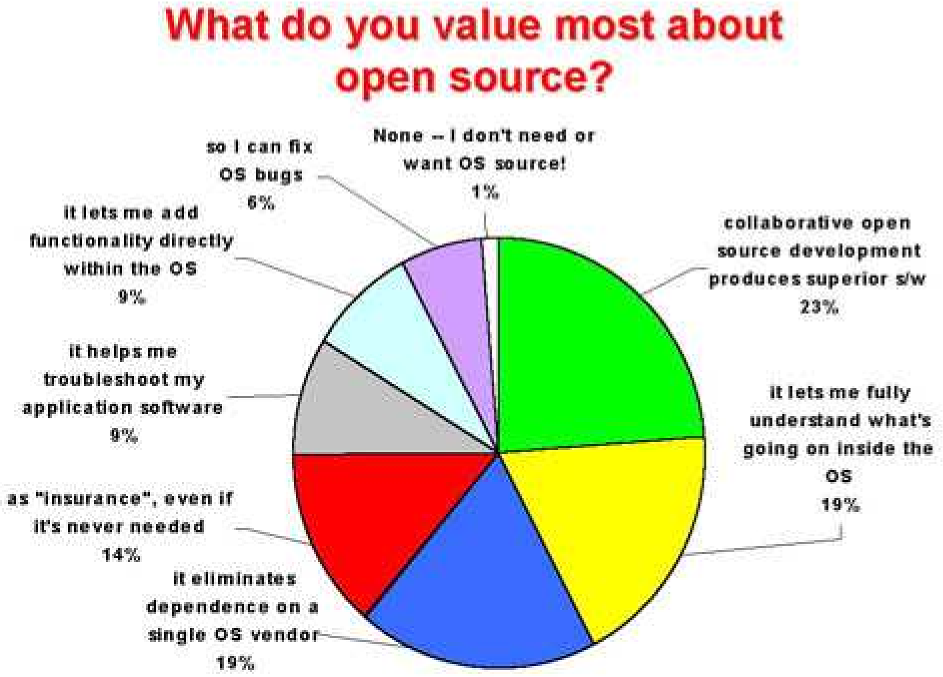
It would be untrue to say that using embedded Linux is a solution that will cost developers nothing. Most developers want some support for their version of Linux other than searching the internet. They also would like help configuring their versions of Linux. There are several Linux vendors who provide just this kind of support. They provide developers with a “distribution” and the support they need to tweak it to meet developers’ needs. In addition, if any issues come up they hand them to the vendor to research and resolve. In a survey of embedded developers 68% said they would pay for Linux development support and services, 19% were undecided and 13% said they would not pay for support. In the same survey, 58% of developers said that they would be unwilling to pay for per unit royalties, 21% were undecided and 21% would pay for per unit royalties. Below is a chart showing that developers will spend increasing amounts of money on embedded Linux distributions and support over the next five years



Source: Venture Development Corporation (VDC) 2000 Embedded Linux Market Study

(Copyright ©2001, CNET Networks, Inc.)

Next the results from the same poll asking what developers like most about open source software



The number one reason why developers value open source software is that they believe that it is superior software, not, as some might think, so that they can fix bugs in the operating system. Almost one quarter of all developers believe open source software is superior to commercial implementations. This is a significant fact primarily because it means that commercial operating vendors can't regain the market share simply by providing source code. It appears that if embedded Linux really takes off that these vendors might be in big trouble.

Conclusion:

Embedding Linux into devices adds significant advantages to embedded development. Linux is a full-featured, extremely well tested operating system with functionality that even the best embedded real time operating systems lack. Through some clever solutions this non-real time operating system is able to handle the strictest of real time requirements. Because it is free and feature rich, embedded Linux is an excellent choice for operating system running on the embedded devices.

Bibliography:

Victor Yodaiken, “The RTLinux Approach to Real Time (A short position paper)”, 1997
http://www.rtlinux.com/whitepaper.html

Kevin Dankwardt, “A Simple Model of Real Time Applications”, 2000
http://www.linuxdevices.com/articles/AT5709748392.html

Jeff Dionne, “When Hard Real Time Goes Soft”, 2000
http://www.linuxdevices.com/articles/AT3524337625.html

Kevin Dankwardt, “Comparing Real Time Linux Alternatives”, 2000
http://www.linuxdevices.com/articles/AT4503827066.html

Rick Lehrbaum, “Using Linux in Embedded and Real Time Systems”, 2000
http://www.linuxdevices.com/articles/AT3611822672.html

Evans Data Corporation, “Embedded Linux Tops Developers’ 2002 Wishlist”, 2001
http://www.linuxdevices.com/articles/NS5134111490.html

Stephen Balacco, “Linux’s Future in the Embedded Systems Market”, 2001
http://www.linuxdevices.com/articles/AT4705998392.html

Bill Peisel, “Whitepaper: Embedding Linux”, 2001
http://www.linuxdevices.com/articles/AT9306437540.html

Tim Bird, “Comparing Two Approaches to Real Time Linux”, 2000
http://www.linuxdevices.com/articles/AT7005360270.html

Rick Lehrbaum, “My Linux is Smaller Than Your Linux”, 2000
http://www.linuxdevices.com/articles/AT8482313700.html