Overview of Linux for the Embedded Application Developer

By Gregory Haerr, CEO, Century Software, Inc.

Abstract
The Linux operating system provides great opportunities in the development of applications and technologies for the growing embedded computing market. This paper will give some of the reasons for the success of Linux in the past and it s continued viability in the future for this emerging market. In addition, a technical overview of Linux is presented, followed by some discussion of graphical windowing system technologies, and an embedded web browser project suitable for mobile applications.

Introduction

The Linux operating system was created by Linus Torvalds at the University of Helsinki in 1991. From its simple beginnings, Linux has become one of the fastest growing platforms, with thousands of developers worldwide. One of the main reasons for the success of the Linux platform has been its development as an open source project, using the GNU General Public License (GPL). This license allows all code developed for the Linux kernel to be used freely by others, for personal or commercial use, and specifically disallows distribution of the system without also having accompanying source code, including all kernel modifications. This tag-along source code feature has attracted contributions from thousands of programmers worldwide for new technologies relating to present-day computing needs. By allowing anyone to study existing implementations and accepting enhancements as contributions, Linus and others have built a system that otherwise would have required very large commercial resources in a short period of time.

Open source software development brings high reliability and performance to the Linux operating system. The reliability of Linux stems from a large set of thousands of programmers obvserving the code, improving it, changing it, and testing it on thousands of different systems configurations. Linus s developmental philosophy of release early, and release often has allowed Linux to grow rapidly as an alternative to other operating systems yet still achieve high reliability and performance.

The initial large development community surrounding the Linux kernel has now grown into an even larger community supporting a large, varied set of applications and technologies for programs running on Linux. One of the more recent advancements has been the rapid movement of tailoring Linux for suitability in the embedded systems market. This started with kernel and compiler support for all the popular 32-bit microprocessors being designed into embedded systems today, including Intel x86, ARM, Motorola/IBM PowerPC, NEC MIPS and Hitachi SH. Several fast-growing commercial embedded Linux software distributions have popped up, with support for features required in embedded systems designs.

Because of Linux s free open source availability, lack of royalties, and support for modern processor architectures and tecnologies, it is making big inroads into the traditional real time operating system (RTOS) market. Linux is being designed in to an increasing number of products for these reasons.

The Linux distribution includes kernel support for all of the technologies required for modern 32-bit processors, increasingly found in embedded systems designs. This includes support for memory management, process and thread creation, interprocess communications mechanisms, interrupt handling, execute-in-place ROM filesystems, RAM filesystems, flash management, and TCP/IP networking. Various developer groups and commercial companies concentrate on specific technologies, with GPL modifications being returned to the standard Linux distribution available from http://www.kernel.org. Included in distributions is compiler support for the target processor architectures, along with user mode libraries for application startup. The GNU C Compiler project, at http://gcc.gnu.org, maintains up-to-date versions of compilers for each architecture.

Embedded Linux distributions typically provide cross-development support for a target platform from a host environment. Because the GCC compiler runs across a wide variety of platforms, host development platforms running Linux, Solaris, and Microsoft Windows are becoming available. Sophisticated development tools, such as the *make* program maintainance tool, allow complete applications to be cross-compiled and linked using a single command. A big advantage of using a Linux based host development platform when developing a Linux-based target application is that in many cases, major parts of the target application can be built and tested on the desktop while designing and building the target hardware device. Using graphical windowing environments like the X Window System and Microwindows allow complete graphical emulation of the target graphical hardware without having to cross-develop and download every change. Some of these development features are unique to Linux and are further pushing its advancement and use in leading embedded systems designs.

In the next part of this paper, I will give a short discussion of various parts of the Linux operating system and some graphical windowing system alternatives.

The Linux Kernel

As mentioned previously, the Linux kernel provides support for memory management, process and thread creation, interprocess communications mechanisms, interrupt handling, execute-in-place ROM filesystems, RAM filesystems, flash management, and TCP/IP networking. The Linux kernel provides a POSIX-compliant API to these services. The directory structure of the kernel source tree separates architecture-dependent code out from the core kernel services, allowing greater reliability with known-working core algorithms, with calls to machine-specific code added for particular platforms. Thus, adding support for specific device features is fairly straightforward. This implementation methodology is also followed for memory management, i/o, and driver designs, where the core kernel code abstracts a model that allows implementation on differing architectures.

Memory Management

The kernel provides complete modern virtual memory services to applications programs, including support for large address spaces, protection, demand paging, memory mapping and shared virtual memory. While support for large address spaces or demand paging may not seem important for embedded systems designs, all of the modern 32-bit processor architectures support these features, and Linux will allow growth in application complexity as hardware costs are reduced without redesign or reimplementation. Memory protection allows building systems that allow user-upgradeable or third-party applications to be added to the system, without compromising the entire system. Shared virtual memory support allows multiple copies of application s code segments to be shared across the system using less physical memory, as well as implementation of more sophisticated schemes like high-speed direct application framebuffer access for MPEG digital video players, for instance.

Processes and Threads

Linux provides a relatively cheap process creation mechanism, which allows memory-protected processes and threads to be created quickly for a variety of uses. Recent embedded implementations provide soft real-time scheduling services for applications programs. In most cases, these scheduling services provide all that is required for applications requiring networking, i/o and graphical services. Each process in Linux has its own table of open files and virtual memory allocations, although files and memory can be shared between processes.

Interprocess Communication

The kernel provides signals, pipes and sockets for communications between applications. The signal mechanism allows user programs to be asynchronously notified when a specified event occurs. Signals can be sent to single processes or process groups. The pipe mechanism implements a full-duplex facility for arbitrary byte i/o between processes. Pipes are also very useful in connecting one programs output to another s input, and constructing data filters. Linux sockets act just like file or pipe descriptors but instead communicate to the networking subsystem. A specialized type of socket, known as a *local UNIX socket*, allows communications between local processes similar to the pipe mechanism but using the networking system calls instead. Using a socket creation parameter, processes communicating together on a local machine can be replaced by processes communicating between two different machines, using the networking system, without other architectural changes.

Interrupt Handling and Device Drivers

Linux provides sophisticated methods for interfacing the system with hardware. The interrupt handling architecture allows handling high-priority interrupts with interrupts disabled, with scheduling of later bottom-half code to execute in kernel mode after interrupts have been enabled and other higher-priority kernel tasks completed. In addition, support for polling and DMA is provided. Adding support for a specific hardware device is implemented using a device driver, which may be linked into the kernel, or dymanically loaded as a kernel loadable module. At system startup time, scanning for various hardware options can be performed, and drivers loaded as required.

RAM/ROM Filesystems

The Linux kernel implements a virtual file system that allows the implementation of various real filesystems accessible with a common interface. Standard disk and network file systems are supported, along with filesystems for ROM and RAM residence. A compressed ramdisk image for system startup can reside linked with the Linux kernel, or copied from a compact flash card, for instance. In addition, implementations for execute-in-place ROM filesystems are becoming available. Read-only compressed filesystems can be decompressed partially as required, resulting in fast boot times. Filesystems can also be created for initial boot purposes and then discarded, resulting in more memory use after one-time system initialization.

TCP/IP Networking

A very complete implementation of the standard TCP/IP networking protocol suite, including TCP, UDP, IP, and ARP is found in the Linux kernel. In addition, complete support for NFS, DNS, DHCP, PPP and SLIP is included by applications programs. Linux s networking support and configuration customizability make Linux well suited for the mobile and Internet appliance markets, where data communications is a key technology.

Graphical Windowing Environments

In the fast-changing world of embedded, handheld and wireless devices, there are many hardware and software design changes taking place. Many devices now feature 32-bit microprocessors from Intel, MIPS, Motorola and Hitachi, as well as larger LCD graphical displays. In order to leverage the significant results gained in the desktop arena the last ten years, many developers are turning to using desktop-like operating systems with these new embedded designs. One of the most promising emerging areas is running Linux in these environments, for a couple of good reasons: Linux on embedded systems brings with it the entire power of desktop computing, along with many solutions already running. Linux, being open source, allows any aspect of the solution to be fully understood and then customized for a particular application.

The standard graphical windowing environment used on all desktop Linux systems is the X Window System, originally developed by MIT, DEC and others in the early 1980 s. Although powerful and relatively quick, X is not without its problems.

Problems with the X Window System

Having a freely-usable, open source, high-quality operating system with a TCP/IP stack sets the stage for interesting, innovative applications using mobile devices and Internet appliances. However, the resource requirements of these new applications comes under scrutiny, since most embedded devices lack hard drives and large amounts of RAM. The X Window system features a client/server architecture that allows applications running on any UNIX host (clients) to display on an X-based terminal (server). Quite a bit of code is devoted to making sure that the client/server paradigm works well across different host and server environments. These days, typically the X clients and server run on the same machine, the user s desktop. The X Window Server is typically compiled to support differing hardware chipsets with auto-detection, as well as having multiple copies of all drawing functions both above and below the clipping layer for high speed. As a result, the X Window System is large and complex, typically requiring multiple megabytes of memory for execution.

Microwindows — An Alternative to X

In most embedded designs, the X Window System is overkill, especially when running a dedicated graphics application. For this reason, the Microwindows Project was developed (http://microwindows.org). Microwindows is an open source project aimed at producing desktop-quality graphics functionality for small devices. The architecture allows for ease in adding different display, mouse, touchscreen and keyboard devices, as explained below. Starting with Linux version 2.2, the kernel contains code to allow user applications to access graphical display memory as a framebuffer, which ends up being a memory-mapped region in a user process space that, when written to, controls the display appearance. This allows graphics applications to be written without having to have knowledge of the underlying graphics hardware, or use the X Window System. This is the way that Microwindows typically runs on embedded systems.

Microwindows Features

Microwindows was designed to attempt to bring applications to market quickly, with a minimum of effort. In order to accomplish this, it was felt that designing yet another graphics applications programming interface (API) would steepen the learning curve, thus discouraging interest and increasing time-to-market. Microwindows implements two popular graphics programming interfaces: the Microsoft Windows Win32/WinCE graphics display interface (GDI), used by all Windows CE and Win32 applications, and an Xlib-like interface, known as Nano-X, used at the lowest level by all Linux X widget sets. This allows the extremely large group of Windows programmer talent to be used in developing the graphical side of the application, as well as being familiar to the core group of Linux graphics programmers used to working with X.

Microwindows runs in 100-600k of memory, and including all associated libraries, is an order of magnitude smaller than the X Window System. This is primarily the result of using a single routine for each of the drawing functions in the engine layer. The engine layer checks for clipping and calls a driver-level routine for drawing a pixel or line if unclipped. The X Window System duplicates the entire drawing routine for each pixel depth and has clipped and unclipped versions for speed. This is useful when running large desktop displays, but rarely required for embedded systems.

Microwindows fully supports the new Linux kernel framebuffer architecture, and currently has support for 1, 2, 4, 8, 16, 24, and 32 bits per pixel displays, with support for palettized and truecolor display color implementations, as well as grayscale. With both APIs, all colors are specified in the portable RGB format, and the system has routines to convert to the nearest available color, or shade of gray for monochromatic systems. Although Microwindows fully supports Linux, it s internal, portable architecture is based on a relatively simple screen device interface, and can run on many different RTOS s as well as bare hardware. This brings a big benefit: graphics programming by the customer can be shared between projects, and even run on different targets with different RTOS s, without having to rewrite the graphics side of the application.

Microwindows supports host platform emulation of the target platform graphically. That is, Microwindows applications for Linux can be developed and prototyped on the desktop, run and tested without having to cross-compile and run on the target platform. This is accomplished using Microwindows X screen driver, where the target application is run on the desktop host and displayed within an X window. The driver can be told to emulate exactly the target platform s display in terms of bits per pixel and color depth. Thus, even though the desktop system is 24 bit color, it can display a 2bpp grayscale for previewing the target application. Since both the host and target are running Linux, most all operating system services are available on the desktop host.

There are significant differences in the way that programmers familiar using Microsoft Windows or the Linux X Window System implement graphical applications. Microsoft Windows programmers typically use Microsoft s Foundation Classes (MFC) C++ applications framework, or the newer Active Template Library (ATL) framework. Source code is provided for both of these interfaces, and both must use the Win32 GDI for all graphics drawing. Windows also includes quite a few user interface controls that are useable from the Win32 GDI, including buttons, listboxes, and the like. The X Window System, on the other hand, provides a low level interface known as Xlib, which implements low level drawing primitives only, and packages them up for execution on the display server. Most user interface solutions rely on widget sets implemented on top of Xlib, for modern functionality. We are working at porting some of the more popular widget sets to Microwindows, including GTK+/GDK, the basis of the GNOME project s desktop, and FLTK, a small C++ toolkit which implements quite a few user controls.

Microwindows Architecture

A key component in the design of Microwindows is to keep things small. Although this can sometimes preclude implementation of more complex applications, the Microwindows design has distinct separation of driver level, engine level, and API level functionality so that, if desired, complex functionality can be added on an as-needed basis without complicating the entire design. Each of these implementation levels will be discussed in more detail below.

At the lowest level, Microwindows abstracts a data structure for a display screen, a mouse or touchpad input device, and a keyboard device. This structure appears identical on the top side, for use by the drawing engine, and includes code to interface with the specific device the driver is written for. Microwindows includes drivers for quite a few different devices and operating systems, with more being included with each release. Screen drivers for the Linux framebuffer include support for displays running 1, 2, 4, 8, 16, 24 and 32 bits per pixel, supporting palettized, grayscale, and truecolor display. The screen driver definition includes entry points for reading and writing a pixel, drawing a horizontal or vertical line, and optionally, blitting memory from the screen to memory or vice versa. By implementing only these basic entry points, all of the upper level Microwindows functionality, including TrueType or Adobe Type 1 font support, RGB colors, jpeg and bmp image handling, will run. If your screen device has hardware acceleration, or a peculiar method of implementation in one of these areas, a modication to the driver is all that is required. In the same manner, touchscreen and keyboard/button input is brought into the system.

At the middle level of Microwindows is the drawing engine. This code is device-independent, since all drawing is implemented by calling a screen driver. The drawing engine presents a set of standard entry points for the API level above to call for drawing functionality. The Microwindows engine abstracts an RGB color model for all colors, regardless of the physical display device. Routines are provided that determine the hardware pixel value from an RGB triple, using the same interface regardless of whether the physical display is running pallettized grayscale or truecolor. In addition, Microwindows implements all clipping at the engine layer, using a fast, multiple-rectangle approach that allows for arbitrarily complex regions to be specified for drawing. All font display is handled at the engine layer, including our new support for Unicode-based TrueType fonts using FreeType, and Adobe Type 1 fonts using T1Lib. Compiled in proportional fonts are also supported. The engine implements anti-aliasing using alpha blending for smoother looking fonts on higher resolution displays. Currently, support for Ascii, Unicode-16, Unicode-32, UTF-8, Chinese GB2312 and Big5 encoding are supported.

The uppermost level of Microwindows implements one of the two supported APIs. At this level, the window abstraction is implemented, which allow applications programmers to contain their display data in a full screen or overlapped window. This layer is also responsible for event handling, and passing received hardware events such as touch screen or button presses to the application.

The Nano-X API allows applications to be built using a client/server protocol over a network socket, or local UNIX domain sockets. This allows several applications, running on the embedded device or a remote host to connect to the Microwindows server to for display. In addition, the client/server protocol can use shared memory for passing data between the client and server. A compile time option allows all applications to be linked with the server for smaller environments, or environments without a filesystem. Microwindows also supports linking directly with a real time operating system for extremely compact environments. Compilation for various features is turned on or off in a configuration file, which allows Microwindows to be built in modules, and include only the items that will be used.

ViewML Embedded Web Browser — An Embedded Linux Application

The ViewML Browser Project, created by Century Software, is an open source project whose aim is to produce a small footprint, high-quality web browser targetted towards the needs of the embedded Linux applications developer community. The project is discussed in this paper as an example of a key applications technology implemented on the Linux platform for mobile and embedded Linux systems. ViewML runs on both the X Window System and Microwindows, and provides a highly customizable web browser for use in Internet appliances, PDAs, and other mobile applications. In this section, a brief summary of the design goals of the ViewML project are presented, along with some issues we confronted during the design.

ViewML Design Goals

The family of available desktop browsers for Linux has evolved into quite a clan, with over 20 member browsers vying for attention. Why, then, introduce another one? After surveying the field of available browsers, in search of even one appropriate for embedded deployment, we found that no single web client would suit. Browsers were either too large, like Netscape's Mozilla, and would never run on most embedded systems, or too small, with very incomplete HTML parsing and lacking in other capabilities. So we decided to design a new browser, one that was specifically targeted at the needs of the embedded Linux community.

The initial design goals for the project were:

- Create the smallest browser possible, but retain 100% standards compliance for HTML parsing. The browser would be used in many applications from embedded-device documentation display to Internet appliances and set-top boxes. We had to make sure that the browser always displayed pages correctly.
- Use available open source code for the HTML parsing and display engine. We didn t want to get into the business of writing an HTML engine from scratch, the most common pitfall of most smaller browser implementations. It takes a lot of knowledge and experience to display all the HTML language quirks correctly, especially since so much HTML is still written by hand.
- Use the selected HTML widget code as-is. We didn t want to change any of the core HTML display engine code, even though it is open source. This bought two major benefits: the ability to upgrade the HTML display capabilities as the original parsing engine is enhanced by HTML experts. It also meant that no bugs would be introduced directly in the core display routines, keeping the quality high.
- Use the Fast Light Tool Kit (FLTK) applications framework for the user interface. FLTK provides a set of user-interface widgets ideally suited for small environments.
- Run on both Microwindows and the X Window System. In order to gain large acceptance, the browser would need to run on the standard X Window System as well as Microwindows. In addition, we wanted to make sure that the selection of either windowing system was seamlessly integrated into the software design, and didn t adversely affect the architecture.

FLTK Applications Framework

Two different versions of the FLTK applications framework are used, depending on the windowing system used. Standard versions of FLTK include support for Win32 and X. Century Software and Microwindows project contributors ported FLTK to the Nano-X API available in Microwindows. This support allows client/server interaction with the Microwindows server, just like the Xlib model. Choosing FLTK is a great choice, since both FLTK and Microwindows support the X Window System. This allows the ViewML browser to be debugged or enhanced on the Linux desktop, using either the X Window System directly with FLTK, or running the Microwindows server on top of X. In this way, the exact characteristics of the target environment, whether running Microwindows or X, can be emulated. We also like the idea of being able to run almost the identical code paths on the desktop as the target device, which greatly improves quality control.

ViewML Summary

The ViewML Project has produced a high-quality web browser in a short amount of time, directly targeting the embedded and mobile Linux environment. By including open source core components, we ve been able to use a high-quality display engine while keeping the overall RAM and ROM requirements quite low. Currently, the ViewML browser runs in about 2MB of RAM while having a ROM file size of around 900k. Combined with Microwindows, the entire environment can run in less than 2.1MB RAM, which allows its use on most 32-bit embedded Linux systems running graphical displays. With the entire ViewML project in open source, other contributors will quickly join the effort and enhance ViewML even further. By leveraging the vitality of the open source community, ViewML can meet the challenge of providing a high-quality web browser for the embedded Linux environment.

Conclusion

The Linux operating system is well suited for use in the rapidly growing embedded computing market. It s technologically advanced kernel, open source development model, free availability and royalty free distribution make it an ideal choice for future designs. The large developer environment and fast pace of contributions ensure that Linux will meet the requirements of emerging embedded and mobile applications for some time to come.

Links

www.kernel.org	Linux kernel distribution
gcc.gnu.org	GNU compiler collection
microwindows.org	The Microwindows Project
xfree86.org	The Xfree86 Project — X Window System
www.viewml.com	The ViewML Embedded Browser Project
www.kde.org	KDE Desktop Project
www.gtk.org	GTK+/GDK Widget Set
www.fltk.org	Fast Light Tool Kit Homepage
embedded.centurysoftware.com	Century Software Embedded Technologies
www.linux4.tv	Linux Open Source Interactive TV Project