Using Linux to Implement 8- and 16- Bit Device Networking Solutions

Glen Mann
Extend The Internet Alliance FAE
emWare, Inc.
6322 South 3000 East, Suite 250
Salt Lake City, UT 84121

Abstract:
By using gateway systems on large 32-bit platforms, networks of small, 8- and 16-bit microcontrollers can be monitored and controlled over the Internet. With embedded Linux, these gateways are easily moved from full-blown host PCs to embedded platforms like the PC104. In this class you will learn about hardware platforms that support embedded Linux, Linux kernel configuration, feature selection, installation, booting and tuning.

The Gateway Approach

A gateway system provides an embedded architecture that provides networking capability to even the smallest of embedded controllers. By establishing a gateway that acts as an intermediary "broker" between lightweight device networks (RS232, RS485, Modem, IR, RF) and heavyweight networks, including intranets and the Internet, you off-load the device and achieve the desired communication. The gateway can reside on a PC, single board computer, handheld, or a device with sufficient resources (32-bit microprocessor). The gateway, having more resources, can augment a device and enable more information and data to be exchanged. (A Gateway Method, Figure 1)

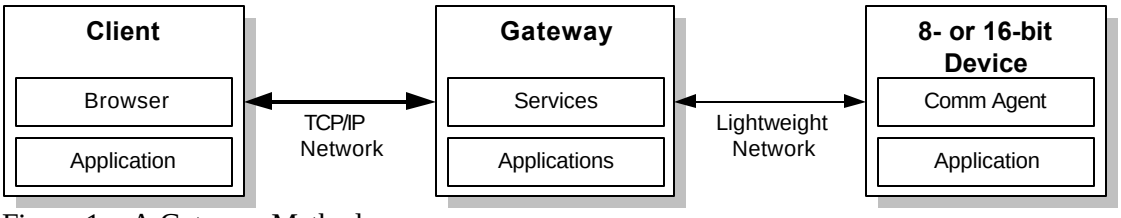

Figure 1 – A Gateway Method

A gateway provides device access and management services to nodes on the Internet or an intranet. A gateway method for networking embedded devices has a number of unique characteristics that it can provide:

- *Device Firewall*. The devices can rely on a gateway process to provide authentication and encryption where appropriate. In addition, the gateway can augment the security on the device by enforcing the most current security policies in this rapidly changing, Internet-connected world.
- *Protocol Translation.* The gateway can provide methods for adapting to the variety of device connectivity hardware and software solutions and delivering a uniform TCP/IP based access service.

- *Device Watchdog*. An optional service of the gateway is monitoring device status. Whether the devices are to remain connected at all times and the gateway reports failure of the ancillary network or device, or the device connection is required to periodically contact the gateway process, the gateway reports to a designated client when a device becomes unavailable.
- *Event Handling.* A particular event from a device may require a launch of an application, or dispatch of a pager or e-mail message. In this situation, there is no connection to a "client" waiting for a message.
- *Device Services*. Instead of viewing the network as a client of an embedded application, it may be desirable to offer a set of network services to an embedded application. The gateway acts as a provider of services to embedded device applications.
- *Support for any lightweight network*. Just like embedded processors, there is no single physical network transport that is appropriate for all environments. Whether it is cost, communications robustness, security, or a myriad of other reasons, solutions for networking embedded devices must support a wide variety of networking types that offer a range of capabilities**.**
- *Wide Variety of Interface Options.* There are countless methods to interface with devices including Windows, the Web browser and the phone. There is no such thing as the "ubiquitous interface" for embedded electronic devices. For certain types of applications, a simple, text-based HTML interface may be sufficient, however, in other types of applications it may be overkill or not sophisticated enough. To provide rich, device-networking solutions, human and machine interface options need to be provided that scale from DTMF—phone key—to complex enterprise databases and resource planning applications—such as SAP R-3. In addition, many of these "interfaces" may be used with these devices at the same time.

Five aspects of a gateway-based system to consider include the following points.

Gateway: Any Processor
There are more than 1,000 “flavors” of microcontrollers currently on the market. Each one is designed to cut costs and service a certain set of functions. There is no single processor or processor family that can service all embedded system requirements. In many cases, assumptions are being made in the embedded market that higher-end processors are required for any form of communications external to the embedded device – but the major logic flaw is that this is OK due to the ever-lowering cost in higher-end processors due to Moore’s Law. We have seen OEM manufactures that are “sweating” the decision of a $0.50 8-bit microcontroller vs. a $1.00 8-bit microcontroller for a high-volume embedded application. The lowest cost 32-bit microprocessor was not even in consideration with costs only starting at 10x to 20x the 8-bit cost.

Moore’s Law is being misapplied. We can’t assume that 32- and 64-bit processors will drop so low in price—to pennies—and still deliver incredible performance. Even if cost is not an issue, there are still the issues of power consumption, package size, thermal dissipation, and other “embedded” requirements that are difficult, if not impossible, for larger processors to satisfy. At the same time, Moore’s law also applies to 8- and 16-bit microcontrollers. These smaller MCUs are gaining additional capabilities while also costing less. Every device and application has different needs, therefore, a large array of processor choices with different performance, power requirements and capabilities is critical.

Gateway: Any Capillary Network
Lightweight, capillary networks using transports such as RS232, dial-up, 485, RF, IR and CAN must all be supported to provide adequate options for networking 8- and 16-bit devices. Just like

embedded processors, there is no single physical network transport that is appropriate for all environments. Whether it is cost, communications robustness, security, or a myriad of other reasons, solutions for networking embedded devices must support a wide variety of networking types that offer a range of capabilities. In addition, networks for 8- and 16-bit-based devices need to be able to operate in low-power, transient environments where devices may or may not have continuous access to a network. The method by which these lightweight networks communicate with larger, wide-area networks, including the Internet, must be efficient and timely. Unlike Ethernet and TCP/IP, which are proving to be an effective single-communications method for larger computer systems, there is no single lightweight network that is appropriate for smaller electronic devices.

Gateway: Flexibility

Flexibility - the architecture must be flexible, scalable and can be configured for a wide variety of applications, including existing devices and proprietary networks. The user interface can operate on a remote Web browser, a directly connected laptop, or even a handheld PDA. Where a user interface is not necessary, it can connect directly to an application or database. By utilizing a gateway approach, implementers can also have flexibility on where interface descriptions are stored. With a gateway, the interface can be fully described and stored on the gateway, eliminating any resource requirements for storing the interface on the device. Or, the gateway can be used to augment partial interface description such as a tagging method.

Gateway: Any Interface

There are countless methods to interface with devices including Windows, the Web browser and the phone. There is no such thing as the "ubiquitous interface" for smaller electronic devices. For certain types of applications, a simple, text-based HTML interface may be sufficient, however, in other types of applications it may be overkill or not sophisticated enough. To provide rich, device-networking solutions, human and machine interface options need to be provided that scale from DTMF—phone key—to complex enterprise databases and resource planning applications—such as SAP. In addition, many of these "interfaces" may be used with these devices at the same time.

Gateway: The Object Model

An object model for lightweight devices is required to accommodate the diversity of embedded devices, the different transport systems and collaboration requirements, creating the building blocks for making the integration of multiple object environments (CORBA, JINI, and DCOM) possible. This will allow for even greater sharing of information and control.

The end result for embedded device communication should be an object representing the capabilities of the device. Everything that the application engineer wants to make public is exported from the embedded application to a device object server and then through a Gateway to applications on the Internet or an intranet.

In many applications it is desirable to treat embedded devices as abstract data types. By encapsulating functional behaviors of a particular device type, and standardizing around these encapsulations, we create an environment that will foster the development of network-client applications. These applications use an object model and definition of a particular application function to de-couple the client interface from the manufacturer's device. The gateway provides access to the device through an object service, and isolates the client from network and device-specific considerations. A home management software package, which could run on a PDA or browser, may access a heating control unit based on the object definition of an HVAC. The gateway provides the necessary object interface to an HVAC and isolates the client from

the specific network and device characteristics.

Gateway Pros:	**Gateway Cons:**
Supports any appropriate networking transport (RS232, RS485, RF, IR, CAN, …)	Possible Software Distribution Issues
Leverages TCP/IP standard and ISP infrastructure	Distributed Networking can be more complex
Leverages HTML and Java Development Tools	
Supports Internationalization and multiple HTML page interfaces	
Can provide robust APIs for applications and databases to access and control devices	

Operating Systems and Gateways

As microprocessors and microcontrollers have evolved to meet specific industry needs, so too has the variety of operating systems. In the early days, engineers had to build their own OS from the ground up. Today, that option still exists but is often passed over in favor of commercial off-the-shelf products. Buying an OS allows designers to concentrate on the application rather than the OS and the application. Time to market pressures are eased significantly with an OS out of the way and the path cleared to build the application. Having a pre-built OS allows the engineer to select from a variety of host processors if the OS vendor has support for the various CPUs.

Recently, the embedded world has been able to benefit from developments in the open source movement around the Linux operating system. The rapid adoption of this operating system has added support for "real time" functions for embedded design. The flexibility of the Linux operating system allows the user to implement only what is needed.

The name Linux is a compilation of Linus Torvalds and UNIX™. Linus is widely credited for creating Linux. However, Linux is the creation of thousands of programmers and the free software efforts of the GNU Project (see www.gnu.org). The concept of Linux is that the community of programmers and users would benefit from having access to the source code of the operating system such that features one group may find desirable could be quickly implemented for the benefit of the group. The architecture of Linux is strikingly similar to Unix. This is not a coincidence. The Linux system commands, directory structure and applications that are supported are typically identical to Unix. Most applications built for Unix systems run on Linux with a simple re-compile of the source to the specific version of Linux. The reason Linux is not branded Unix is due to the fact that Unix is a trademark of AT&T. Unix is a brand only allowed for software which is POSIX compliant and the vendor has paid a royalty to AT&T for the right to use the Unix brand name.

Linux has been built on the POSIX standards. Since no one owns Linux and the effort is often the result of free volunteers, there is no single entity to "pay" a Unix fee to AT&T or pay for the testing. Several distributions have become available from commercial concerns but those groups do not "own" the source code, which is the heart of Linux due to GPL. GPL is the General Public License which entitles everyone to take and improve the code, but must supply source code for any modifications for the benefit of the group. Details on the GPL can be found at www.gnu.org or www.linux.org.

The GPL program does not require a vendor to share their application source but any modifications to the Linux kernel source to enable the application must be shared for the benefit of the group.

Key features of Linux:

- Multitasking
- Multithreading
- Shared Library
- POSIX compatibility
- TCP/IP networking/Routing

Some of the reasons a designer would choose Linux over Windows NT as an operating system are as follows:

- The need to run on a wider variety of MCUs and CPUs
- The need for small code size approx. 450 KB program storage for the OS and 4-8 MB RAM (See table 1 for details)
- The need for a stable system
- Remote administration without being local to the machine to gain "root" privileges
- Resistance to viruses and security violations
- Ability to take out the "bloat" of unnecessary portions of the OS

Table 1

Requirements	Miniscule	Tiny	Mid-range	High-end Embedded	Embedded PC	Desktop	Server	High-End Server
RAM in MB	0-0.1	0.1-4	2-8	8-32	16-64	32-128	128+	Lots!
ROM/Flash/Disk	0.1-0.5 MB	.5-2 MB	2-4 Flash MB	4-16 Flash MB	MBs	GBs	GBs	GB-TBs

Source: Lineo

The option to press the reset button on an embedded system is typically NOT acceptable. A highly reliable system starts with complete control over every aspect of the embedded system. The base layer of Linux under the GPL license conditions allows access to the source code of the kernel and other vital system components. A Linux kernel can be "tuned" to meet the needs of the system's feature and performance requirements. Often an experienced Linux user will get a system running to insure the application runs correctly and then will re-compile the kernel of the OS to optimize the performance of the system. Unnecessary components will be stripped away.

There has been a big push to put the open standards of the Internet into embedded devices. One of the glaring trends in the Internet-enabling saga is the desire to place the native protocol of the Internet (TCP/IP) onto the device. This is at first a rather simple way of setting things right with support of an "open standard" and moving on to build the application on the device. Upon closer inspection however, it turns out that a wide-area networking protocol such as TCP/IP or UDP/IP

or SNMP over IP is overkill for the majority of applications that are perfectly supported via a lighter-weight networking protocol. What the cost conscious embedded engineer really needs to solve the problem is a simple lightweight protocol to address the small network issues. In order to tie to the larger network it makes more sense to let a server handle the heavyweight protocol and other services on a more capable server. What is the point in expending 250 KB of code space just to handle the TCP/IP stack on the device when 90% percent of the protocol's capabilities are not even used? Several clever "hacks" have been devised to wrap a TCP/IP packet header around a lightweight protocol. However, this places the burden at the end of the network that must, in turn, be managed by some sort of gateway.

Why not just place embedded Linux down onto the embedded device and run SNMP? The Linux "footprint" can be small by conventional server standards and Internet appliances are starting to show up on the market for a "reasonable" consumer price. Combined with Moore's law, it would seem reasonable to simply save oneself a lot of extra work in cramming an embedded system into a low cost 8- or 16-bit microcontroller and spend it on higher level general programmers targeting larger processors. This would make the product faster to market and reduce the dependence on dedicated embedded hardware designers. It would also enable the development system and the target systems to be very similar, and it would save the engineer from becoming concerned about managing the complex interaction of peripherals and the application on the MCU.

The problem is, there are so many devices that need simple, effective functionality without a large cost associated with the hardware, that it still makes sense to hand-craft an embedded solution with emphasis placed on software overhead at the embedded device. As a testament to the volume of products that do not use a processor capable of running Linux or similar large OS, consider that the annual volume of 8- and 16-bit microcontrollers is 3.29 billion units vs. 0.24 billion 32 bit MCUs/CPUs. Linux, RTOSes, other OSes and even Sun's Jini virtual machine typically require the latter 32 bit processors. See figure 1.

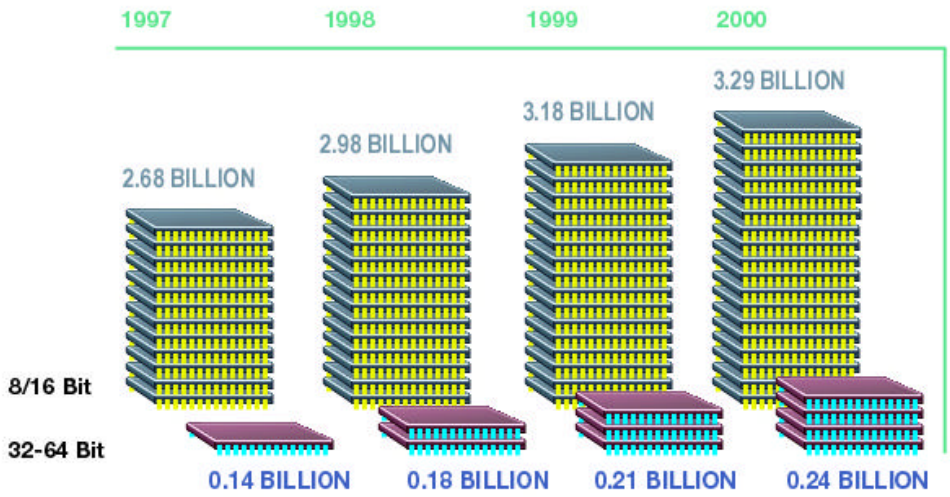

Fig. 1
Source: International Architects WSTS & ICE

With all the industry talk about Moore's Law making 32-bit MCUs viable for low cost applications, it's interesting to note just how dramatically different the volumes are between the 8- and 16-bit markets vs. 32-bit. What drives this delta in volume?

Consider the cost issues for a product that uses large quantities of 8-bit MCUs. The automotive industry is a large consumer of 8- and 16-bit microcontrollers. A single modern automobile uses up to 80 8-bit MCUs for a variety of functions including safety (airbags), antilock brakes, suspension and engine controls.

One of the challenges facing a designer when building a complex embedded application is keeping track of all the critical time and resource related processes. An operating system is often selected to take care of this peripheral administration.

Consider a simple design trade-off for an airbag controller that needs a UART function to communicate to other systems on the vehicle. The designer may choose a hardware UART and pay more to the MCU vendor for this luxury. Or, the engineer could write a software UART to manage communication and place more of the burden on the MCU for communications. Finally, the software UART and other peripheral communications tasks could be left to an operating system from a 3rd party so the designer is free to concentrate on the application development. This last option may push the MCU requirements up to support an OS with additional overhead. As a result many embedded design cost trade-off choices come down to time-to-market and cost-per-device issues.

For some applications it's more cost effective to pay a clever engineer to make a low-cost solution than to find an OS (i.e. Linux or an RTOS) and insulate the designer from the peripheral management details in order to focus on the application.

It is this kind of super cost sensitive design that drives many OEMs to create a simple lightweight networking model. The designer can choose to "outsource" the communications methods or spend valuable time creating a protocol and communications algorithm to interact with other devices in the network. Just as an OS may make sense for a designer to save time, it has now become possible to use an off-the-shelf embedded device-networking model that has been established.

Linux is a key component in this kind of cost control strategy. An engineer may not place an OS on the 8-bit device for the application, but may instead rely on the communications power of the Linux OS to provide a platform for connectivity to the outside world.

The x86 processors are the dominant platform for Linux due to the history of Linux. However, an embedded system is not always best served by an Intel x86 CPU architecture. Linux is well supported on other CPU platforms. A sample list includes PowerPC, ARM, Dragonball, MIPS, Atmel and Transmeta's TM3120 Crusoe processor and various other RISC architectures.

Linux offers the convenience that an application developed on a x86 platform can be readily ported to the target system without a large amount of re-work to the code. This portability is one of the key reasons companies value supporting the embedded Linux server approach.

Case Study: Automotive Design Engineer

An automotive design engineer is testing an 8-bit MCU based design to go into production. There are two key design issues, cost and product field upgrade.

For the cost issue, there is the trade-off of building masked ROM versions vs. newer FLASH based MCUs. In addition there is the issue of product upgrade or recall. On the first issue the designer takes advantage of Moore's Law and chooses a flash-based MCU over the traditional masked ROM method of cost reduction.

In performing the design, the designer implements remote connectivity methods that are appropriate. The production product will not necessarily benefit the end user on a daily basis by having the remote connectivity feature. The engineer realizes that this is the case and so chooses a lightweight method that will not significantly impact cost of the device performance or his own time to market concerns.

What other benefits does remote connectivity offer the designer? It turns out the engineer has provided this remote connectivity method to deal with three other cost drivers.

1. Time to market
   Reduction of travel time to visit the "proving grounds" and make design changes based on first hand field observations.

2. Government and Insurance companies
   Remote connectivity allows the device to verify ROM revision and compliance with product recalls.

3. Feature creep
   The marketing department has discovered a new feature that will better differentiate the product in the market.

Because of this "backdoor" diagnostic mechanism, the engineer has insured a path to observe and change the application on the FLASH-based MCU. The diagnostic equipment vendors have designed in the capability to inspect and upgrade vehicles quickly with a standard method of interface.

So where does Linux come into the picture?

Due to the variety of interaction points, it is vital that each client be able to access the device with the appropriate interface. The engineer designed the product on a x86 based Linux workstation, the OEM field technician used a 32-bit CPU based handheld instrument to field test and gather data.

The repair/recall dealer used a 32-bit CPU handheld instrument and a sophisticated 64-bit CPU database to connect "online" to the factory database to "flag" the suspicious MCU. The fix was performed and all concerned parties were informed of the successful 8-bit target MCU update via e-mail, verbal, paper mail and reports. The Linux OS supporting these connections was used on the two portable instruments, the engineer's desk and the database. In each instance, the gateway running on the Linux OS was able to be scaled to the respective 32-bit CPU and 64 bit platforms. The gateway was simply an application on the Linux OS servicing connectivity to the 8-bit target device.

The diagnostic tool vendor could not justify the cost, bulk and power requirements of a traditional PC for the field technician or the dealer service person. Linux was chosen to enable the low-power and low-cost CPU to provide the needed instrument functionality.

Linux running a gateway serviced the connection to the embedded device with the appropriate level of CPU resources. Additionally, the e-mail and large database applications were served via the gateway on Linux via TCP/IP connectivity.

Example: Building Automation

Consider the value of the following scenario.

An employee walks into his office on a weekend during non-business hours. The employee presents a valid "key" to the building access security system. The security system is disarmed and the employee takes the elevator to his office on the 5th floor. The rest of the building is still secured and the lights and HVAC systems are on low-power standby mode. However, the 5th floor lights and air conditioning systems have anticipated his arrival and have gone into active mode. The value of the three separate systems interacting presents a good business case for integrating the communications between these systems. Further, a case can be made for a new program offered by the building's energy provider. The energy provider proposes an annual savings of 30% via "load shedding" during peak hours. In order for the load shedding program to work however, the energy provider must have the capability to control the discretionary loads.

Load shedding allows the power provider to reduce costs during peak power usage when the end user allows the supplier to signal a need to turn off the power consumption on the discretionary loads.

The key in the above scenario is that the devices must have a communications method that is accessible via low cost "open standards". TCP/IP is the open standard that will best interface to the energy provider's sophisticated load management software at their offices. However, TCP/IP at the device for the end user would require a 32-bit CPU for each and every device. The cost savings would be hard to recoup if this were the method on the devices. A gateway approach makes more sense in this situation.

The cost issues for this project demand that the following devices are strictly controlled for their cost. A motion sensor (8-bit), electronic lighting ballasts (8-bit), building access control (8-bit), HVAC system (8-bit).

A system is proposed that must allow the security system to interact with the lighting and heating so the above scenario can be realized in a cost effective way.

Using methods described earlier, the reader will see that the gateway method makes the best use of the limited resources at the device.

Summary

With the addition of support for the Linux embedded environment, a gateway can be easily supported on the range of platforms that Linux offers. A designer using a gateway method is free to select the best fit for the particular design requirements.