

ABSTRACT

The embedded system is a combination of computer hardware, software and, perhaps, additional mechanical parts, designed to perform a specific function. A good example is an automatic washing machine or a microwave oven. Embedded systems need only the basic functionalities of an operating system in real-time environment—a scaled down version of an RTOS. They demand extremely high reliability plus the ability to customize the OS to match an application's unique requirements. However, commercial RTOSes, while designed to satisfy the reliability and configuration flexibility requirements of embedded applications, are increasingly less desirable due to their lack of standardization and their inability to keep pace with the rapid evolution of technology. The alternative is: open-source Linux. Linux offers powerful and sophisticated system management facilities, a rich cadre of device support, a superb reputation for reliability and robustness, and extensive documentation. Also, Linux is inherently modular and can be easily scaled into compact configurations.

INTRODUCTION

Computers have evolved from a few, huge mainframes shared by many people, to today's PCs -millions in number, miniscule in size compared to the mainframes, and used by only one person at a time. The next generation could be invisible, with billions being around and each of us using more than one at a time. Welcome to the world of embedded systems, of computers that will not look like computers and won't function like anything we're familiar with. An embedded computing system uses microprocessors to implement parts of functionality of non-general-purpose computers. Early microprocessor-based design emphasized input and output. Modern high performance embedded processors are capable of a great deal of computation in addition to I/O task. Microprocessors that were once prized centerpieces of desktop computers are now being used in automobiles, televisions and telephones. The huge increase in computational power can be harnessed only by applying structured design methodologies to the design of embedded computing systems. Historically, Linux was developed specifically as an operating system for the desktop/server environment. More recently, there has been a growing interest in tailoring Linux to a very different hardware and software needs of the embedded applications environment.

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.

EMBEDDED SYSTEMS

History

The computers, which are used to control equipment, or embedded systems, have been around for almost as long as computers themselves. The word “computer” was not as ubiquitous back then, and the stored program referred to the memory that held the program and routing information. Storing this logic, instead of hard-wiring it into the hardware, was a real breakthrough concept. Today, we take it for granted that this is the way things work. These computers were custom-designed for each application. By today’s standards, they look like a collection of mutant deviants, with strange special-purpose instructions and I/O devices that were integrated with the main computing engine. The microprocessor changed that by providing a small, low-cost, CPU engine that could be used as a building block in a larger system. It imposed a rigid hardware architecture based on peripherals connected by a bus and provided a general purpose-programming model, which simplified programming. Off-the-shelf operating systems for embedded systems began to appear in the late 1970s. Many of these were written in assembly language, and could be used only on the microprocessor for which they were written. When microprocessor became obsolete, so did its operating system, unless it was rewritten to run on a newer microprocessor. When the C language came along, operating systems could be written in an efficient, stable and portable manner. This had instant appeal to management, because it held the hope of preserving the software investment when the current microprocessor became obsolete. This sounded like a good story in a marketing pitch. Operating systems written in C became the norm and remain so today. In general, reusability of software has taken hold and is doing rather

nically. A number of commercial operating systems for embedded systems sprang to life in the 1980s. This primordial stew has evolved to the present-day stew of commercial operating systems. Today, there are a few dozen viable commercial operating systems from which to choose. A few big players have emerged, such as VxWorks, pSOS, Neculeus and Windows CE. Many embedded systems do not have any operating system at all, just a control loop. This may be sufficient for very simple ones; however, as systems grow in complexity, an operating system becomes essential or the software grows unreasonably complex. Sadly, there are some horribly complex embedded systems that are complex only because the designers insisted they did not need an operating system. Increasingly, more embedded systems need to be connected to some sort of network, and hence, require a networking stack. For simple embedded systems that are just coded in a loop, adding a network stack may raise the complexity level to the point that an operating system is desirable. Linux as an embedded OS is a new candidate with some attractive advantages. It is portable to many CPUs and hardware platforms, stable, scalable over a wide range of capabilities and easy to use for development.

Definition

As the name signifies, an embedded system is 'embedded' or built into something else, which is a non-computing device, say a car, TV, or toy. Unlike a PC, an embedded computer in a non-computing device will have a very specific function, say control a car, or display Web pages on a TV screen. So, it need not have all the functionality and hence all the components that a PC has. Similarly, the operating system and applications need not perform all the tasks that their counterparts from the PC sphere are expected to. In short, we can define an embedded system as a computing device, built into a device that is not a computer, and meant for doing specific computing tasks.

These computing tasks could range from acquiring or transferring data about the work done by the mother device to displaying information or controlling the mother device. Embedded systems could thus enable us to build intelligent machines. Embedded systems are not a new and exotic topic that is still confined to research theses. There are many live examples of embedded systems around us. MP3 players (computing capability built into a music system), PDAs (computing in what essentially is an organizer), car-control systems, and intelligent toys are but a few examples of such systems already in place.

Features

Most embedded applications have little or nothing in common. Yet there are several embedded system features that are commonly required by systems many different, regardless of their target applications. A comprehensive set of these enhanced features are listed below.

Solid State Disk Support

True embedded systems often require extremely strong data integrity and have rigorous specifications for power consumption, weight, and immunity to adverse environmental conditions. These requirements may prohibit the use of mechanical storage media such as floppy and hard disk drives. Solid state disks provide all of the functionality of magnetic drives, but feature extremely low power consumption, very light weight, and high durability and data reliability.

Power Management

Low power consumption is one of the most common requirements of embedded systems. This is particularly true of portable and other battery-powered applications, which often must run for extended periods of time on a single battery charge. Embedded computer modules provide comprehensive BIOS support for both

automatic and manual power management operations. These features allow dramatic power reductions to be achieved during periods of non-operation. Thermal monitoring and control is also offered on products which require CPU thermal conditioning.

Watchdog Timer

Many mission-critical systems cannot tolerate the down time resulting from crashes due to software problems, power fluctuations, or other abnormal events. The Watchdog timer protects against fatal stoppages by monitoring system operation and resetting in the event of a failure.

Battery-Free Operation

Embedded computer modules employ a serial EEPROM chip to store a backup copy of the CMOS RAM data. This eliminates any vulnerability of configuration data to battery failure, which is a common problem among desktop systems. Battery less operation also allows products to be used in systems which may be exposed to explosive environmental gases.

No-Fail Startup

Embedded computer modules employ various techniques for assuring correct system startup even under adverse conditions. Years of effort have gone into fine-tuning the BIOS to intelligently manage startup errors in case user intervention is not possible.

Instant-On Support

Many embedded systems are required to boot up and begin processing within seconds of system power-up. This is much different than desktop PCs, which commonly take a minute or more to reach a fully operational state. Embedded computer modules provide sophisticated capabilities for dramatically reducing startup delays.

Unattended Operation

Some types of embedded systems are designed to operate unattended for weeks, months, or even years at a time with no user intervention. Such embedded systems should be designed to allow independent operation for extended periods of time.

Embedded Systems versus General Purpose Systems

An embedded system is usually classified as a system that has a set of predefined, specific functions to be performed and in which the resources are constrained. Take for example, a digital wrist watch. It is an embedded system, and it has several readily apparent functions: keeping the time, perhaps several stopwatch functions, and an alarm. It also has several resource constraints. The processor that is operating the watch cannot be very large, or else no one would wear it. The power consumption must be minimal; only a small battery can be contained in that watch and that battery should last almost as long as the watch itself. And finally, it must accurately display the time, consistently, for no one wants a watch that is inaccurate. Each embedded design satisfies its own set of functions and constraints.

This is different from general purpose systems, such as the computer that sits on a desk in an office. The processor running that computer is termed a "general purpose" processor because it was designed to perform many different tasks well, as opposed to an embedded system that has been built to perform a few specific tasks either very well or within very strict parameters.

Real time embedded systems

Embedded systems are often misclassified as real-time systems. However, most systems simply do not require real-time capabilities.

Real time is a relative term. A real-time system (defined by IEEE) is a system whose correctness includes its response time as well as its functional correctness. In other words, in a real-time system, it not only matters that the answers are correct, but it matters when the answers are produced. In other words a real-time computer system can be defined as a system that performs its functions and responds to external, asynchronous events within a specified amount of time. Most control and data acquisition applications, for example, fall into this category. A realtime operating system is an operating system capable of guaranteeing timing requirements of the processes under its control. While time-sharing OS like UNIX strive to provide good average performance, for a real-time OS correct timing is the key feature. Throughput is of secondary concern. There are hard and soft real-time systems depending on time constrains.

Soft real-time systems

Soft real-time systems are those in which timing requirements are statistically defined. An example can be a video conferencing system: it is desirable that frames are not skipped, but it is acceptable if a frame or two is occasionally missed. The soft requirements are much easier to achieve. Meeting them involves a discussion of context switch time, interrupt latency, task prioritization and scheduling. Context switch time was once a hot topic among OS folks. However, most CPUs handle this acceptably well, and CPU speeds have gotten fast enough that this has ceased to be a major concern.

Hard real-time systems

In a hard real-time system, the deadlines must be guaranteed. For example, if during a rocket engine test this engine begins to overheat, the shutdown procedure must be completed in time. Tight real-time requirements should usually be handled by an interrupt routine or other kernel context driver functions in order to assure

consistent behavior. Latency time, the time required to service the interrupt once it has occurred, is largely determined by interrupt priority and other software that may temporarily mask the interrupt. Hard real time means that the system (i.e., the entire system including OS, middleware, application, HW, communications, etc.) must be designed to guarantee that response requirements are met. It doesn't matter how fast the requirements are (microsecond, millisecond, etc.) to be hard realtime, just that they must be met every time. This means that every resource mechanism (i.e., scheduler, I/O manager, mutex mechanism, communications mechanism, etc.) must select the work to be done in the correct order to meet time constraint requirements. This means that mechanisms (e.g., priority inheritance) must be provided to avoid unbounded priority inversion (such as was encountered and corrected in the hard real-time failure in the Mars Pathfinder). This means that FIFO queues must be avoided or kept empty. This means that all processes and threads, including those within the kernel must either be preemptible (i.e., a high priority request can preempt a lower priority one). Hard real-time functions in this tight time range are being implemented in dedicated DSP (digital signal processor) chips or ASICs (application-specific ICs). Also, these requirements are often simply designed out through the use of a deeper hardware FIFO, scatter/gather DMA engines and custom hardware.

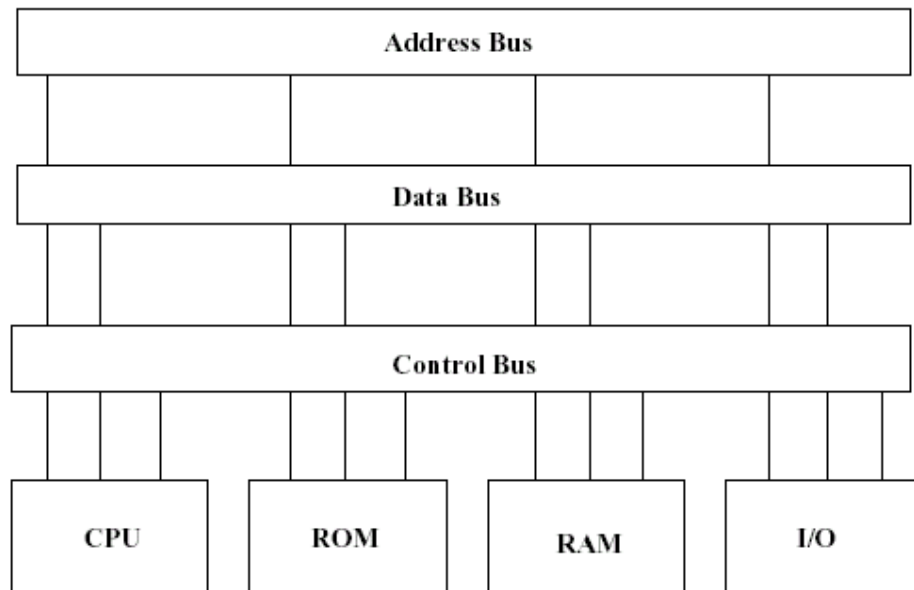
Embedded Hardware

Embedded system consists of hardware (typically VLSI or very large-scale integrated circuits) specifically built for the purpose, an embedded operating system, and the specific application or applications required. The user interface could be push buttons, numeric displays, and LCD panes and so on. One of the problems with

building specific hardware for each type of application is that for every new application you have to literally start from scratch and reinvent the wheel. This increases costs of the system as well as the time taken to develop one. But the benefits of using standard PC components in embedded systems were obvious. So, a method was developed to shrink the PC. What was done was to make the PC bus smaller, and also make the cards stackable, one on top of another, instead of connecting all cards to one big motherboard, so that space is saved. The PC/104 standard, established in 1992 defines the embedded PC. The specification is considered to be an extension of the ISA bus specification. The PC /104 standard have since been extended to PC/104-plus to include the PCI bus. So, today you have PC-based embedded systems that have the ISA bus, the PCI bus, or both. Unlike with regular PCs, in the world of the embedded PC, 386s, 486s and Pentiums are still good enough. Besides these, there are a number of CPUs meant specifically for embedded applications, like the StrongArm, Motorola 68k family and the MIPS.

An embedded system also needs memory for two purposes - to store its program, and to store its data. Unlike normal desktops, in which programs and data are stored at the same place, embedded systems store data and programs in different memories. This is simply because the embedded system doesn't have a hard drive and the program must be stored in memory, even when the power is turned off. This special memory that remembers the program, even without power, is called ROM or Read Only Memory. Embedded applications commonly employ a special type of ROM like Flash Memory that can be programmed or reprogrammed with the help of special devices, unfortunately this kind of memory is not fit for storing data, so embedded systems need some additional regular memory for that

function. Any additional requirement in an embedded system is dependent on the equipment it is controlling. Very often these systems have a standard serial port, a network interface, input/output (I/O) interfaces like push buttons, numeric displays, LCD panes or hardware to interact with sensors and activators on the equipment. So an embedded system has a microprocessor or micro controller for processing of information and execution of programs, memory in the form of Flash memory for storing embedded software programs and data I/O interfaces for external interface. All these devices are hardwired on a printed circuit board (PCB) .The functional diagram of a typical embedded system is shown in figure. The processor uses the address bus to select a specific memory location within the memory



Functional diagram of a typical embedded system

subsystem or a specific peripheral chip. The data bus is used to transfer data between the processor and memory subsystem or peripheral devices. The control bus provides timing signals to synchronise the flow of data between the processor and memory subsystem or

peripheral devices. With embedded PCs you can even go beyond the single-function definition of an embedded system, and could build an entire PC into another machine; a PC inside a refrigerator, or a PC inside a car, for instance.

Embedded Processors

With so many applications, all major microprocessor manufacturers are building their own embedded processors. Many companies have started using existing microprocessor cores and modifying them to suit embedded devices. AMD, for example, recently introduced its AMD-K6-2E processor in two flavors for embedded applications. These have gained support from industry players like Lucent for its WAN/ VPN products line. Motorola has been a significant player in the embedded processors field over the last couple of years. They have the 68K cores at the low-end, ColdFire in the mid-range and PowerPC for higher-end applications. Another contender for the market share is Intel, who went the embedded way with its i960 processor, based on 1.0 micron technology. The same team was then put into developing the StrongArm, which is based on 0.18 micron technology. This processor became quite popular, and found its way into devices like the Compaq iPAQ pocket PC, HP Jornada handheld PC, mobile phones and various digital imaging products.

Embedded Hardware Trends

With the increase in interest and research of embedded systems have come a flood of new design trends. It is hard to envision that five years from now embedded systems will bear much resemblance to the systems today, other than their basic functionalities and even those may be replaced in the future. Two of the trends currently hot in the embedded systems world that are discussed here are that of application specific integrated circuits (ASICs) and systems on a chip (SOC).

Application Specific Integrated Circuits

As the title suggests, this is a IC that has been designed for a specific application. In ASICs, the drawback is that they need heavier investments and longer time spans to develop. Plus, they can't be customized later as the software instructions for them are put on a ROM, which is difficult to modify. Examples of ICs that are ASICs are a chip designed for a toy robot or a chip designed to examine sun spots from a satellite .The reason for mentioning this is that since ASICs are developed for a specific purpose, they are most likely constrained with both a tight budget and a short time to market. Any and all methods that might aid in the development of these chips would be welcomed with open arms in the industry.

System -on- a-Chip

A system-on-a-chip offers all the functions of a computer, but with a difference-all these features including a processor, chipset, video encoder, graphics processor, super I/O, clock generator, and the various buses used to interconnect, except the host memory of the system, are integrated on a single silicon chip .This has the added benefits of size reduction, power reduction, cost reduction, and high performance. One of the more popular forms of SOC is that of Dave Patterson's Intelligent-RAM (IRAM). IRAM is the combination of a processor on a chip with a large area of DRAM. Like all forms of SOC, it reduces the number of chips in a system, allowing the product to be smaller and less expensive. IRAM addresses the key bottlenecks in many systems: memory bandwidth and memory latency. However, the difficulties are: One , there is not so much area on a chip, and it puts upper limits on the amount of main memory that you can have with a system. Another large problem is that the design team creating the system on a chip must contain all of the knowledge to create a processor, a main memory, a I/O controller, and optimize all of them together.

REAL TIME OPERATING SYSTEM AND EMBEDDING LINUX

Why Linux?

Intelligent dedicated systems and appliances used in interface, monitoring, communications, and control applications increasingly demand the services of a sophisticated, state-of-the-art operating system. Many such systems require advanced capabilities like: high resolution and user-friendly graphical user interfaces (GUIs); TCP/IP connectivity; substitution of reliable (and low power) flash memory solid state disk for conventional disk drives; support for 32-bit ultra-high-speed CPUs; the use of large memory arrays; and seemingly infinite capacity storage devices including CD-ROMs and hard disks. This is not the stuff of yesteryear's "standalone" code, "roll-your-own" kernels, or "plain old DOS". No, those days are gone -- forever! Then too, consider the rapidly accelerating pace of hardware and chipset innovation -- accompanied by extremely rapid obsolescence of the older devices. Combine these two, and you can see why it's become an enormous challenge for commercial RTOS vendors to keep up with the constant churning of hardware devices. Supporting the newest devices in a timely manner -- even just to stay clear of the unrelenting steamroller of chipset obsolescence -- takes a large and constant resource commitment. If it's a struggle for the commercial RTOS vendors to keep up, going it alone by writing standalone code or a roll-your-own kernel certainly makes no sense. With the options narrowing, embedded system developers find themselves faced with a dilemma: On the one hand, today's highly sophisticated and empowered intelligent embedded systems -- based on the newest chips and hardware capabilities -- demand nothing less than the power,

sophistication, and currency of support provided by a popular high-end operating system like Windows. On the other hand, embedded systems demand extremely high reliability (for non-stop, unattended operation) plus the ability to customize the OS to match an application's unique requirements. Here's the quandary: general purpose desktop OSes (like Windows) aren't well suited to the unique needs of appliance-like embedded systems. However, commercial RTOSes, while designed to satisfy the reliability and configuration flexibility requirements of embedded applications, are increasingly less desirable due to their lack of standardization and their inability to keep pace with the rapid evolution of technology.

What's a developer to do?

Fortunately, a new and exciting alternative has emerged: open-source Linux. Linux offers powerful and sophisticated system management facilities, a rich cadre of device support, a superb reputation for reliability and robustness, and extensive documentation. Best of all (say system developers), Linux is available at no charge -- and with completely free source code. Is Linux, like Windows, too large and demanding of system resources to fit the constraints of embedded systems? Unlike Windows, Linux is inherently modular and can be easily scaled into compact configurations -- barely larger than DOS -- that can even fit on a single floppy. What's more, since Linux source code is freely available, it's possible to customize the OS according to unique embedded system requirements. It's not surprising, then, that open-source Linux has created a new OS development and support paradigm wherein thousands of developers continually contribute to a constantly evolving Linux code base. In addition, dozens of Linux-oriented software companies have sprung up -- eager to support the needs of developers building a wide range of applications, ranging from factory automation to intelligent appliances.

Which Linux?

Because Linux is openly and freely available in source form, there are many available variations and configurations. So, how do you decide which distribution to use? That depends. First, realize that all Linux distributions are variations on the same theme -- that is, they are pretty much collections of the same basic components, including the Linux kernel, command shells (command processors), and many common utilities. The differences tend to center around which of the many hundreds of Linux utilities have been included, what extras are included, and how the installation process is managed.

Even though Linux is free, purchasing a "commercial" Linux distribution can have many advantages. The many companies who are now building businesses around distributing and supporting Linux are busy investing in developing tools and services to differentiate their Linux offerings from the pack. Some of the special capabilities being developed include:

- Installation tools to automate and simplify the process of generating a Linux configuration that is tuned to a specific target's hardware setup.
- A variety of Windows-like GUIs to support a wide range of embedded requirements.
- Support for the specific needs of various embedded and real-time computing platforms and environments (e.g. special CompactPCI system features).

Embedded Linux systems

Historically, Linux was developed specifically as an operating system for the desktop/server environment. More recently, there has

Linux in Embedded Systems

been a growing interest in tailoring Linux to a very different hardware and software needs of the embedded applications environment. In practice, most embedded systems run from ROM or flash memory, and a more typical footprint would be 386/486/586 or equivalent processor, 8-16Mbytes of 'hard disc' implemented as flash memory, and 16 Mbytes or more RAM. Since Linux provides both a basic kernel for performing the embedded functions and also has all the user interface bells and whistles you could ever want, it is very versatile. It can handle both embedded tasks and user interfaces. Look at Linux as a continuum: scaling from a stripped-down micro-kernel with memory management, task switching and timer services and nothing else, to a full-blown server, supporting a full range of file system and network services. A minimal embedded Linux system needs just these essential elements:

- a boot utility
- the Linux micro-kernel, composed of memory management, process management and
- timing services
- an initialization process

To get it to do something useful and still remain minimal, you need to add:

- drivers for hardware
- one or more application processes to provide the needed functionality

As you add more capabilities, you might also need these:

- a file system (perhaps in ROM or RAM)
- TCP/IP network stack
- a disk for storing semi-transient data and swap capability

Real time operating systems

Real-Time Operating Systems (RTOS) are commonly used in the development, productizing, and deployment of embedded systems. Unlike the world of general purpose computing, real-time systems are usually developed for a limited number of tasks and have different requirements of their operating systems. This section first gives the requirements of realtime operating systems, then how real time performance is achieved in Linux, a general purpose operating system (GPOS).

Real-Time Operating Systems: The Requirements

A good RTOS not only offers efficient mechanisms and services to carry out real-time scheduling and resource management but also keeps its own time and resource consumption predictable and accountable. A RTOS is responsible for offering the following facilities to the user programs that will run on top of it. The first responsibility is that of scheduling: a RTOS needs to offer the user a method to schedule his tasks. The second responsibility is that of timing maintenance: the RTOS needs to be responsible in both providing and maintaining an accurate timing method. The third responsibility is to offer user tasks the ability to perform system calls: the RTOS offers facilities to perform certain tasks that the user would normally have to program himself, but the RTOS has them included in its library, and these system calls have been optimized for the hardware system that the RTOS is running on. The last thing that the RTOS needs to provide is a method of dealing with interrupts: the RTOS needs to offer a mechanism for handling interrupts efficiently, in a timely manner, and with an upper bound on the time it takes to service those interrupts. Several systems adopt POSIX.1b-1993 standard for real-time features in UNIX. The standard defines prioritized scheduling, locking of user

memory pages in memory, real time signals, improved IPC and timers, and a number of other features. Compliance with this standard makes UNIX systems much more appropriate for real-time applications.

Linux and real time

Many (if not all) embedded applications have some sort of real-time performance requirement. Many of these real-time requirements prove to be "soft" - missing a deadline once in a while does not impact the overall system viability. Even when "hard" real-time deadlines do exist, the scope of deterministic response can be reduced to the driver level or overcome by the "real-fast" performance offered by combining Linux with Pentium and PowerPC silicon. So, when developers choose to embed Linux to leverage the wealth of available networking, database, and interface software, real-time concerns often take a back seat. But should they? Using a GPOS, like WindowsNT, for real-time and embedded applications, can spell disaster - recall the U.S. Navy has to tow their Aegis destroyers back to port, repeatedly, because of crashed WindowsNT steering systems. A GPOS typically suffers from several challenges to real-time applicability: determinism in general, and response under load in specific. GPOS schedulers, optimized for time-sharing, can induce unpredictably long blocking times; drivers developed by a mix of GPOS-vendor engineers, peripheral-board vendors, and other third parties add their own variable latencies. Linux, developed for desktops and servers, is also a GPOS, but enjoys a promising future in real time embedded designs. Two primary paths exist to providing a real-time Linux: by inserting a second kernel into the system, and by refining the standard Linux scheduler and tuning Linux device drivers. The first approach, which has also been applied to WindowsNT, presumes that to use Linux for real-time, you must first "throw it out". The addition of a second OS, regardless of its putative real-time characteristics, vastly complicates both the

development and run-time considerations of the embedded Linux developer. A much more sensible approach is to optimize the existing, open Linux code base to address the needs of actual applications, but first to characterize the performance of standard versions of Linux. Before even attempting to enhance Linux responsiveness, it is key to measure its real-time performance, thoroughly, in terms familiar to real-time/embedded designers: worst-case interrupt latency, context switch, and maximum blocking times. Linux already enjoys provably superior compute and networking performance throughput, even when compared to supposedly lightweight RTOS products, implying good to excellent average response times, even under load. Developers, whose requirements exceed such real-time characterizations in terms of the Linux kernel itself, need not despair. Linux partially supports the POSIX.1b standard. As of May 1, 1997, functions for the control of the scheduler and memory locking are fully implemented in Linux, and timers are partially implemented. The problems of kernel non-preemptability, low timer resolution, and high interrupt latency remain unresolved. Thus, POSIX.1b compatibility only permits certain kinds of soft real-time processing in Linux.

Embedding Linux

One of the common perceptions about Linux is that it is too bloated to use for an embedded system. This need not be true. The typical Linux distribution set up for a PC has more features than you need and usually more than the PC user needs also. The standard Linux kernel is always resident in memory. Each application program that is run is loaded from disk to memory where it executes. When the program finishes, the memory it occupies is discarded, that is, the program is unloaded. In an embedded system, there may be no disk. There are two ways to handle removing the dependence on a disk,

depending on the complexity of the system and the hardware design. In a simple system, the kernel and all applications processes are resident in memory, when the system starts up. This is how most traditional embedded systems work and can also be supported by Linux.

Scaling Linux

Traditional embedded operating systems go to great lengths to tout the size and efficiency of their kernels. Realistically, viable commercial OS configurations come in at 128-256 KB for a reasonably configured kernel, another 100-200 KB for a TCP/IP stack and sockets library, and for a web appliance, 50-150 KB for a HTTP server, plus a minimum 64 KB of working RAM. An embedded system software profile of 800 KB to 1 MB no longer looks gargantuan! These same vendors point out that a desktop distribution of Linux runs into the hundreds of megabytes. Well, they are right. When you embed Linux, you choose only those components that make sense for your application. Don't need read/write file system? Don't use it! The same logic applies to networking, GUI, shells, and countless other utilities and libraries. If your project does need more functionality than fits into local non-volatile storage, you can craft either a tiny Linux boot loader, a stand-alone bootable Linux system, or a slim Linux kernel that pulls down additional modules and application code over a network, frequently in 500 KB or less!

Un-Virtual Memory

Another feature of standard Linux is its virtual memory capability. This is that magical feature that enables application programmers to write code with reckless abandon, without regard to how big the program is. This powerful feature is not needed in an embedded system. In fact, you probably do not want it in real-time critical systems, because it introduces uncontrolled timing factors. The software must be more tightly engineered to fit into the available

physical memory, just like other embedded systems. On many CPUs, virtual memory also provides memory management isolation between processes to keep them from overwriting each other's address space. This is not usually available on embedded systems which just support a simple, flat address space. Linux offers this as a bonus feature to aid in development. It reduces the probability of a wild write crashing the system. Many embedded systems intentionally use "global" data, shared between processes for efficiency reasons. This is also supported in Linux via the shared memory feature, which exposes only the parts of memory intended to be shared.

The file systems -components

Many embedded systems do not have a disk or a file system. Linux does not need either one to run. As mentioned before, the application tasks can be compiled along with the kernel and loaded as one image at boot time. This is sufficient for simple systems. In fact, if you look at many commercial embedded systems, you'll see that they offer file systems as options. Most are either a proprietary file system or an MS-DOS-compatible file system. Linux offers an MS-DOS-compatible file system, as well as a number of other choices. The other choices are usually recommended, because they are more robust and fault-tolerant. Linux also has check and repair utilities, generally missing in offerings from commercial vendors. This is especially important for flash systems which are updated over a network. If the system loses power in the middle of an upgrade, it can become unusable. A repair utility can usually fix such problems. The file systems can be located on a traditional disk drive, on flash memory, or any other media. Also, a small RAM disk is usually desirable for holding transient files. Flash memories are segmented into blocks. These may include a boot block containing the first software that runs when the CPU powers up. This could include the Linux boot code. The

Linux in Embedded Systems

rest of the flash can be used as a file system. The Linux kernel can be copied from flash to RAM by the boot code, or alternatively, the kernel can be stored in a separate section of the flash and executed directly from there. Finally, for networked embedded systems, Linux supports NFS (Network File System). This opens the door for implementing many of the value-added features in networked systems. First, it permits loading the application programs over a network. This is the ultimate in controlling software revisions, since the software for each embedded system can be loaded from a common server. This can be a very powerful feature for user monitoring and control. For example, the embedded system can set up a small RAM disk, containing files which it keeps updated with current status information. Other systems can simply mount this RAM disk as a remote disk over the network and access status files on the fly. This allows a web server on another machine to access the status information via simple CGI scripts. Other application packages running on other computers can easily access the data. For more complex monitoring, an application package such as MatLab can easily be used to provide graphical displays of system operation at an operator's PC or workstation. Linux applications are usually linked to shared libraries. These, just as their name implies, are libraries of functions that are shared between applications and utilities. The most frequently required shared library is C runtime library (GLIBC)-around 4 Mbytes. Luckily, libraries can often be reduced in size by removing all debugging information. However, if this is too large for your embedded system, it is possible to turn to less standardized embedded libraries, for example uClibc and newlib. Again, depending upon application requirements for a build of embedded Linux, additional file system components will be needed. For Java support, we may need to add JVM such as Kaffe or IBM's J9.

The file systems -high availability

An embedded Linux file system, unlike desktop or server implementations, must offer user independent support for recovery in the event of power failure. Also, power consumption, size and failure rate considerations mean that the file system is likely to be running from some variant of flash or ROM, rather than hard disc or other rotating media. There are a number of alternatives when using flash-based storage media. Some flash devices connect to the standard hard drive connector (IDE port) and emulate a hard disc drive. These are extremely easy to use-requiring no changes to the kernel-but are generally susceptible to corruption. An alternative is to use a 'smart' flash device such as DiscOnChip from M-systems. The Linux kernel must be rebuilt to support this device, but, again depending on the file system used, corruption can occur. Another is to make use of on-board flash memory. As with DiscOnChip devices, drivers must be added to Linux kernel -but there is the option of using the Journaling Flash File System (JFFS), currently under devolvment within open-source community. The 2.4 series of Linux kernels currently include support for the JFFS. This is a 'log structured 'file system, which means that old data is not lost when new data is written to flash. Periodically, this old information is garbage collected from the flash, but write interruption will not cause the file system to become corrupt. Work is also underway to implement a compressed JFFS for embedded devices.

Booting

When a microprocessor first powers up, it begins executing instructions at a predetermined address. Usually there is some sort of read-only memory at that location, which contains the initial start-up or boot code. In a PC, this is the BIOS. It performs some low-level CPU initialization and configures other hardware. The BIOS goes on to figure out which disk contains the operating system, copies the OS to

RAM and jumps to it. Linux systems running on a PC depend on the PC's BIOS to provide these configuration and OS-loading functions. In an embedded system, there often is no such BIOS. Thus, you need to provide the equivalent startup code. Fortunately, an embedded system does not need the flexibility of a PC BIOS boot program, since it usually needs to deal with only one hardware configuration. The code is simpler and tends to be fairly boring. It is just a list of instructions that jam fixed numbers into hardware registers. However, this is critical code, because these values need to be correct for your hardware and often must be done in a specific order. There is also, in most cases, a minimal power-on self-test module that sanity-checks the memory, blinks some LED's, and may exercise some other hardware necessary to get the main Linux OS up and running. This startup code is highly hardware-specific and not portable.

Advantages/disadvantages of using embedded Linux

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. 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

Linux in Embedded Systems

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. The core Linux operating system, the kernel itself has a fairly simple micro-kernel or monolithic architecture, meaning the whole operating system-process management, memory management, file system and drivers-is contained within one binary image which is in compressed form. 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.

Comparison with existing embedded operating systems

An off-the-shelf RTOS like QNX, PSOS, and VxWorks are 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. Also they have been designed from the ground up to conform to the constraints of inherent in an embedded environment. Similarly, the demands for realtime performance were addressed during the initial design phase. As a result, commercial non-Linux embedded operating systems have tended to be more scalable at the low end and have better real-time performance. However embedded Linux has now evolved to the point where it can address, at low or zero cost, all but the most demanding of embedded applications .The real time performance issue

can be important in the market, and embedded Linux vendors are working hard to match the real time capabilities of established products. Its future looks bright-the penguin has come of the age.

Embedded software

C has become the language of choice for embedded programmers, because it has the benefit of processor independence, which allows programmers to concentrate on algorithms and applications, rather than on the detailed of processor architecture. However, many of its advantages apply equally to other high-level languages as well. Perhaps the greatest strength of C is that it gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. Compilers and cross compilers are also available for almost every processor with C. Any source code written in C or C++ or Assembly must be converted into an executable image that can be loaded onto a ROM chip. The process of converting the source code representation of your embedded software into an executable image involves three distinct steps, and the system or computer on which these processes are executed is called a host computer.

First, each of the source files that make an embedded application must be compiled or assembled into distinct object files, Second, all of the object files that result from the first step must be linked into a final object file called the re locatable program. Finally, the physical memory address must be assigned to the re locatable program. The result of the third step is a file that contains an executable image that is ported on the ROM chip. This ROM chip, along with the processor and other devices and interfaces, makes an embedded system run. There are some very basic differences between conventional programming and embedded programming.

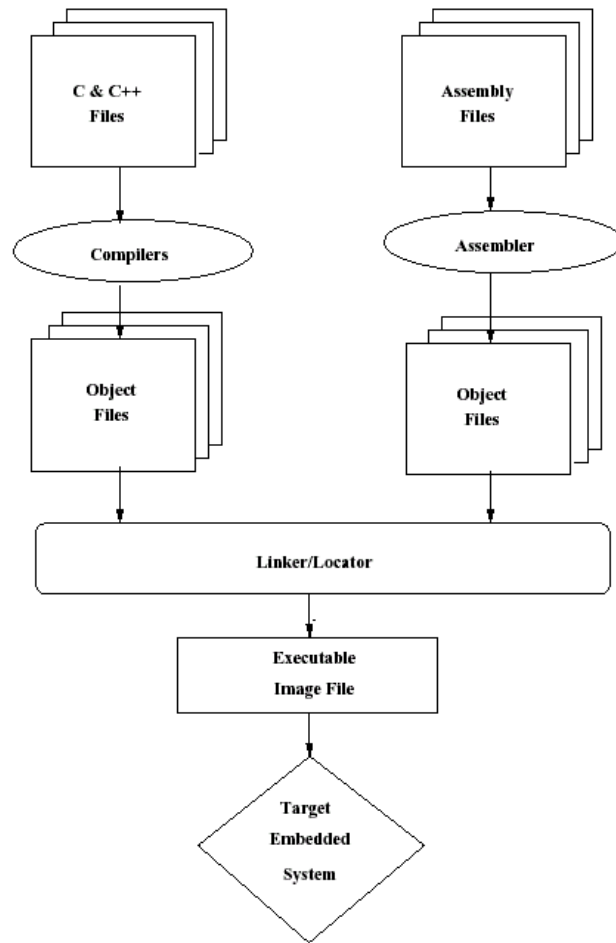


Figure 4.1: Basic functional architecture of an software in embedded system

First each target platform is unique. Even if the processor architecture is the same, I / O interfaces or sensors or activators may differ. Second, there is a difference in the development and debugging of applications.

APPLICATIONS

Embedded software is present in almost every electronic device you use today. However, a common obstacle for developers has been the need to develop different sets of hardware and software, for different devices. An 'intelligent' washing machine uses a hardware chip different from that used by an 'intelligent' wristwatch. In addition, the software running on the hardware chip is different. This often results in increased costs and time taken for development. Defense services use embedded software to guide missiles and detect enemy aircrafts. Communication satellites, medical instruments, and deep-space probes would have been nearly impossible without these systems. Embedded systems cover such a broad range of products that generalization is difficult. Here are some broad categories.

- Aerospace and Defense electronics (ADE)
- Automotive
- Broadcast and entertainment
- Consumer/internet appliances
- Data communications
- Digital imaging
- Industrial measurement and control
- Medical Electronics
- Server I/O
- Telecommunications

Linux and PDA's

In no area is this improvement more significant than for PDAs, a market for which Linux currently seems stunningly appropriate. Here is a market in which connectivity with desktop computers is important, so mature and stable networking support is a necessity. The presence of a GUI virtually defines what a PDA is, so good GUI support is also important. Current retail prices for PDAs are in the low hundreds of US dollars, a price range that allows designing in RAM, flash, and processing power sufficient to support Linux well. On the other hand, the market is sufficiently price sensitive that most manufacturers find it onerous to pay even a token royalty to Microsoft or Palm for the use of an OS. Yet using a low-end or home-brew OS, in an attempt to target cheaper hardware, is simply out of the question –the market is moving too fast for the longer design cycles that would result. In recent months there have been numerous announcements of new Embedded Linux support for PDAs and other handheld personal computing devices. Additionally, a growing number of PDAs (and similar devices) are known to be in development that will offer Linux as their primary embedded operating system. They include Sharp Zaurus SL-5000 ,G.Mate Yopy ,Empower PowerPlay III ,HNT Exilien ,SK Telecom IMT2000Web Phone etc. In short, all signs now point to embedded Linux, as a platform for today's increasingly sophisticated PDAs.

CONCLUSION

The developments of embedded systems have been fairly dynamic over the past couple of years with the rapid digitization of various parts of our day-to-day utility items. For example, in the industrial segment the biggest turnaround came with the aircraft industry adopting the fly-by-wire as a standard. This promulgated the total automation of airplanes, which percolated further with the digitization of cars, down to simple ignition systems that are used today. The rapid development of industries has led to plant automation across the world. Today robots do most of the welding and placing in large factories. These robots are again powered by microprocessors that tell them to fire the right sparks at the right place and the right time. The possibilities in this field are only limited by our imagination. Embedded developers are a flexible, forward-looking bunch, and despite the need to reorient themselves technically, they are flocking to Linux like penguins to their nesting ground. They are choosing Linux for the technical advantages cited above, for its greater reliability, for the comprehensive set of standard APIs, and to lower their cost of goods sold, and bring their products to market faster.

FUTURE DIRECTIONS

The trend of embedded systems now involves the miniaturization of electronics so that it can fit into compact devices. In the future these systems will be moved by the forces of nature. Soon we will see more digitization of appliances, and these will be fueled by human need. Today when we lose our way, we have no better option but to ask people for directions. However, we will soon be assisted by GPS and pathfinders that will power cars and people alike. Looking into future, we can see industries with automated time clocks, which will compute with the back-end supply chain to decide on raw materials and stocking. For several years, Linux advocates have predicted that Linux will become a significant factor in the embedded market. In addition to its virtues as a full-featured modern operating system, it is inexpensive to duplicate, an especially important factor for embedded systems. Time will tell, but it certainly looks as though Linux has already altered the embedded and real-time operating system landscape in a fundamental and irreversible way. Developers now have greater control over their embedded OS; manufacturers are spared the costs and headaches of software royalties; end users get more value. And the penguins of the South Pole are celebrating.

References

1. Alex Lennon, "Embedding Linux", *IEE Review*, pp. 33-37, May 2001.
2. Wayne Wolf, "Computers as Components-Principles of Embedded Computing System Design"
3. Sumeet Kumar, "Delving Deeper into Embedded Systems", *Information Technology Magazine*, pp.13-16, October 2001.
4. Krishna Kumar, "Computers in your Toys, cars, appliance...", *PC Quest Magazine*, Pp.3-8, May 2001.
5. Joel R. Williams, "Embedding Linux in a Commercial Product",
6. <http://embedded.linuxjournal.com/articles/3587.php>.
7. Rick Lehrbaum, "Using Linux in Embedded and Real-Time Systems",
8. <http://embedded.linuxjournal.com/articles/3980.php>.
9. S. Markon and K. Sasaki, "Linux for Embedded Systems ",
10. <http://embedded.linuxjournal.com/articles/0133.php>.
11. Kevin Dankwardt, "Comparing real-time Linux alternatives",
12. <http://www.linuxdevices.com/articles/AT4503827066.html>.
13. Rick Lehrbaum, "The Linux-PDA and PDA-Linux Quick Reference Guide", <http://www.linuxdevices.com/articles/AT8728350077.html>.
14. Sebastien Huet, "Embedded Linux HOWTO", <http://linuxembedded.com/howto/Embedded-Linux-Howto.html>.
15. <http://www.linuxdevices.com>.
16. Stephen Balacco, "Linux's Future in the Embedded Systems Market", <http://www.linuxdevices.com>

CONTENTS

1. Introduction
2. Embedded Systems
 - 2.1 History
 - 2.2 Definition
 - 2.3 Features
 - 2.4 Embedded versus General purpose systems
 - 2.5 Real Time Embedded Systems
 - 2.6 Embedded Hardware
3. Real Time Operating System and embedding Linux
 - 3.1 Why Linux?
 - 3.2 What is developer to do?
 - 3.3 Which Linux?
 - 3.4 Embedded Linux Systems
 - 3.5 Real Time Operating Systems
 - 3.5.1 Real Time Operating Systems: the requirements
 - 3.5.2 Linux and Real Time
 - 3.6 Embedding Linux
 - 3.6.1 Scaling Linux
 - 3.6.2 Un-virtual Memory
 - 3.6.3 The file systems- components
 - 3.6.4 The file systems-high availability
 - 3.6.5 Booting
 - 3.7 Advantages and Disadvantages of Using Embedded Linux
 - 3.8 Comparison with existing Embedded Operating Systems
 - 3.9 Embedded Software
4. Applications
 - 4.1 Linux and PDAs
 - 4.2 Linux's Future in the Embedded System Market
5. Conclusion
6. Future Trends
7. References

ACKNOWLEDGEMENT

I thank God Almighty for the successful completion of my seminar. Sincere feelings of gratitude for Dr.Agnisharman Namboothiri, Head of the Department, Information Technology.

I express my heartfelt gratitude to Staff-in-charge, Miss. Sangeetha Jose and Mr. Biju, for their valuable advice and guidance. I would also like to express my gratitude to all other members of the faculty of Information Technology department for their cooperation.

I also express my gratitude to my parents for the support they have given me so far.

I would like to thank my dear friends, for their kind-hearted cooperation and encouragement.