

# Linux, Nucleus... or Both

By Colin Walls, Mentor Graphics

Until recently, operating system (OS) specification for embedded systems has been seen largely as an ‘either/or’ exercise. Similarly, OSs that have their foundations in the embedded market and those that have grown out of desktop computers have been seen as competing rather than complementary technologies.

Cost and performance criteria within specifications will often lead to one technology winning out over another. But as hardware moves increasingly to multicore architectures, it is also the case that different types of OS can be specified within a single overall end-product, each specifically handling those tasks it carries out most efficiently.

This article compares one real-time OS (Nucleus from Mentor Graphics) of the type traditionally associated with embedded systems with a general-purpose OS (Linux) that is increasingly being used in that market to identify their various advantages, and also the emerging opportunities for their use alongside one another.

Recent years have seen a lot of publicity and enthusiasm around implementations of Linux on embedded systems. It seems that another mobile device manufacturer announces its support for the Linux general purpose OS (GPOS) every week. At first, many developers viewed Linux as an outright competitor to established and conventional real-time OSs (RTOSs). Over time, though, as the various options have been tried out in the real world, a new reality has dawned. Each OS has its own strengths and weaknesses. No one OS fits all.

To illustrate this new reality, this article takes a closer look at the differences between a commercial RTOS (Nucleus) and a GPOS (Linux)—and considers how the two OSs might even work together.

Desktop Computer	Embedded System
Runs different programs at different times depending upon the needs of the user.	Runs a single, dedicated application at all times.
Has large amounts of (RAM) memory and disk space, both can be readily and cheaply expanded if required.	Has sufficient memory, but no excess; adding more is difficult or impossible.
All PCs have an essentially identical hardware architecture and run identical system software. Software is written for speed.	Embedded systems are highly variable, with different CPUs, peripherals, operating systems, and design priorities.
Boot up time may be measured in minutes as the OS is loaded from disk and initialized.	Boot up time is almost instantaneous—measured in seconds.

Figure 1: Differences between a desktop computer and an embedded system

## Embedded vs. Desktop

The phrase ‘operating system’ causes most people to think in terms of the controlling software on a desktop computer (e.g., Windows), so we need to differentiate between the programming environments for a PC and those for an embedded system. Figure 1 highlights the four key differences.

It is interesting to note that as new OSs are announced (such as the Chrome OS for netbooks), the differences between a desktop computer and an embedded system become even less obvious. Further, more complex embedded systems now have large amounts of memory and more powerful CPUs; some often include a sophisticated graphical user interface and require the dynamic loading of applications. While all these resources appear to facilitate the provision of ever more sophisticated applications, they are at odds with demands for lower power consumption and cost control.

Obviously OS selection is not a simple matter. It is driven by a complex mix of requirements, which are intrinsic in the design of a sophisticated device. The ideal choice is always a combination of technical and commercial factors.

**Technical Factors for OS Selection** From a technical perspective, the selection criteria for an OS revolve around three areas: memory usage, performance, and facilities offered by the OS.

**Memory** All embedded systems face some kind of memory limitation. Because memory is more affordable nowadays, this constraint may not be too much of a problem. However, there are some situations where keeping the memory usage to a minimum is desired; this is particularly true with handheld devices. There are two motivations for this: cost—every cent counts in most designs; and power consumption—battery life is critical, and more memory consumes more power.

For memory-constrained applications, it is clearly desirable to minimize the OS memory footprint. The key to doing this is scalability. All modern RTOS products are scalable, which means that only the OS facilities (i.e., the application program interface (API) call service code) used by the application are included in the memory image. For certain OSs, this only applies to the kernel. Of course, it is highly desirable for scalability to apply to all the OS components such as networking, file system, the user interface and so on.

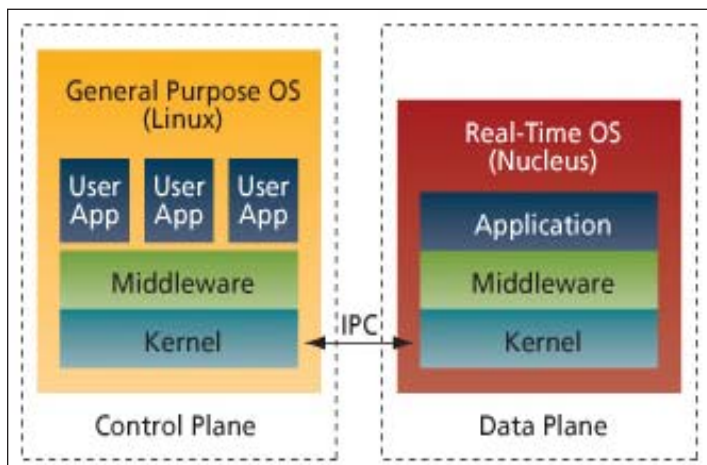


Figure 2: High-level software architecture with the separation between the control and data planes in a multi-OS/AMP system

**Performance** The performance of a device on an embedded application is a function of three factors: application code efficiency, CPU power and OS efficiency. Given that the application code performance is fixed, a more efficient OS may enable a lower power CPU to be used (which reduces power consumption and cost), or may allow the clock speed of the existing CPU to be reduced (which can drastically reduce power consumption). This is a critical metric if the last ounce of performance must be extracted from a given hardware design.

**Facilities** Ultimately, an OS is a set of facilities required in order to implement an application. So, the availability of those facilities is an obvious factor in the ultimate selection. The API is important, as it facilitates easy porting of code and leverages available expertise. Most RTOSs have a proprietary API, but POSIX is generally available as an option. POSIX is standard with Linux.

Most embedded OSs employ a thread model; they do not make use of a memory management unit (MMU). This facilitates maximum OS performance, as the task context switch can be fast, but it does not provide any inter-task protection. Higher-end OSs, like Linux, use the process model where the MMU is employed to completely insulate each task from all of the other tasks. This is achieved by providing a private memory space at the expense of context switch speed. An interesting compromise is for a thread-based OS to utilize a MMU to protect other tasks' memory, without re-mapping address spaces. This provides significant inter-task protection, without so much context switch time overhead.

Of course, there is much more to an OS than the kernel and the breadth and quality of the middleware. The availability of specific application code, preconfigured for the chosen OS, may be very attractive. Even if the current target application does not need all of these capabilities, possible upgrades must be anticipated.

**Commercial Factors for OS Selection** The primary consideration in any business decision is cost. Even though selecting an OS is apparently a technical endeavor, financial issues can strongly influence the choice. There are initial costs, which include the licensing of software or procurement of tools, and there are ongoing costs, such as runtime licenses or maintenance charges.

All software has some kind of license, so the costs of legal scrutiny must be factored in, as lawyers with appropriate technical skills do not come cheap. There is also the question of ongoing support and who is going to provide it.

Finally, for most state-of-the-art devices, time-to-market is extremely tight, so the extent to which the choice of OS can accelerate development may be measured not as a cost, but rather, as a cost saving measure.

### Linux or Nucleus? One Company's Experience

BitRouter, a successful software company from San Diego, California, builds turnkey software solutions for set-top box and television applications.

The company has implemented solutions using the Mentor Graphics Nucleus RTOS, uC/OS-II, VxWorks, OS20, WIN32, commercial Linux, as well as embedded Debian Linux for ARM. Some of BitRouter's main customers include Texas Instruments, Toshiba Semiconductors, NXP Semiconductors, STMicroelectronics, Motorola, RCA and NEC.

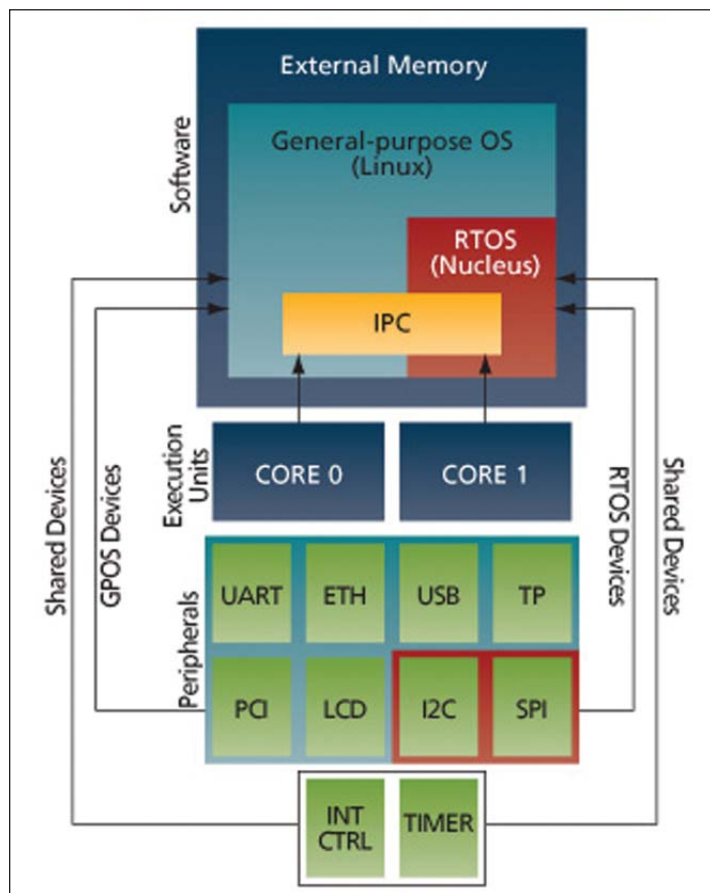


Figure 3: Partitioning of system resources in a multicore, multi-OS design

BitRouter had the opportunity to implement similar digital-to-analog converter set-top boxes using Linux and Nucleus. The Nucleus-based set-top box had a Flash and RAM footprint of roughly half that required by the similar Linux-based set-top box. The boot time required for video to play was three seconds with Nucleus compared to ten seconds with Linux.

This is just one example of how an application dictates the most suitable OS. In this situation, a commercial RTOS was better suited because it was small, compact, and it was being built into a high-volume system where memory footprint and boot-up time were key issues.

BitRouter is nevertheless still a big supporter of Linux and believes Linux will be a good fit for Java-based set-top boxes and TV sets where the total RAM footprint can exceed 64MB—and where constrained space is not such a critical issue.

## The next frontier: Multicore, Multi-OS Designs

The OS selection between Nucleus and Linux may not even comprise the right question. Perhaps it is better to ask how these two OSs can work together, maximizing their respective strengths to address ever-challenging performance and power efficiency requirements for today's increasingly complex embedded applications.

One solution is moving to multicore system development. And while multicore has been around for some time, what is new are the recent advancements in asymmetric multi-processing (AMP) systems. An AMP multicore system allows load partitioning with multiple instantiation of the same or different operating systems running on the same cores.

Figure 2 shows a basic, high-level software architecture for an AMP system. The design includes both a GPOS and an RTOS—each serving distinctly different purposes. The system is partitioned in such a way that the GPOS handles the control plane activities (e.g., drivers, middleware, GUI) and the RTOS handles data plane activities that are time-sensitive, deterministic, and computationally more intensive.

A key ingredient in the success of the multi-OS/AMP system is the Inter-Process Communication (IPC) method, which until recently, varied from one design to the next. IPC allows the OSs to communicate back and forth. Today, there are a number of open standards for IPC, which will further expedite multicore, multi-OS development.

Figure 3 takes the multi-OS example one step further. It shows a few of the design decisions behind the integration of the GPOS and RTOS, and a real-world example can be envisioned in terms of what fabless chip vendor Marvell recently accomplished with its Sheeva line of embedded processors.

The company is a specialist in storage, communications and consumer silicon, with a focus on low-power, high-performance devices. Sheeva, allows developers to use dual OSs to manage separate function requirements. For example, in one application for enterprise printing, Nucleus could be used for inter-operational tasks where speed is of prime importance, while Linux could be used for networking and the user interface.

**Conclusion** Conventional embedded RTOSs and desktop-derived OSs each have a place in the embedded ecosystem. An RTOS makes less demand on resources and is a good choice if memory is limited, real-time response is essential, and power consumption must be minimized. Linux makes good sense when the system is less constrained and a full spectrum of middleware components can be leveraged.

Finally, there are an increasing number of instances where multicore design can benefit from a multi-OS approach—on a single embedded application—maximizing the best of what Nucleus and Linux have to offer.

END

# ARIUM

## The First and Last Word in Emulators for ARM Core-based Processors!

All Arium Emulators support the entire family of ARM Core-based Processors: switch from ARM7 to ARM9 or OMAP, and keep the Emulator and SourcePoint™ Debugger--it's that easy!



### *All Arium Emulators*

- Handle Code Debug from Board Reset
- Deliver Full symbolic Source-level Debug of Linux Kernel Code and Applications
- Provide Seamless Transitions to and from the Linux Kernel and each Process

*Order Yours Today!*

[www.arium.com](http://www.arium.com)

 **arium**

ARM7  
ARM9/9E  
ARM11  
OMAP  
XScale  
Cortex-M1  
Cortex-M3  
Cortex-R4  
Cortex-A8  
ARM7  
ARM9/9E  
ARM11  
OMAP  
XScale  
Cortex-M1  
Cortex-M3  
Cortex-R4  
Cortex-A8  
ARM7  
ARM9/9E  
ARM11  
OMAP  
XScale  
Cortex-M1  
Cortex-M3  
Cortex-R4  
Cortex-A8