

Achieve the Impossible in IoT: Full-Featured Apps on Resource-Limited Platforms

[Electronic Design](#)

[Andrew Caples](#)

Fri, 2015-08-14 10:13



The expectations of the Internet of Things (IoT) are driving together two conflicting goals—building advanced operational features and minimizing cost and size. Today’s embedded-system developers are basically tasked with designing “TARDIS-like” products that appear bigger on the inside than on the outside. Building these types of systems requires a bit of magic, and fortunately, this magic exists today. By using frameworks for memory and power management, as well as scalability features available on a full-featured, real-time operating system (RTOS), it’s possible to have a system behave as if it has more resources available than what’s actually present.

MCU Proliferation

As expectations for the IoT turn into reality, we’re seeing billions of connected products integrated with the cloud. The growth in software required to handle this connectivity and IoT integration will certainly explode over the coming years. Much of the IoT’s growth will be driven by low-cost microcontrollers (MCUs). Today’s MCUs have plenty of compute power and come with low-cost wireless connectivity that serves as an enabling technology for cloud integration, such as Bluetooth and, eventually, Wi-Fi.

With billions of connected products, IoT end nodes will have to be smart. They must be able to act independently, as well as act in concert with each other. These connected products should be able to make their own decisions or respond to decisions based on a higher-level interaction. In short, IoT-connected products will require complex M2M middleware for communication, zero configuration networking, and advanced protocols for cloud connectivity. Because many of these end products will be resource-limited to minimize cost, size, space, power consumption, heat dissipation, etc., the task of creating advanced embedded systems to meet IoT requirements becomes extremely challenging.

Finding an Operating System

For many IoT end nodes, the complexity of the embedded system itself will rule out bare-metal software designs. If anything, it’s more than likely additional software will be needed in the form of an operating system (OS) to manage the advanced middleware and IoT protocols.

Related

[Q&A: Mentor’s Kurisu Discusses Embedded-Systems Solution for Industrial Automation](#)

[SELinux 101: What You Should Know](#)

[IoT Is Here Whether You Like It or Not](#)

U-based development has historically been driven largely by reducing footprint to maximize the use of limited, system-on-chip (SoC) resources. Many MCU-based designs do not include external memory resources, and limit the runtime software to internal memory that can be measured in tens of kilobytes. As form factors shrink and complexity grows, the task to squeeze the operating system, middleware, and application becomes even more daunting.

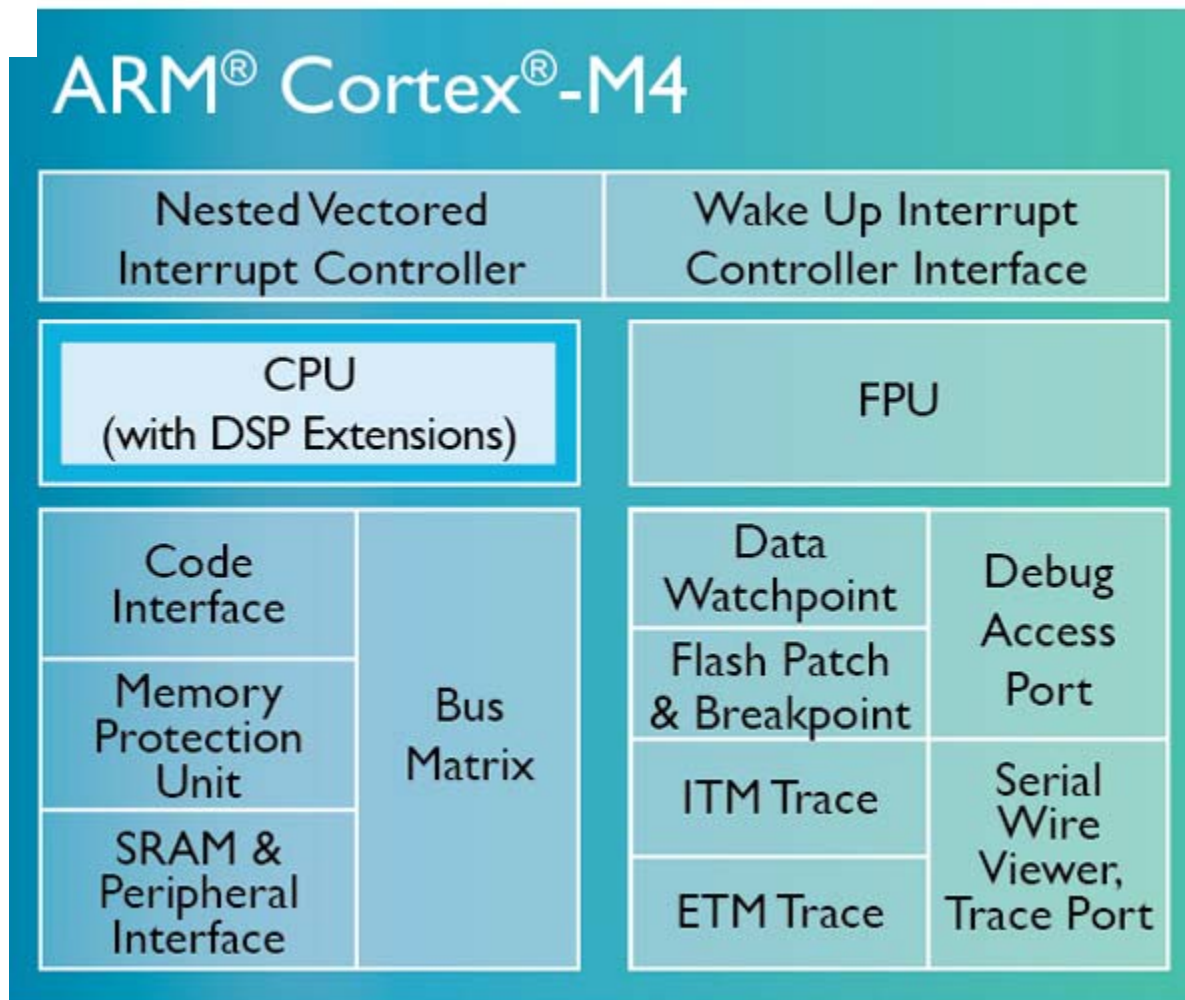
For embedded developers, footprint reduction starts with the underlying framework: the operating system. Although a vast array of embedded operating systems is available, selecting a scalable OS can be critical to success. The scalability of a commercial RTOS allows developers to include only the features required to minimize footprint. A commercial RTOS that's scalable, such as the [Mentor Graphics](#) Nucleus RTOS, can easily be reduced to approximately 10 kB with a full-featured kernel. For designs with very limited system resources, additional size reduction is possible.

For instance, using CSGNU tools in Thumb2 Instruction Mode, the Nucleus kernel can be configured for approximately 6 kB with OS objects that include: Task Control, Interrupt Control, Events and Semaphores, and Timer. To meet the most stringent footprint requirements, the RTOS can be configured to include only the OS objects needed for a footprint less than 3 kB.

Though configuring the software framework for minimal size has historically proved effective to meet system requirements, there's a limit. And the growing need for advanced operational features in IoT devices will drive the demand for more, rather than less, kernel features. Maximizing resource utilization thus requires a new approach.

The Need for a “Scalable” RTOS

With the convergence of powerful MCUs and advanced operational features in IoT, reducing the software footprint becomes only part of an overall approach to meet requirements in connected IoT products with limited system resources. Most commercial RTOSs available today lack the scalability needed to take advantage of the actual features in the underlying silicon itself.

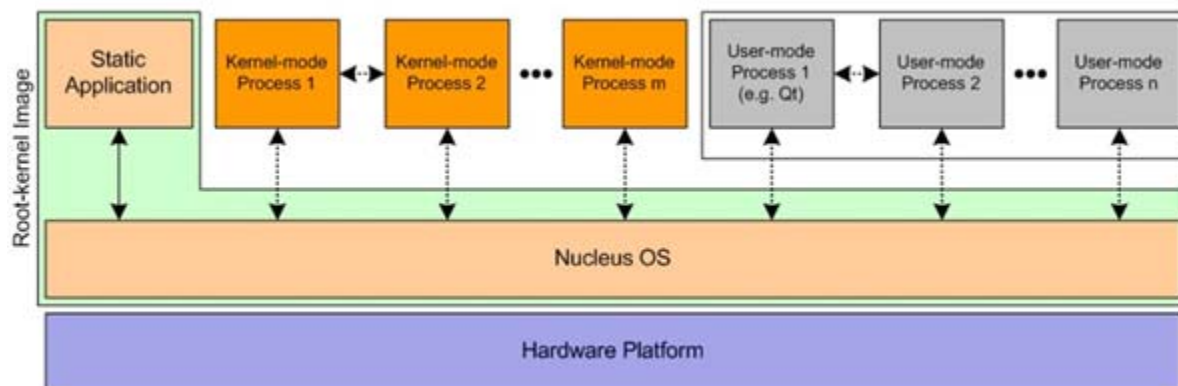


As an example, many IoT SoCs are based on ARM Cortex-M3 and M4 cores (*Fig. 1*), which offer a Memory Protection Unit (MPU). These MPUs provide access control to memory regions to create spatial domain partitioning capable of isolating software subsections from each other. The MPU can be used to enforce privilege and access rules, which are needed to separate processes in their respective protected memory domains. Not only does this create added system reliability (contain faults to a single process without degradation to the entire system), it makes possible advanced capabilities such as dynamical linking and loading.

Support for spatial partitioning has been a mainstay feature in high-end, general-purpose operating systems such as Linux. However, the overhead associated with memory virtualization and lack of deterministic responses limit the use of embedded Linux in real-time systems. And, of course, the large Linux footprint all but rules it out as a possible OS in MCU-based designs.

Maximize Resource Utilization with Process Models

Although there are plenty of commercial RTOSs to choose from, a majority don't seem to focus on abstracting the MPU, which would provide a mechanism that allowed embedded developers to design software for spatial partitioning. With a framework built into the operating system, the MPU can be easily configured at runtime to establish memory regions in both kernel and user space. Intuitive application programming interfaces (APIs) can be used to load processes at runtime or based on the use-case during execution. For example, Nucleus employs a lightweight approach to a process model (*Fig. 2*). The MPU in Cortex-M3/M4-based SoCs can handle spatial domain partitioning without the need, or overhead, to virtualize memory.



Processes are loadable directly from ROM or flash into memory. In addition, with pre-linked embedding, processes can execute in situ in flash, which is a feature commonly required in MCUs with very limited RAM. Utilizing the MPU in Cortex-M-based SoCs gives embedded developers a powerful feature to design TARDIS-like products (*Fig. 3*). In other words, the interior of the system behaves as if it's much larger than its exterior. Limited memory resources in IoT devices can be effectively allocated and employed based on the use case.

Furthermore, processes can be dynamically linked and loaded during runtime in either kernel or user space. Kernel-mode processes run at a higher privilege level, which is the same privilege as the root-kernel image for access to system resources. User-space processes run at a reduced privilege level, with a full-featured RTOS enforcing memory protection to ensure that user-mode processes aren't able to impact the system. Application code and middleware can be loaded into individual processes.



On top of that, processes can export symbols (functions) for use by other processes or application code, which adds functionality to a running system. Once processes are loaded, they can execute within their protected regions for as long as required. APIs can be called to gracefully stop and unload the application, freeing system resources. Applications can be loaded based on the use case, and memory can be released once the use case changes to free up system resources.

What About Power Management?

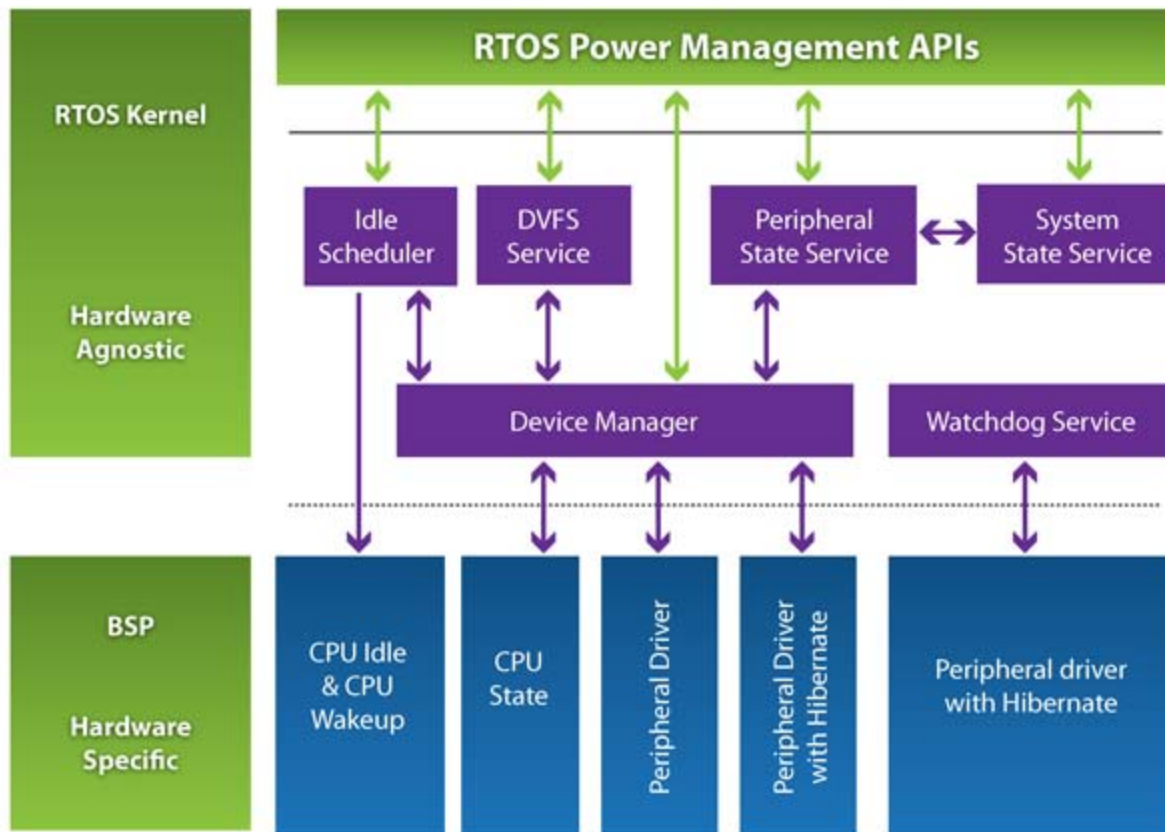
A framework may also play a key role in terms of power management. The importance of power management and power optimization will steadily grow as an increasing number of battery-powered products connect to the cloud. Contemporary MCUs offer low-power features such as idle modes, sleep modes, dynamic voltage frequency scaling (DVFS), and hibernate.

Still, the complexity to implement the low-power features in the hardware typically prevents application developers from generating power-efficient code. If the underlying operating system doesn't have a framework to exploit the low-power features in the silicon, the amount of code required not only

creates a new level of complexity, it also adds to code bloat.

Working Within a Power-Management Framework

Just as modern software architectures abstract hardware functions through device drivers, a power-management framework provides a structured mechanism for all system devices to be controlled using intuitive APIs calls (Fig. 4). Any alteration of one device that impacts other devices results in a coordinated transition across all involved subsystems. The power-management framework approaches the conservation of power usage from four directions: system states control peripheral power; DVFS focuses on the entire system; idle power management prevents expending energy without an ascertainable goal; and hibernate/sleep modes allow the system to go offline during long periods of inactivity.



With a power-management framework, embedded-software developers can effectively write code to meet power requirements without creating code bloat or increasing footprint. No doubt, IoT will drive the growth of connected products and battery-powered wearables. Most of these devices will be based on low-cost MCUs with limited system resources. A power-management framework allows embedded-software developers to consider power specifications early in the software design cycle. Code can be written to minimize both the footprint and power consumption—and tested throughout the development process to ensure power requirements are achieved.

Conclusion

With the rollout of IoT and its many connected products and services, software developers are challenged to add advanced operational features in embedded systems with limited system resources. Scalable operating systems will be required to minimize footprint. However, this alone will not meet the requirements for resource-limited IoT end-nodes and the like. Rather, it requires a full-featured real-time operating system that's both scalable and includes a power-management framework to leverage the features in the silicon.

question is often raised among fellow embedded developers, “...but how do I add advanced features while shrinking the footprint to minimize cost, size, and power consumption?” When you use a scalable RTOS with a built-in framework to leverage the features in the silicon, it is possible to have a system behave as if it has more resources than are actually present. It is possible to pull a “TARDIS” rabbit out of the IoT hat.

Source URL: <http://electronicdesign.com/iot/achieve-impossible-iot-full-featured-apps-resource-limited-platforms>