RTOS power management emerges as a key for MCU-based IoT nodes

Nick Lethaby
IoT Ecosystem & TI-RTOS Product Manager
Texas Instruments
The emergence of the Internet of Things (IoT) promises to greatly increase the deployment of low-cost sensors or actuators, such as intelligent lighting, thermostats, and smoke detectors, which will need to communicate to the internet. These sensors and actuators (henceforth referred to as ‘IoT nodes’) will often need to run for months or years on coin cell or AA batteries. As a result, energy efficiency will be a critical concern for developers.

Users of laptops, mobile phones, and tablets are accustomed to have the operating system control power saving activities such as dimming displays or hibernation of the system after periods of no usage. However, these devices are based on sophisticated operating systems such as Windows, Linux, iOS, or Android. The low cost nature of IoT nodes will result in many implementations using MCUs with limited on-chip memory, precluding the use of such high-level operating systems. While traditional MCU developers are often satisfied with a set of low-level libraries for managing the hardware functionality, such an approach will often be insufficient for IoT nodes for several reasons:

1. Over the last decade, new silicon processes have created significantly more power leakage compared to devices built using older CMOS processes. To achieve the energy efficiency optimal for IoT nodes, more sophisticated power management features are being designed into MCUs aimed at IoT applications. Only providing a low-level software interface to these creates a significant learning curve for potential users, making it less likely they will exploit them.

2. Achieving optimal energy efficiency will require using more complex power down modes, where much of the SoC – CPU, peripherals, and memory – is shut down or power cycled. The silicon vendor should provide higher-level functions that implement these ultra-low power modes reliably to insulate the user from device-specific complexities. In addition, these higher-level power management solutions should address issues such as maintaining a reliable timebase in applications that are spending significant time in sleep modes.

3. Many IoT devices are originating from companies not traditionally associated with embedded systems development and it is anticipated that there will be insufficient traditional embedded developers to address all the opportunities available in the IoT marketplace. Developers of MCU-based IoT nodes who lack prior embedded development experience will certainly not want to be dealing with low-level register-abstraction APIs. They will expect something much closer to what is available in Windows or Linux where one can select a specific power down mode or have the operating system actively manage power.

In the MCU space, RTOS offerings have been a popular vehicle for delivering higher-level software functionality, such as network connectivity, to embedded developers. In this article, we will examine a power management framework that has been deployed in a basic operating system to enable developers to build energy-efficient IoT nodes. Since many software power management techniques are
inherently dependent on underlying hardware features, we will use the combination of TI-RTOS running on TI’s SimpleLink® CC2640 Ultra-low Power Wireless MCU as a real-world example of a power-aware RTOS executing on an MCU designed for low-power IoT node applications.

Since many MCU-based applications don’t use an RTOS, it is worth discussing why using an RTOS has some inherent advantages for energy efficient designs. The first of these is that the preemptive multitasking design paradigm encourages interrupt-driven rather than polling-based drivers. This eliminates unnecessary CPU usage simply spent polling peripheral registers. The second generic advantage is that the OS automatically drops into an idle thread when there is nothing to do, clarifying when power saving techniques can be applied. Furthermore, as we will see in later discussion, some of the more advanced power management capabilities require that the device drivers communicate with a centralized database that tracks which resources are in use. This fits naturally into an OS, which typically manages some or all of a system’s peripherals. Beyond these natural advantages, a power-aware RTOS must offer numerous other capabilities to achieve an optimal low power operating performance. We will examine the specific power management techniques that combine to produce a comprehensive framework. However, before getting into the specifics of the software, we will briefly overview some of essential hardware power management features that must be present on the device.

**Hardware Power Management Features**

To comprehend the software power management techniques explained later, it is necessary for the developer to have a basic understanding of some of the underlying hardware features that assist in effective power management:

**Clock Gating:** Clock Gating enables the clock to be turned off for a particular peripheral, which in turn reduces the power consumed by the peripheral’s logic.

**Power Domains:** Although turning off the clock to a peripheral eliminates most power consumption, depending on the process used to manufacture the device, there will often still be some power drain due to leakage. To address this issue, a SoC may implement power domains to completely shut off power to a particular circuit. Unlike clock gates, which will usually have a one-to-one correspondence to a peripheral, a power domain typically controls multiple peripherals, such as all the UARTs or all the serial I/O peripherals.

**Wake-up Generator:** To implement very aggressive low power modes, both the CPU and virtually all the peripherals domains are powered down. Since no interrupts can normally reach the CPU in these circumstances, addition logic that enables a subset of peripherals to wake up the CPU is required. The SoC designer must decide which interrupts can wake up the CPU and ensure that the wake-up generation logic is able to catch these interrupts, take the CPU out of reset so it can respond to the interrupt, and then forward the interrupt to the correct vector.

**CPU-independent High-resolution Timer:** Since the great majority of embedded applications have some time-driven events, it is essential that an
accurate timebase can be maintained across power saving modes. This requires a timer to be kept active while the rest of the SoC is powered down. This timer must have sufficient resolution to maintain something similar to a 1ms tick count and sufficient width to avoid rollovers during periods of deep sleep. The required resolution and width will depend on the CPU clock rate and how long the application will sleep.

**Fast wake up time and appropriate run-time performance:** Although not explicitly used for power management, the ability of the SoC to wake up quickly, complete work quickly, and go back to a low-power mode quickly is of paramount importance to maximize time in low power states. Important design choices here include having high-frequency clock source stabilize quickly and selecting the right CPU speed and performance so that the work can be done quickly.

We will discuss how an RTOS power manager utilizes these features, beginning with a discussion on how to minimize run-time power consumption.

**“CPU Active” Power Management Techniques**

Minimizing power consumption while the CPU is active primarily means aggressively managing power consumed by peripherals such as timers, serial ports, and radios. To do so, the RTOS power manager is reliant on the clock gating and power domains designed into the CC2640 silicon, which enable inactive peripherals to be powered down. Leveraging this hardware requires knowing when a particular peripheral is in use or not. Such knowledge can be tracked by an operating system and its associated device drivers. Each device driver must declare a dependency on the specific peripheral it will use. For example, when the UART driver is invoked, it declares a dependency to the OS power manager on the specific UART port (e.g., UART3). The OS power manager knows the clock gate and power domain that are associated with UART3 and verifies that these are enabled. If they are not, it enables them. When the driver completes execution, it informs the OS power manager to release the dependency on the chosen UART. The power manager maintains a database of dependency counts on the clock gates and power domains. Whenever the dependency count for a clock gate or power domain goes to zero, the power manager is responsible for disabling them to reduce power. These peripheral power downs are done during normal system run-time and help increase energy efficiency.

**Maximizing CPU Power Mode Efficiencies**

In many IoT nodes, it will be common for the SoC to spend much or even most of its time in some form of sleep mode. To maximize energy efficiency, it is critical to not only maximize the amount of time spent in sleep modes, but also appropriately utilize the most power efficient sleep modes where possible. Achieving the most power efficient sleep state will typically go beyond just putting the CPU into a sleep state. It may be desirable to power down memories in addition to on-chip peripherals. It also is essential to have a real-time clock or high-resolution timer be kept alive across power downs to ensure proper functioning of the application’s time-based functions. In the CC2640 implementation, the real-time clock is part of the “always on” hardware, so the application always has access to it. However, in other silicon implementations, it may be necessary for the power manager to specifically keep a timer or clock alive. There are a number of different techniques that can be utilized to ensure that sleep modes are as efficient as possible. We will begin with a discussion of tick suppression.
Tick Suppression

Embedded applications typically employ a regular timer interrupt as a ‘heartbeat’. This timer interrupt is used as the basis for calculating when any time-based activities such as periodic functions or timeouts should occur. For RTOS-based applications, this timer interrupt is known as the system tick, but no-OS applications will typically have a similar regular timer tick.

In practice, ticks execute periodically, at a rate sufficient for the most granular timing needed by the application. As a result, most system ticks will not result in a time-driven function being executed. In energy efficient applications, it is clearly undesirable to be woken up from a low-power mode just to service the system tick timer interrupt and then find there is nothing to do. Fortunately the OS knows when any periodic functions or timeouts are due to occur. To implement tick suppression, the OS reprograms the timer associated with the system tick so the next timer interrupt only occurs when the next time-based function must run. As illustrated in figure 1, this approach can eliminate the majority of timer interrupts associated with the system tick.

In the TI-RTOS implementation, the user has to simply set a configuration parameter to enable tick suppression. An alternative approach is to provide application-driven control through APIs. However, this forces the tick suppression logic into the application code as well as adding the overhead of APIs calls to a relatively simple operation. The core overhead of tick suppression is low as reprogramming the timer peripheral is simply a register write. TI-RTOS and most other RTOSs automatically track the next tick interval when work is scheduled for so this information is always available. A minor side effect is that it may take somewhat longer to execute OS system calls that must return tick counts, especially on architectures with poor math performance. This is because the count must be calculated, versus just returning a count variable that is simply incremented upon each timer interrupt.

Figure 1: The screen captures illustrate two oscilloscope traces. The top (yellow) trace tracks a timer that is triggering every one ms for use a comparison against the bottom (blue trace) that tracks triggering for a timer used to generate a one ms system tick for TI-RTOS. In the left hand screens, these two traces line up as expected. In the right hand screens, TI-RTOS tick suppression is enabled, resulting in many few interrupts as they occur only when work is actually scheduled for execution.
A Power Policy Manager

In earlier versions of the TI-RTOS power manager that worked on DSPs in mobile phone applications, decisions on when to go a particular low power mode and which power mode to select were pushed up to the application. Once a decision had been made to go to a specific power mode, a register/notify framework enabled the power manager to notify relevant system entities such as device drivers, which would then take steps to complete any activities and prepare for a power mode change. Once all the system entities had reported that they were ready, the power manager would then proceed with the power mode change. This approach was sufficient in the mobile phone space where large application development teams incorporate power management experts and the non-deterministic nature of the notification process is acceptable when the main CPU is running a high-level operating system such as Android, which inherently has a lot of overhead.

For IoT node applications, which often execute on MCUs, a simpler and lower-overhead approach is required. For the reasons earlier discussed concerning tick suppression, the OS power manager is well-placed to make any decision about transitioning to a different power mode. A program called a power policy manager was developed to provide a simple way to automatically decide on and manage power transitions. The register/notify framework was scaled back and largely replaced by the concept of a constraint. The power policy manager is configurable by the developer but comes with a set of default policies that can be used without the user having to understand significant levels of detail.

When a multitasking OS-based application has nothing to do, it drops into an idle loop and the OS can invoke the power policy manager. The role of the power policy manager is to determine which low power mode can be entered at this point. It is always safe to simply place the ARM core in a WaitForInterrupt (WFI) state as the core register contents are fully maintained and application execution can be resumed with minimal latency. However, since other power modes offer much greater power savings, the policy manager will first determine if one of these can be entered.

A common reason an application may drop into the idle loop is because one or more tasks are blocked waiting for peripheral I/O operations to complete. If completing these I/O operations or any other function is essential for the system’s correct operation, the application needs to be able to communicate this to the OS power manager. In the power manager implementation for the CC2650, the application informs the power manager of such critical functions by setting constraints. An example of when a constraint is appropriate would be when transmitting data over a BLE or 802.15.4 radio. An application that is waiting for acknowledgement or data from the wireless network would typically block on a semaphore. If no other application task needs to run, the application will then drop into the idle loop and the power policy would be run. Obviously, it would not be appropriate to shut down the radio and put the CPU into a long latency deep sleep mode, because this would result in the incoming BLE packets being lost. To prevent this from happening, the BLE stack or radio driver would set a constraint while it was operating. When its action was complete, it would release the constraint. The constraint should be limited to only the power down modes that would impair successful operation. For example, going into an IDLE state (see next section for more details of the different CC2650 power modes) may be safe for a particular operation, but not going into a STANDBY state. The power manager tracks constraints in a relatively similar manner to dependencies. However it’s important to understand that the power policy
only checks for constraints, not dependencies. The assumption is that power downs can be done regardless of on-going peripheral activity unless a peripheral’s associated stack or device driver sets a constraint.

Assuming constraints are not preventing the system from transitioning to a lower power mode, the power policy manager must weigh information from various sources to decide on which power saving mode to invoke. Each power saving mode is characterized by a specific latency, characterized by combining the time taken to perform the power down operation and time required for the SoC to fully wake-up and be ready for normal system execution. Similar to the technique used in tick suppression, the power policy will check when the next periodic functions or timeouts are due to occur and then compare this time against the latencies of the different power modes. It will then chose the lowest applicable power mode and program the appropriate wake-up configuration. The power policy understands the wake-up latencies from each power mode and therefore will program the wake-up to occur sufficiently early to ensure the processor is ready to respond instantly to perform the previously scheduled work. When the power policy triggers a transition to a new power mode, it will invoke callback functions registered by drivers that need notification of sleep transitions to shut down the peripheral’s activity. The default implementations of these callbacks are minimalistic and based on the assumption it is safe (due to no constraint being set) to shut down the peripheral as quickly as possible.

**Power Modes**

A key attribute of the Power Manager is that it provides proven implementations of a pre-defined set of power modes for a device. These are extensively tested to ensure reliable transitions to and from the mode. This eliminates the need for developers to become

---

**Figure 2:** The TI-RTOS power manager is able to deliver significantly lower power performance with minimal user intervention. Key system components, such as drivers and stacks, are power-aware and inform the power manager when it is safe to aggressively power down. The Power Policy Manager combines this information with the knowledge of the next upcoming events to decide which power mode can be selected.
RTOS power management emerges as a key for MCU-based IoT nodes

March 2015

... or for engineering resources to be consumed developing low-level power management code.

The power modes for the CC2650 are listed in Table 1 as an example of those that can be present for a device power-optimized for an IoT node. As can be seen from the data in the table, to achieve ultra-low power consumption, it is important to implement SoC-specific power modes that do much more than simply sleep the main CPU.

WaitForInterrupt mode simply results in gating the clock to portions of the main CPU. This may be used in any situation as it has virtually no latency. The primary role of the power policy manager is to determine if the IDLE or STANDBY modes can be used, as these greatly reduce power consumption, especially the latter. The IDLE mode will additionally power off some CPU logic completely, while retaining state of vital registers. It should be noted that no actions are taken in either the WaitForInterrupt or IDLE implementations to turn off peripherals. As a result, the actual power usage will vary depending on which peripheral and associated power domains are active.

In STANDBY mode, all peripheral domains are powered down, except for always on logic used for wake-up generation. The real-time clock in the ALWAYS ON domain is used to maintain an accurate time base while in this state. The device’s SRAM is put in retention mode and power supply is duty-cycled to achieve further power savings, while maintaining sufficient charge to maintain vital state.

The shutdown mode is provided for applications that wish to sleep for hours or even days. The main advantages of this mode compared to simply turning the whole SoC off is that any pin can be used to cause the SoC to power back up and there is no need for additional external circuitry to turn on the SoC. Because shutdown would only be used for very long power downs, the default power policy manager does not utilize it. The application can invoke it directly if appropriate or modify the power policy manager to use it.

## Summary

With the advent of the IoT triggering an explosion in battery-powered connected sensors and actuators, power management has become a critical technology for MCU developers. While aggressive power management strategies require specific features to be implemented in the silicon itself, it is equally important that a software layer be provided that enables such features to be easily leveraged. This is especially true in the IoT market, where many developers lack embedded experience. We illustrated RTOS-based power management components that provide low-level libraries for

<table>
<thead>
<tr>
<th>Power Mode</th>
<th>Wake-up Time to CPU Active</th>
<th>Current Used</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>NA</td>
<td>4.145 mA</td>
<td>Standard feature of ARM Cortex M core</td>
</tr>
<tr>
<td>WaitForInterrupt</td>
<td>A few cycles</td>
<td>2.028 mA</td>
<td>Standard feature of ARM Cortex M core</td>
</tr>
<tr>
<td>IDLE</td>
<td>1.4 µs</td>
<td>796 µA</td>
<td>SoC-specific</td>
</tr>
<tr>
<td>STANDBY</td>
<td>14 µs</td>
<td>1-2 µA</td>
<td>SoC-specific</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>700 µs</td>
<td>0.1 µA</td>
<td>SoC-specific</td>
</tr>
</tbody>
</table>

Table 1: The current consumed at the different power modes varies exponentially. The wake-up times and current draw are based on an ARM Cortex M3 Running at 48 MHz. In the WaitForInterrupt and IDLE measurements, no peripheral domains were active.
RTOS power management emerges as a key for MCU-based IoT nodes.

Managing peripheral clocks and domains and transitioning to and from specific power modes. These are complemented by power-aware drivers that enable the OS to understand when specific peripherals may be switched to a lower power mode. Finally, the OS power manager has the intelligence to decide when to transition to a lower power state, eliminating the need for the application to manage such details and simplifying the process for developers.

**Acknowledgements:** I would like to thank Scott Gary, Senior Member of Technical Staff at Texas Instruments, for providing technical insight on software power management in operating systems.
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio
Amplifiers
Data Converters
DLP® Products
DSP
Clocks and Timers
Interface
Logic
Power Mgmt
Microcontrollers
RFID
OMAP Applications Processors
Wireless Connectivity

Applications

Automotive and Transportation
Communications and Telecom
Computers and Peripherals
Consumer Electronics
Energy and Lighting
Industrial
Medical
Security
Space, Avionics and Defense
Video and Imaging

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2015, Texas Instruments Incorporated