RTOS power management: Essential for connected MCU-based IoT nodes

Nick Lethaby
IoT Ecosystem Manager
Connected MCU

Texas Instruments

Introduction

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 with the Internet. These sensors and actuators (henceforth referred to as "IoT nodes") often need to run for months or years on coin-cell or AA batteries. As a result, energy efficiency is a critical concern for developers.

Laptops, mobile phone and tablet users are accustomed to having operating systems (OSs) control power-saving activities such as dimming displays or system hibernation after periods of no usage. However, these devices are based on sophisticated OSs such as Windows®, Linux®, iOS® or Android™. The low-cost nature of IoT nodes will result in many implementations using microcontroller units (MCUs) with limited on-chip memory, thus precluding the use of high-level OSs. Although 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:

- Over the last decade, new silicon processes have created significantly more power leakage compared to devices built using older complementary metal-oxide semiconductor (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. A low-level software interface creates a learning curve for potential users, making it less likely they will exploit them.
- Achieving optimal energy efficiency requires using more complex power-down modes, where much of the system on chip (SoC)—including the central processing unit (CPU), peripherals and memory—shuts down or power cycles. Silicon vendors should provide higher-level functions that implement these ultra-low power states reliably to insulate users from device-specific complexities. In addition, these higher-level power-management solutions should address issues such as maintaining a reliable time base in applications that spend significant time in sleep modes.
- Many IoT devices originate from companies not traditionally associated with embedded systems development; it's unlikely that there will be enough traditional embedded developers to address all of the opportunities available in the IoT marketplace. MCU-based IoT node developers who lack previous embedded development experience will certainly not want to deal with low-level register-abstraction application programming interfaces (APIs). They will expect something much closer to what is available in Windows or Linux, where they can

select a specific power-down mode or have the OS actively manage power.

In the connected MCU space, real-time operating system (RTOS) offerings deliver higher-level software functionality such as network connectivity to embedded developers. In this white paper, I will examine a power-management framework deployed in the SimpleLink™ MCU software development kit (SDK) that enables developers to build energy-efficient IoT nodes. The SimpleLink MCU SDK includes an RTOS called TI-RTOS that serves as the foundation of the power-management framework; in this paper I will refer to TI-RTOS rather than to the SimpleLink MCU SDK. However, the power-management framework is OS independent and also runs, for example, on FreeRTOS, which is also supported in the SimpleLink MCU SDK. Since many software power-management techniques are inherently dependent on underlying hardware features, the combination of TI-RTOS running on TI's SimpleLink *Bluetooth*® low energy CC2640R2F ultra-low-power wireless MCU is a real-world example of a power-aware RTOS executing on an MCU designed for low-power IoT node applications.

An RTOS has some inherent advantages for energy-efficient designs. The first is that the preemptive multitasking design paradigm encourages interrupt-driven rather than polling-based drivers, which eliminates unnecessary CPU usage spent polling peripheral registers. The second advantage is that the OS automatically drops into an idle thread when there is nothing to do, clarifying when developers can apply power-saving techniques. Furthermore, as I will discuss later, some of the more advanced power-management capabilities require the device drivers to 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. I will examine specific power-management techniques that when combined produce a comprehensive framework. However, before getting into the software specifics, let's briefly review some 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, you should have a basic understanding of some of the underlying hardware features that assist in effective power management:

- **Clock gating**: Clock gating turns off the clock for a particular peripheral, which in turn reduces the power consumed by that 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, an SoC may implement power domains to completely shut off power to a particular circuit. Unlike clock gates, which usually have a one-to-one correspondence to a peripheral, a power domain typically controls multiple peripherals, such as all of the universal asynchronous receiver transmitters (UARTs) or serial input/output (I/O) peripherals.
- **Wake-up generator**: To implement very aggressive low-power states, both the CPU and virtually all peripheral domains power down. Since no interrupts can normally reach the CPU in these circumstances, additional logic that enables a subset of peripherals to wake

up the CPU is required. SoC designers must decide which interrupts can wake up the CPU and ensure that the wake-up generation logic catches these interrupts, takes the CPU out of reset so it can respond to the interrupt and then forwards 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 to maintain an accurate time base across power-saving modes. This requires keeping a timer active while powering down the rest of the SoC. The timer must have sufficient resolution to maintain something similar to a 1 ms 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 for.
- **Fast wake-up time and appropriate run-time performance**: Although not explicitly used for power management, the SoC's ability to wake up quickly, complete work quickly, and go back to a low-power state quickly is of paramount importance to maximize time in low-power states. Important design choices here include having a high-frequency clock source stabilize quickly, and selecting the right CPU speed and performance so that the work can be done quickly.

Let's discuss how an RTOS power manager uses 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 relies on the clock gating and power domains designed into the CC2640R2F silicon, which enables the power down of inactive peripherals. Leveraging this hardware requires knowing when a particular peripheral is in use or not. An OS and its associated device drivers can track such knowledge. Each device driver must declare a dependency on the specific peripheral it will use.

For example, when invoking the Serial Peripheral Interface (SPI) driver, that driver declares a dependency to the OS power manager on the specific SPI port (e.g., SPI2). The OS power manager knows the clock gate and power domain associated with SPI2 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 SPI. 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 occur during normal system run time and help increase energy efficiency.

Maximizing CPU power-state efficiencies

In many IoT nodes, the SoC spends 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 to appropriately use 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 is also essential

to have a real-time clock or high-resolution timer kept alive across power downs to ensure proper functioning of the application's time-based functions. In the CC2640R2F 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 to ensure that sleep modes are as efficient as possible. Let's begin with a discussion of tick suppression.

Tick suppression

Embedded applications typically employ a regular timer interrupt as a "heartbeat." This timer interrupt calculates 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.

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 the execution of a time-driven function. In energy-efficient applications, it is clearly undesirable to wake from a low-power state just to service the system tick timer interrupt and then find that 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 that the next timer interrupt only occurs when the next time-based function runs. 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 developer simply sets 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, and adds the overhead of API calls to a relatively simple operation. The core overhead of tick suppression is low, as reprogramming the timer peripheral is simply a register write.

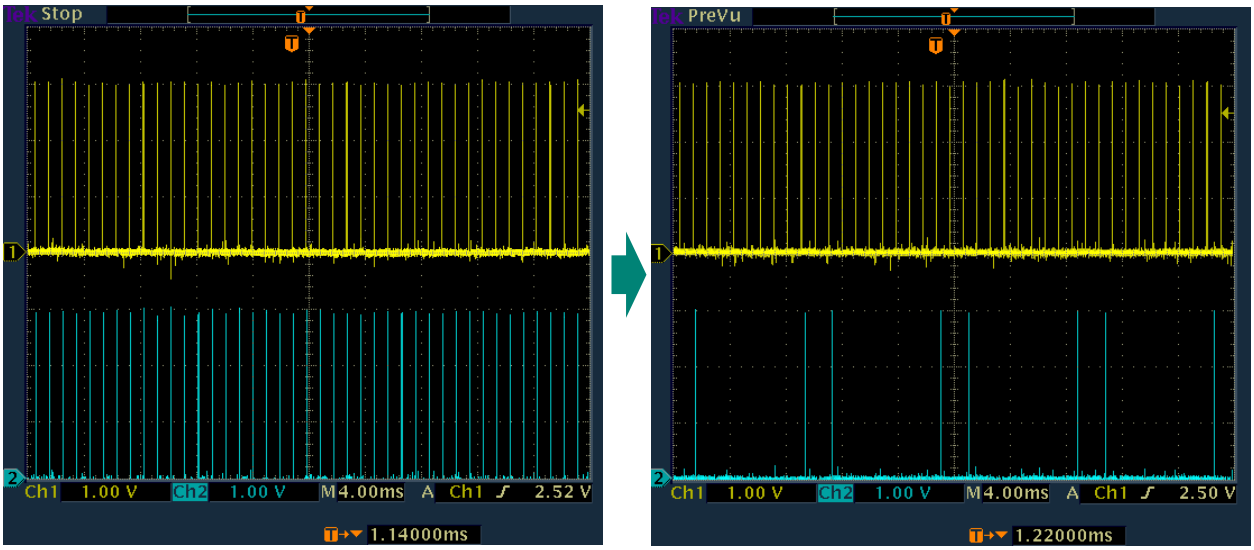

***Figure 1**. These screen captures illustrate two oscilloscope traces. The top (yellow) trace tracks a timer that triggers every 1 ms for use as a comparison against the bottom (blue trace), which tracks triggering for a timer used to generate a 1-ms system tick for TI-RTOS. In the left-side screens,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

TI-RTOS and most other RTOSs automatically track the next tick interval when work is scheduled, 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. That is because the count must be calculated versus just returning a count variable that increases upon each timer interrupt.

A power policy manager

In earlier versions of the TI-RTOS power manager, designed for digital signal processors (DSPs) in mobile phone applications, decisions on when to go into a particular low-power state and which power state to select were pushed up to the application. Once a decision had been made to go to a specific power state, 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-state change. Once all the system entities had reported that they were ready, the power manager would then proceed with the power-state change.

This approach was sufficient in the mobile phone space, where large application development teams incorporate power-management experts, and the nondeterministic nature of the notification process is acceptable when the main CPU is running a high-level OS such as Android, which inherently has a lot of overhead.

IoT node applications require a simpler and lower-overhead approach. Just as for tick suppression, the OS power manager is well-placed to make any decision about transitioning to a different power state. A function called a power policy manager provides a simple way to automatically decide on and manage power transitions. This function scales back the register/notify framework and makes greater use a concept known as a constraint to simplify decisions about power-state transitions. The power policy manager is configurable, but comes with a set of default policies that don't require understanding 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 state to enter at this point. It is always safe to simply place the ARM® core in a wait for interrupt (WFI) state, as the core register contents are fully maintained and application execution can resume with minimal latency. However, since other power states offer much greater power savings, the policy manager will first determine whether to enter one of them.

An application may drop into an idle loop 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 are 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 CC2640R2F wireless MCU, the application informs the power manager of critical functions by setting constraints.

An example of when a constraint is appropriate would be when transmitting data over a Bluetooth low energy or 802.15.4 radio. An application 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 runs. Obviously, it would not be appropriate to shut down the radio and put the CPU into a long-latency deep-sleep mode, because that would result in a loss of the incoming Bluetooth low energy packets. To prevent this from happening, the Bluetooth low energy stack or radio driver sets a constraint while it

is operating. When its action is complete, it releases the constraint.

The constraint should be limited to only those power-down modes that would impair successful operation. For example, going into an idle state (see the next section for more details about the different CC2640R2F device power states) 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. But the power policy only checks for constraints, not dependencies. The assumption is that power downs can occur—regardless of ongoing peripheral activity—unless a peripheral's associated stack or device driver sets a constraint.

Assuming that constraints are not preventing the system from transitioning to a lower power state, the power policy manager must weigh information from various sources to decide which power-saving mode to invoke. Each power-saving mode is characterized by a specific latency, calculated by combining the time to perform a power-down operation and the time that the SoC requires to fully wake up and be ready for normal system execution.

Much like the technique used in tick suppression, the power policy checks when the next periodic functions or timeouts are due to occur, and then compares this time against the latencies of the different power states. It will choose the lowest applicable power state and program the appropriate wake-up configuration. The power policy understands the wake-up latencies from each power state and therefore will program the wake-up to occur sufficiently early enough to ensure that the processor is ready to respond instantly to perform previously scheduled work.

When the power policy triggers a transition to a new power state, it invokes driver-registered callback functions 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 that it is safe (in the absence of any set constraints) to shut down the peripheral as quickly as possible.

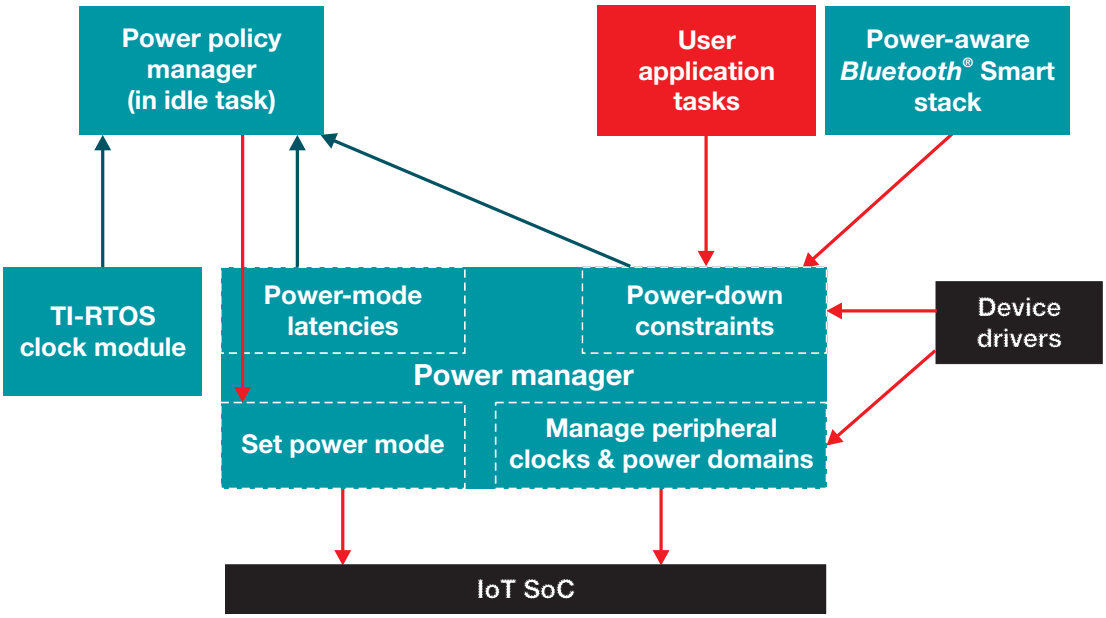

***Figure 2**. The TI-RTOS power manager delivers significantly lower power performance with minimal user intervention. Key system components such as drivers and stacks are power-aware and inform the power manager when it is safe to aggressively power down. The power policy manager combines this information with the knowledge of upcoming events to decide which power state to select.*

Power states

A key attribute of the power manager is that it provides proven implementations of a pre-defined set of power states for a device. These are extensively tested to ensure reliable transitions to and from the mode, eliminating the need for developers to become experts in the device's power-management features or the need to devote engineering resources to low-level power-management code development.

Table 1 lists the power states for the CC2640R2F wireless MCU as an example of those that can be present for a device power optimized for an IoT node. As you can see from the data in **Table 1**, to achieve ultra-low power consumption, it is important to implement SoC-specific power states that do much more than simply put the main CPU to sleep.

WFI mode simply results in gating the clock to portions of the main CPU, which applies 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 are usable, as these greatly reduce power consumption (especially the latter). See **Figure 3**. The idle mode will additionally power off some CPU logic completely, while retaining the state of vital registers. Neither the

Power state	Wake-up time to active CPU	Current used	Comments
Active	NA	4.145 mA	Standard feature of ARM Cortex®-M core
WFI	A few cycles	2.028 mA	Standard feature of ARM Cortex-M core
Idle	1.4 µs	796 µA	SoC-specific
Standby	14 µs	1–2 µA	SoC-specific
Shutdown	700 µs	0.1 µA	SoC-specific

***Table 1**. The current consumed at different power states varies exponentially. The wake-up times and current draw are based on an ARM Cortex-M3 core running at 48 MHz. In the WFI and idle measurements, no peripheral domains were active.*

***Figure 3**. This current-consumption profile highlights the < 10 µA average for maintaining a Bluetooth low energy connection with a 1-s connection interval. The device wakes from standby and the radio powers up to do a receive operation, followed by a transmit operation. The device then quickly transitions back to standby. Note the very low current used in standby mode.*

WFI or idle implementations take action to turn off peripherals. As a result, actual power usage will vary depending on which peripheral and associated power domains are active.

In standby mode, all peripheral domains power down, except for always-on logic used for wake-up generation. The real-time clock in the always-on domain maintains an accurate time base while in this state. The device's static random access memory (SRAM) goes in retention mode and the power supply is duty cycled to achieve further power savings, while sustaining a sufficient charge to maintain a vital state.

The shutdown mode is 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 cause the SoC to power back up; there is no need for additional external circuitry to turn on the SoC. Because shutdown would only occur for very long power downs, the default power policy manager does not use 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 the implementation of specific features in the silicon itself, it is equally important that a software layer enable easy leverage of such features. This is especially true in the IoT market, where many developers lack embedded experience.

In this paper, I reviewed RTOS-based power-management components that provide low-level libraries for managing peripheral clocks and domains and transitioning to and from specific power states. These components are complemented by power-aware drivers that enable the OS to understand when to turn off specific peripherals. 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 to thank Scott Gary, Senior Member Technical Staff at Texas Instruments, for providing technical insight on software power management in OSs.

Important Notice: The products and services of Texas Instruments Incorporated and its subsidiaries described herein are sold subject to TI's standard terms and conditions of sale. Customers are advised to obtain the most current and complete information about TI products and services before placing orders. TI assumes no liability for applications assistance, customer's applications or product designs, software performance, or infringement of patents. The publication of information regarding any other company's products or services does not constitute TI's approval, warranty or endorsement thereof.

The platform bar and SimpleLink are trademarks of Texas Instruments. All other trademarks are the property of their respective owners.

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources 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.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2017, Texas Instruments Incorporated