Power Management By Degrees

Scott Gary
Texas Instruments
sg@ti.com, 805-562-5118

Power Management by Degrees

Abstract
Power efficiency can determine the success or failure of a product in its marketplace. A wide variety of hardware and software-based power management techniques have been deployed, and many are being actively studied. Unfortunately, the most effective techniques are often complex, optimized for specific, highly-characterized applications and are limited in applicability on multi-purpose compute platforms. Also, there is very little cross-platform support for power management broadly available. This article first summarizes common techniques, and then takes a few of the higher-payoff techniques and describes how their support has been integrated as an adjunct module for a real-time operating system for digital signal processors (DSP) in such a way that a developer can pick and choose the appropriate techniques to meet specific application requirements. A sample audio application shows how this framework has been used in steps to progressively improve the power efficiency of the application.

Introduction
Power efficiency is an emerging requirement across a broad range of digital systems, and often distinguishes a product against its competition. From small, mobile devices, to rack-mounted processor farms, power efficiency is a key care-about.

For mobile devices, power efficiency means increased battery life, and a longer time between recharge. It also enables selection of smaller batteries, possibly a different battery technology and a corresponding reduction in product size.

For wired systems, power efficiency will typically enable a reduction in power supply capacity, a reduction in cooling requirements and fan noise and ultimately a reduction in product and energy costs. Power efficiency can allow an increase in component density as well. For example, a designer may be limited by the number of processors that can be placed on a circuit board simply because the cumulative power consumption would exceed compliance limits for the board's bus specification. Increased component density can result either in increased capacity, a reduction in product size, or both.

Power efficiency is determined both by hardware design and component choice and software-based runtime power management techniques.

The Basic Relationships
Before looking at the different power reduction techniques, it is worth reviewing a few basic relationships. The total power consumption of a CMOS circuit is the sum of both active and static power consumption (see Reference 3). Active consumption occurs when the circuit is active, switching from one logic state to another, and is due to both switching currents (those needed to charge internal nodes), and through currents (those

which flow when both P and N-channel transistors are both momentarily on). Active power can be approximated by the equation:

$$P_{active} \sim C_{pd} \times F \times V_{cc}^{2} \times N_{sw}$$

where C_{pd} is the dynamic capacitance, F is the switching frequency, V_{cc} is the supply voltage, and N_{sw} is the number of bits switching. An additional relationship is that voltage (V_{cc}) determines the maximum switching frequency (F) for stable operation. The key relationships here are: 1) the active power consumption is linearly related to switching frequency, and quadratically related to the supply voltage, and 2) the maximum switching frequency is determined by the supply voltage.

Static power consumption is the component that occurs even when the circuit is not switching, due to transistor leakage currents. Traditionally, CMOS static power consumption has been negligible when compared to active power. However, new, smaller, higher-performance transistors are bringing significant boosts in leakage currents, which require new attention to static power consumption. A variety of new circuit design techniques are being studied to address the higher leakage currents, including: multi-gate transistors, usage of strained silicon, metal replacements for polysilicon, etc. As these new processes come online, software support will be needed to augment the new hardware features, for example, to gate power to subsystems ON/OFF, with context save/restore, to dynamically activate low leakage retention modes, etc.

Common Power Reduction Techniques

A variety of power reduction techniques have been used to increase the power efficiency of embedded devices (see References 1 and 2). Tables 1 and 2 contain brief overviews of some hardware-design decisions and runtime techniques, respectively. Two runtime techniques that can provide significant reduction in active power are clock idling, and voltage and frequency (V/F) scaling.

Clock Idling

Looking at the active power equation, the simplest way to save on active power consumption is to stop clocking the circuit (i.e., idle it) when it is not needed. Most processors incorporate a mechanism to temporarily suspend the CPU while waiting for an external event. This idling of the CPU clock is typically triggered via a 'HALT' or 'IDLE' instruction, called during application or OS idle time. As an interrupt occurs to ready a thread for execution, the CPU will quickly wake to service the interrupt.

Some processors support idling the CPU clock only, others support selective idling of multiple clock domains. For example, the Texas Instruments (TI) TMS320C5510 DSP used in the example below has six separate clock domains partitioned, which can be selectively gated ON/OFF as needed: CPU, cache, DMA, clock generator, peripherals, and external memory interface. To idle a clock domain, a bitmask is written to an idle

Decision	Description
Choose a low-power silicon process	Choosing a power-efficient process (e.g., CMOS) is perhaps the most important up-front decision, and directly drives power efficiency.
Choose low-power components	Components that have been designed from the ground up for power efficiency (e.g., a processor with multiple clock and voltage domains) will provide dramatic savings in overall system power.
Partition separate voltage and clock domains	By partitioning separate domains different components can be wired to the appropriate power rail and clock line, eliminating the need for all circuitry to operate at the maximum required by any specific module.
Enable scaling of voltage and frequency	Designing in programmable clock generators and programmable voltage sources will enable dynamic voltage and frequency scaling by the application or OS. Also, designing the hardware to minimize scaling latencies will enable broader usage of the scaling techniques.
Enable gating of different voltages to modules	Some circuit modules (e.g., static RAMs) require less voltage in retention mode vs. normal operation mode. By designing in voltage gating circuitry, power consumption can be reduced during inactivity, while still retaining state.
Utilize interrupts to alleviate polling by software	Often software is required to poll an interface periodically to detect events. For example, a keypad interface routine might need to spin or periodically wake to detect and resolve a keypad input. Designing the interface to generate an interrupt on keypad input will not only simplify the software, but it will also enable event-driven processing and activation of processor idle and sleep modes while waiting for interrupts.
Use hierarchical memory model	Leveraging caches and instruction buffers can drastically reduce off-chip memory accesses, and subsequent power draw.
Reduce loading of outputs	Decreasing capacitive and DC loading on output pins will reduce total power consumption.
Boot with resources un-powered	Many systems boot in a fully active state, meaning full power consumption. If sub-systems can be left un-powered until they are actually needed, it will eliminate unnecessary wasted power.
Minimize number of active PLLs	Using shared clocks can reduce the number of active clock generators, and their corresponding power draw. For example, a processor's on-board PLL might be bypassed in favor of an external clock signal.
Use clock dividers for fast selection of an alternate frequency	A common barrier to highly-dynamic frequency scaling is the latency of re-locking a PLL on a frequency change. Adding a clock divider circuit at the output of the PLL will allow instantaneous selection of a different clock frequency.

Table 1, Up-Front Design Decisions

Technique	Description
Gate clocks off when not needed	By turning off clocks that are not needed, unnecessary active power consumption is eliminated.
On boot turn off un-necessary power consumers	Processors typically boot up fully powered, at a maximum clock rate. There will inevitably be resources powered that are not needed yet, or that may never be used in the course of the application. At boot time, the application or OS can traverse the system, turning off or idling unnecessary power consumers.
Gate power to subsystems only as needed	A system may include a power-hungry module that need not be powered at all times. For example, a mobile device may have radio subsystem that only needs to be ON when in range of the device it is to communicate with. By gating power on-demand, unnecessary power dissipation can be avoided.
Activate peripheral low-power modes	Some peripherals have built-in low power modes that can be activated when the peripheral is not immediately needed. For example, a device driver managing a codec over a serial port may command the codec to a low power mode when there is no audio to be played, or if the whole system is being transitioned to a low-power sleep mode.
Leverage peripheral activity detectors	Some peripherals (e.g., disk drives) have built-in activity detectors that can be programmed to power down the peripheral after a period of inactivity.
Utilize auto-refresh modes	Dynamic memories and displays will typically have a self or auto-refresh mode where the device will efficiently manage the refresh operation on its own.
Benchmark application to find minimum required frequency and voltage	Typically systems are designed with excess processing capacity built in, either for safety purposes, or for future extensibility and upgrades. For the latter case, a common development technique is to fully exercise and benchmark the application to determine excess capacity, and then 'dial-down' the operating frequency and voltage to that which enables the application to fully meet its requirements, but minimizes excess capacity. Frequency and voltage are usually not changed at runtime, but are set at boot time, based upon the benchmarking activity.
Adjust CPU frequency and voltage based upon gross activity	Another technique for addressing excess processing capacity is to periodically sample CPU utilization at runtime, and then dynamically adjust the frequency and voltage based upon the empirical utilization of the processor. This "interval-based scheduling" technique improves on the power-savings of the previous static benchmarking technique because it takes advantage of the dynamic variability of the application's processing needs.
Dynamically schedule CPU frequency and voltage to match predicted work load	The "interval-based scheduling" technique enables dynamic adjustments to processing capacity based upon history data, but typically does not do well at anticipating the future needs of the application, and is therefore not acceptable for systems with hard real-time deadlines. An alternate technique is to dynamically vary the CPU frequency and voltage based upon predicted workload. For example, using dynamic, fine-grained comparison of work completed vs. the worst-case execution time (WCET), and deadline

Table 2, Runtime Power Management Techniques

	of the next task, the CPU frequency and voltage can be dynamically tuned to the minimum required. This technique is most applicable to specialized systems with data-dependent processing requirements that can be accurately characterized. Inability to fully characterize an application usually limits the general applicability of this technique. Study of efficient and stable scheduling algorithms in the presence of dynamic frequency and voltage scaling is a topic of much on-going research (e.g., References 7-11).
Optimize execution speed of code	Developers often optimize their code for execution speed; in many situations the speed may be good enough, and further optimizations are not considered. When considering power consumption, faster code will typically mean more time for leveraging idle or sleep modes, or a greater reduction in the CPU frequency requirements. Note that in some situations speed optimizations may actually increase power consumption (e.g., more parallelism and subsequent circuit activity), but in others there may be power savings.
Use low-power code sequences and data patterns	Different processor instructions exercise different functional units and data paths, resulting in different power requirements. Additionally, because of data bus line capacitances and the inter-signal capacitances between bus lines, the amount of power required is affected by the data patterns that are transferred over the bus. And, the power requirements are affected by the signaling patterns chosen (1s vs. 0s) for external interfaces (e.g., serial ports). Analyzing the affects of individual instructions and data patterns is an extreme technique that is sometimes used to maximize power efficiency (see Reference 6).
Use code overlays to reduce fast memory requirements	For some applications, dynamically overlaying code from non-volatile to fast memory will reduce both the cost and power consumption of additional fast memory.
Enter a reduced capability mode on a power change	When there is a change in the capabilities of the power source, e.g., when a laptop goes from AC to battery power, a common technique is to enter a reduced capability mode with more aggressive runtime power management. A similar technique can be employed in battery-only systems, where a battery monitor detects reduced capacity, and activates more aggressive power management, such as slowing down the CPU, not enabling viewing via a power-hungry LCD display, etc.
Tradeoff accuracy vs. power consumption	It may be the case that an application is over-calculating results (e.g., using long integers when shorts would suffice). Accepting less accuracy in some calculations can drastically reduce processing requirements. For example, certain signal processing applications may be able to tolerate more noise in the results, which enables reduced processing, and reduced power consumption (see Reference 2).

Table 2, Runtime Power Management Techniques

configuration register, and then the CPU's IDLE instruction is called to propagate the settings.

Turning the clock to a module ON or OFF is usually as simple as gating the clock with a logic gate. This means that the latency to idle and wake the circuit is negligible, with no impact on real-time performance.

Voltage and Frequency Scaling

Looking back at the active power equation, if an application can reduce the CPU clock rate and still meet its processing requirements, it can have a proportional savings in active power dissipation. If the CPU frequency can be reduced safely, and this frequency is compatible with a lower operating voltage available to the CPU, then a potentially significant additional savings can occur by reducing the voltage, due to the quadratic relationship for V_{cc}. However, it is important to recognize that for a given task set, reducing the CPU clock rate also proportionally extends the execution time of the same task set, requiring careful analysis of the application to see if it still meets its real-time requirements. The potential savings provided by dynamic voltage and frequency scaling has been extensively studied in academic literature, with emphasis on ways to reduce the scaling latencies, improve the voltage scaling range, and schedule tasks so that real-time deadlines can still be met (see References 7-11).

Incorporating Power Management Features into an OS

To be flexible, and to speed time to market, systems are being assembled with programmable processors, using software components from multiple sources (algorithms, function libraries, drivers, OS, etc.). Clock idling and dynamic frequency and voltage scaling can have significant impact upon these components, and the operating system itself, and this is often a barrier for the deployment of these power saving techniques. For example, if clocks are idled independently by application threads, this can easily lead to deadlocks and missed deadlines. With frequency scaling, as the CPU frequency is scaled it can impact the OS time base, if, as is often the case, the same clock drives the OS timer peripheral. This means that the OS's time services, execution of periodic functions, and API timeouts will be disrupted. Additionally, once CPU frequency is scaled, it will affect the application's ability to meet its real-time deadlines. So, for example, a different scheduling algorithm may be needed by the OS to ensure the stability of the application. Frequency scaling will impact device drivers as well, as many may need to be notified of frequency changes so that they can reprogram their peripheral registers to adjust to the new frequency. Device drivers may also need to know about changes to the system sleep state, so that they can command their external devices to low power states during system sleep. Looking at these impacts, it is clear that the operating system or system-wide application framework must become power-aware. The focus of this article now shifts to the development of a supplementary Power Manager module for an operating system.

The first target implementation for the Power Manager was the DSP/BIOS™ operating system from TI (see Reference 4). DSP/BIOS supports three pre-emptive thread types: tasks (blockable), software interrupts (light-weight run-to-completion threads), and hardware interrupts. Priority based scheduling is used at each thread level. The kernel includes standard synchronization primitives (e.g., semaphores, mailboxes, etc.) and includes a memory manager with support for multiple heaps for managing multiple regions and types of memories. A device model is defined for implementing streaming, device-independent I/O. The kernel also provides clock-based services: high and low-resolution time APIs, functions that run on each system tick, periodic functions that run on multiples of system ticks, and timeouts for blocking API calls.

Note that although the Power Manager implementation described below is on a specific OS, the concepts can easily be carried over to other operating systems, or even application environments without an OS; nothing in the approach is specific to DSP/BIOS.

The overall goal of the Power Manager development was to pick a few high-payoff power management techniques, and incorporate them into the OS in such a way that application developers can easily pick and choose those techniques appropriate for their specific application. From the outset, key goals were efficiency, flexibility and loose coupling to the operating system itself.

Requirements

The key requirements for the Power Manager were:

1. Power management actions are application-triggered, vs. OS-triggered. Major decisions to change the DSP operating mode or capabilities are made by the application, and are facilitated by calls to the Power Manager. (Future versions of the Power Manager can take a more active role in the decision process, but the first step is to leave this responsibility with the application).
2. Power management actions are triggered by control portions of the application, but should be transparent to the majority of the application code. For example, high-value, optimized DSP algorithms should not need to be rewritten to function within the power-managed environment.
3. The Power Manager must support voltage and frequency (V/F) scaling, and frequency-only scaling (for when V scaling not implemented, or the V scaling latency is too large, or to quickly throttle the clock for temperature control).
4. The Power Manager must support application-directed clock idling, and automatically leverage CPU and cache idling during OS time. It must also support any available chip sleep modes.
5. The Power Manager must coordinate processing of power events across the entire application (e.g., application code, drivers, and the OS itself), notifying clients who have registered for notification when a particular event has occurred.

6. Power management features must be available at any thread context, and must be available for multiple instances of a particular client (e.g., multiple instances of a device driver).
7. Upon notifying a client of a power event, the Power Manager must support a delayed completion (by the client) of the event processing, notifying other clients while waiting for the completion signal from the delayed client.
8. The Power Manager must be scalable and portable to different platforms that have different capabilities. For example, many platforms will not have a programmable voltage regulator; the voltage scaling functionality of the Power Manager will not be implemented on this platform, but other functions, including frequency-only scaling should still be available.

The Power Manager Module

The Power Manager module, designated as PWRM, was added as a supplementary module to DSP/BIOS, as illustrated in Figure 1. Conceptually, the Power Manager is loosely coupled, and sits aside the kernel; it does not exist as another task in the system, but as a set of APIs that execute in the context of application control threads and device drivers. PWRM uses a few basic kernel services: disable/enable scheduling, atomic queue operations, and logging facilities for instrumentation.

No kernel or device model modifications were necessary to incorporate PWRM. However, on platforms where the CPU clock is coupled to the OS timer clock, a few supplementary routines are needed for the DSP/BIOS clock module (CLK) to allow it to adapt the OS timer on frequency scaling events, (as a client of PWRM).

PWRM interfaces directly to the DSP hardware by writing and reading a clock idle configuration register, and through a platform-specific Power Scaling Library (PSL) (see Reference 5), that controls CPU clock rate and voltage-regulation circuitry. The PSL isolates PWRM and the rest of the application from the low-level implementation details of the frequency and voltage control hardware, and ensures safe transitions between the chip-vendor characterized voltage and frequency pairs (setpoints).

The role of PWRM is to manage all things power-related in the DSP/BIOS application, as statically configured by the application developer, and as dynamically called at runtime. It provides interfaces allowing the application developer to: 1) selectively idle clock domains, and specify which domains should be automatically idled by PWRM during OS idle time; 2) specify a power-saving function to be called by PWRM at boot time to turn off unnecessary resources; 3) dynamically change voltage and frequency at runtime, based upon the application's specific processing requirements; 4) specify if voltage should be scaled along with frequency, and if application execution can continue during down-voltage transitions (once frequency has been scaled back); 5) activate chip-specific and custom sleep modes; and 6) provide a "central registry" whereby entities that care about power management events can register to be notified on those specific power

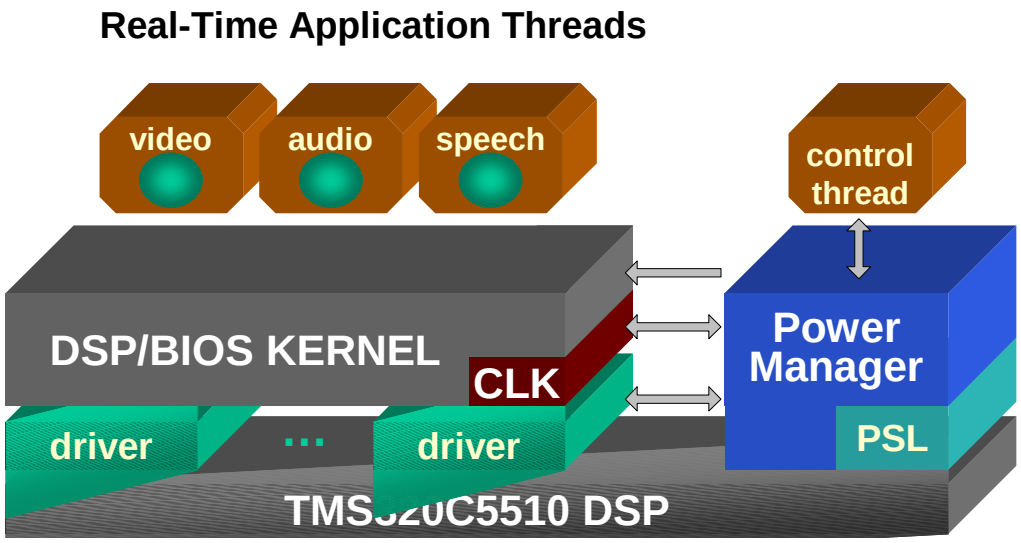

Figure 1, Power Manager Partitioning

events they care about (e.g., 'about to change V/F setpoint', 'done changing V/F setpoint', 'going to sleep mode', 'awake from sleep mode', 'power failing', 'battery below certain capacity, etc.). The registration and notification concept is illustrated in the example shown in Figure 2.

In this example clients register and get notified on certain V/F scaling events. The numbered steps listed in the figure are described below. Steps 1-3 correspond to the registration sequence, and Steps 4-7 the scaling sequence.

1. Application code registers to be notified on V/F setpoint changes. For example, the DSP requires different external memory interface (EMIF) settings for different setpoints, so the application registers control code with PWRM allowing EMIF settings to be changed to follow the change in setpoint.
2. A driver using DMA to transfer data to/from external memory registers to be notified of a V/F setpoint change. For example, prior to a setpoint change the driver may need to temporarily stop DMA operations to external memory; following the change, the driver will need notification to restart the DMA transfers.
3. Purchased binary content similarly registers with PWRM for notification on setpoint changes.
4. The application comes to a decision to change the V/F setpoint (e.g., a change in the device's mode, or maybe a task boundary), and calls the PWRM API to initiate the setpoint change.
5. PWRM checks if the requested new setpoint is allowed for all registered clients, (based on parameters they passed in at registration time), and then notifies all registered clients of the impending setpoint change.
6. PWRM calls into the Power Scaling Library to change the voltage and frequency setpoint. PSL writes to the clock generation and voltage regulation hardware as appropriate to safely change the V/F setpoint.
7. Following the setpoint change, PWRM notifies clients that the setpoint has been changed.

Note that this registration/notification mechanism can be utilized to extend power savings beyond the CPU core. For example, a device driver managing a display device can register for custom events, such as expiration of user inactivity timers, and use these notifications to power manage the display device.

Configuration of the Power Manager

DSP/BIOS enables both static (design time) and dynamic (runtime) creation of kernel objects. Many of the configuration parameters for the Power Manager relate to design time decisions, therefore, static configuration of PWRM was added to the configuration database files used by the DSP/BIOS graphical Configuration Tool. The following Power Manager properties were made statically configurable:

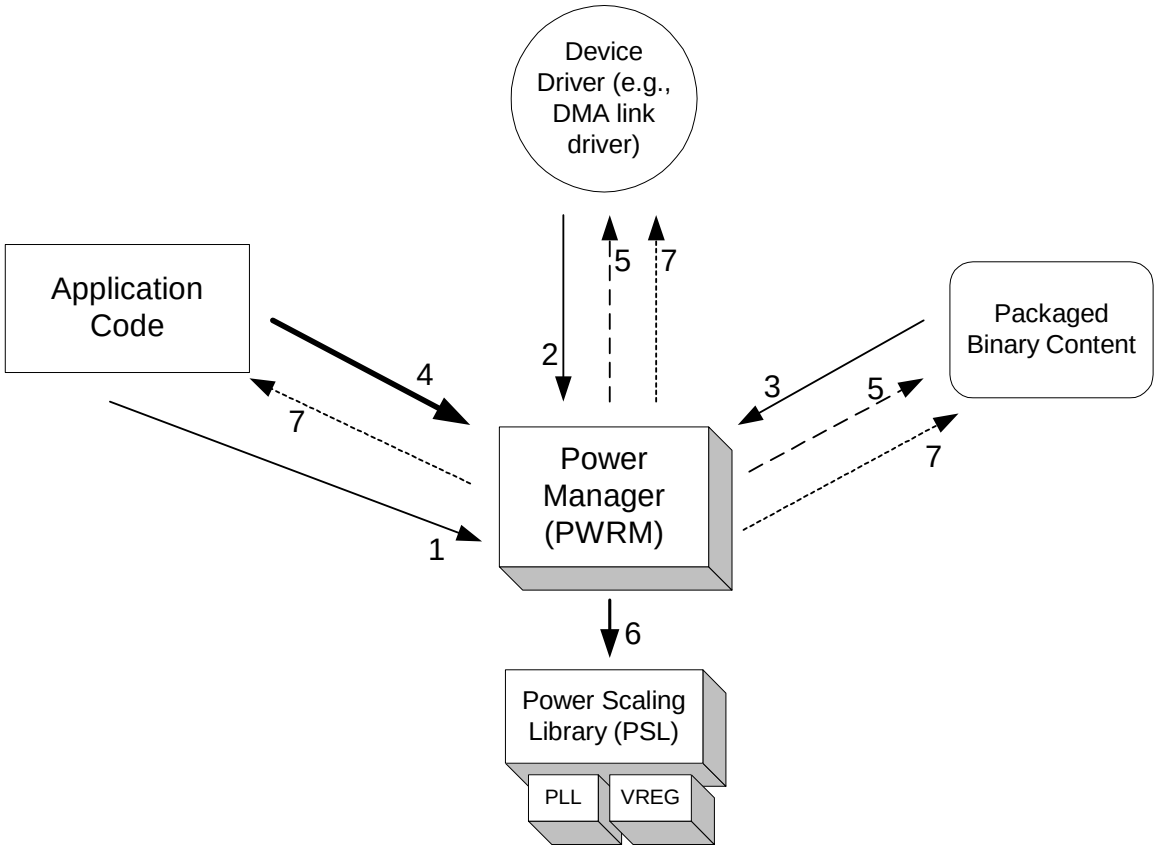

Figure 2, Power Event Notification Concept

1. Enabling/Disabling of the Power Manager
2. Whether a developer-defined function should be called at boot time
3. The name of the developer-defined boot function
4. Whether the OS clock should be adapted upon frequency scaling events
5. The clock domains to be automatically idled in the OS idle loop
6. Enabling/Disabling voltage and frequency scaling support
7. The initial frequency of the CPU at boot
8. The initial voltage of the CPU at boot
9. Enabling/Disabling scaling of voltage along with frequency
10. Waiting while voltage is scaling down
11. The clock domains to be idled when the DSP is put to deep sleep mode
12. The interrupts allowed to wake the DSP from deep sleep

The sample screen shot shown in Figure 3 illustrates the configuration process, showing the first configuration tab sheet for Power Manager General Properties.

Power Manager Application Programming Interfaces (APIs)
The runtime APIs for the Power Manager are summarized in Table 3. Each API has an appropriate set of pre-defined return/error codes, allowing for example, for graceful failure if the function is not implemented on the particular platform, indication of a busy status if the Power Manager is currently busy processing a previous application request, etc.

An Audio Example

A simple audio application was used as a test vehicle to benchmark the effectiveness of the Power Manager features. A working audio application was modified in steps, to gauge the payoff as different power saving features were turned on. A support routine was added to the application to monitor a set of four DIP switches on the DSP board. When each DIP switch was toggled the support code would call appropriate PWRM APIs to activate the different Power Manager features. Using these switches allowed the application to free-run, without interaction from the emulator and debugger, which would have perturbed power measurements. A 1 Ohm resistor was placed in series with the DSP core supply, and a voltmeter was used to measure the voltage drop across this resistor to determine the DSP core current.

The structure of the audio application was as follows: audio data samples were DMAed in and out of the DSP from a stereo codec, via on-chip multi-channel buffered serial ports. The audio input was an MP3 player, and a pair of speakers was used for output. The stereo codec used a USB clock already available on the DSP board to both sample audio at 44KHz, and to master the serial port communications to the DSP (so that no adjustments needed to be made to the serial ports as the DSP clock was scaled).

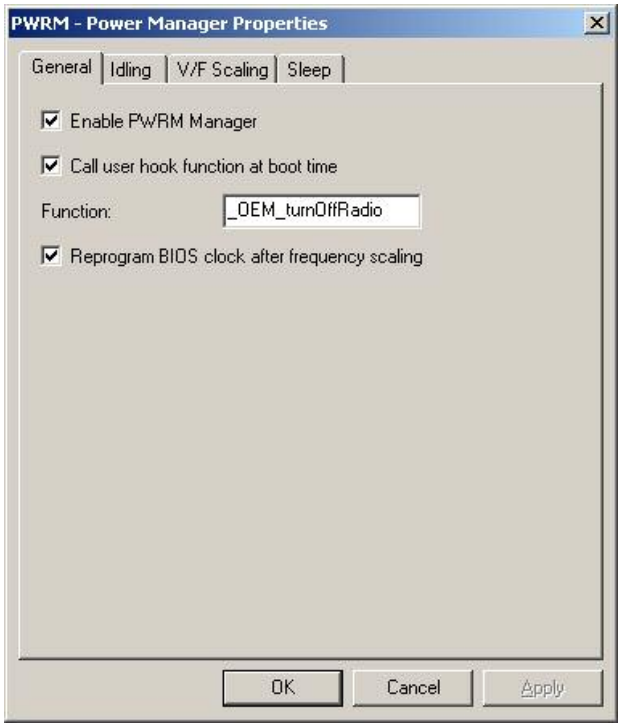

Figure 3, Configuring Power Manager General Properties

Function	Purpose
PWRM_getCapabilities	Get information on PWRM's capabilities on the current platform
PWRM_getCurrentSetpoint	Get the current V/F setpoint in effect
PWRM_getNumSetpoints	Get the number of setpoints supported for the current platform
PWRM_getSetpointInfo	Get the corresponding frequency and CPU core voltage for a setpoint
PWRM_getTransitionLatency	Get the latency to scale from one specific setpoint to another specific setpoint
PWRM_changeSetpoint	Initiate a change to the V/F setpoint
PWRM_configure	Set new configuration parameters for PWRM
PWRM_idleClocks	Immediately idle clock domains
PWRM_registerNotify	Register a function to be called on a specific power event
PWRM_unregisterNotify	Unregister for an event notification from PWRM.
PWRM_sleepDSP	Transition the DSP to a new sleep state
PWRM_snoozeDSP	Put the DSP to sleep for a specified number of milliseconds

Table 3, Power Manager APIs

Step 1
The first step was to simply configure the Power Manager to be ON, and enable idling of the CPU and cache in the OS's idle loop. This was done via check boxes in the Power Manager properties pages in the DSP/BIOS Configuration Tool, and then the application was rebuilt and run. To see the difference with/without idling, with the application running the first DIP switch was toggled up and down to see the resulting savings in power. When the switch transitioned to down, PWRM_configure() API was called by the monitor code to change a bitmask, to specify idling of the CPU and cache from the BIOS idle loop; when the switch transitioned up, PWRM_configure() was called again, to disable idling of CPU and cache in the OS idle loop. The result was a 32% reduction in DSP core power by idling of the CPU and cache domains when no application threads were ready to run.

Step 2
The next step was to measure the effectiveness of frequency-only scaling. Because DMA was used to pump audio data via the buffered ports, the CPU was only lightly loaded at 200MHz. By stepping down the frequency in accordance with the setpoints configured in the Power Scaling Library, it was found that there was no audio degradation down to 24MHz, and there was still some spare idle time. The control code monitoring the DIP switches would call PWRM_changeSetpoint() to alternate the CPU frequency between 200 MHz and 24 MHz as DIP switch 2 was toggled down and up. The results showed an 85% reduction in core power when running at 24 MHz.

Step 3
The next test was to combine CPU and cache idling in the OS idle loop with the frequency scale to 24MHz. The result showed an 89% reduction in DSP core power, compared to the baseline application.

Step 4
The next step was to add in voltage scaling along with the frequency scaling. DIP switch 3 was configure to toggle the DSP voltage/frequency between 200MHz/1.6volt and 24MHz/1.1volt. With CPU and cache idling still enabled via the first DIP switch, the result was a 95% reduction the core power, still with no audio degradation.

Step 5
The last step was to measure static (standby) power consumption when the DSP was put to deep sleep, while waiting for an external application interrupt. The fourth DIP switch was used to activate deep sleep (via the PWRM_sleepDSP() API), which showed a static power consumption that was a reduction of 99.9% from the power consumed at 200MHz.

In addition to the above steps, a boot function was configured to turn off certain board features and clocks at boot time, including the audio codec, which was later powered back up when the device driver was opened. The registration and notification mechanism was also used, to put the codec in its low power mode when the DSP went to deep sleep.

Results
Table 4 summarizes the power measurements described above. Without the Power Manager the DSP CPU was using 209mW to play back audio. As different Power Manager features were turned on, the core power was reduced in steps, down to 9.59mW, with no degradation in audio quality.

Summary

Significant power savings can be achieved by taking a phased approach to implementing power management into embedded systems. One does not need to wait for ongoing research into new techniques to bear fruit; incorporating basic, practical support for a few key techniques into an operating system or application framework can, in some cases, result in CPU power savings in excess of 95%. The power management support can be designed in such a way that it is: easy to use, allows developers to pick and choose the specific techniques that meet their specific application requirements, and is highly portable. This framework can then be used as a springboard for adding in more aggressive, application-specific techniques, as well as future power management techniques as they come online.

References

1. *Power Management Techniques for Real-Time Embedded Systems*, Proceedings of the Embedded Systems Conference, Scott Gary, April, 2003.
2. *System-Level Power Optimization: Techniques and Tools*, ISLPED99, Luca Benini, Giovanni DeMicheli, 1999.
3. *CMOS Power Consumption and C_{pd} Calculation*, Texas Instruments, SCAA035B, June 1997.
4. *TMS320 DSP/BIOS User's Guide*, Texas Instruments, SPRU423, February 2001.
5. *Using the Power Scaling Library on the TMS320C5510*, Texas Instruments, SPRA848, Rob Cyran, October 2002.
6. *Application Report: Calculation of TMS320LC54x Power Dissipation*, Texas Instruments, SPRA164, Clay Turner, June 1997.
7. *Run-time Power Control Scheme Using Software Feedback Loop for Low-Power Real-time Applications*, IEEE ISBN 0-7803-5974-7, Seongsoo Lee, Takayasu Sakurai, 2000.
8. *Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications*, IEEE Design & Test of Computers, Dongkun Shin, Jihong Kim, Seongsoo Lee, 2001.
9. *Run-time Voltage Hopping for Low-power Real-time Systems*, DAC2000, ACM 1-58113-188-7, Seongsoo Lee, Takayasu Sakurai 2000.

Power Manager Mode	DSP Core Power (mW)	Power Reduction (%)
Power Manager Disabled (200MHz, 1.6v)	209	-
Idling CPU and cache in OS idle loop	142	32
No idling, frequency scaled to 24MHz	32.2	85
CPU and cache idling, frequency scaled to 24MHz	22.5	89
CPU and cache idling, frequency scaled to 24MHz, voltage scaled to 1.1v	9.59	95
Deep sleep (no audio)	0.3	99.9

Table 4, Power Measurements for the Audio Application

10. *Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems*, DAC2001, ACM 1-58113-297-2, Jiong Luo, Niraj Jha, 2001.
11. *Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors*, DAC2001, ACM 1-58113-297-2, Gang Quan, Xiaobo Hu, 2001.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI 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 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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265