How to exploit 17 tried and true DSP power optimization techniques for wireless applications

Rob Oshana, Engineering Manager, Texas Instruments - November 15, 2005

Code size, speed and power consumption all have a significant impact on the the system-level product that integrates a DSP. The more power an embedded application consumes, for example, the larger the battery or fan required to drive it.

To reduce power, an application must run in as few cycles as possible because each cycle consumes a measurable amount of energy. In this sense, performance and power optimization are similar using the least number of cycles is an excellent way to meet both performance and power optimization goals.

Although performance and power optimization strategies may share a similar goal, there are subtle differences in how those goals are achieved. This article will explore those differences from the perspective of wireless system design and it will discuss the resulting strategies.

Wireless media systems
Power optimization has been of growing importance in wireless media systems. These wireless systems products have been growing rapidly in the last several years, including broadband wireless coverage that enables devices with more and more powerful processors to perform a wide variety of tasks.

In terms of use scenarios, designers must consider how to design systems that experience more frequent use for longer periods of time. These use models require more battery power to sustain.

Unfortunately, battery technology is not progressing as fast as embedded processor technology. Given that this situation is unlikely to change in the near future, there are at least four different techniques to reduce power consumption in wireless media systems:

- Algorithmic: This approach minimizes power consumption at the terminals by adjusting the complexity of source compression and transmitter power. This technique exploits the significant difference between video processing and voice processing. There is a more complex signal processing requirement in video, and consequently more power consumption. So performance can be varied on the video encode side to reach an optimum distortion/power point for a given application.
- Software design and implementation: By applying power optimization techniques to the software design and implementation phase, significant power savings can be realized.
- Hardware: Processor technology and design allows low power operation and management. DSPs for example, are designed to performed high intensity wireless video processing using low power process technology, on chip peripheral and debug management and other static and dynamic
power control mechanisms.

This article will focus on software and hardware design and their implementation.

**Power use in embedded applications**

Most of the power consumed in embedded applications comes not from the CPU but from the processes used to move data from memory to the CPU. Each time the CPU accesses external memory, busses are turned on and other functional units must be powered on and utilized to get the data to the CPU.

This is where the majority of power is consumed. Overall power consumption of the application will be reduced significantly if the programmer can: (1) design embedded applications to minimize the use of external memory; (2) efficiently move data into and out of the CPU; and (3) make efficient use of peripherals and cache to prevent cache thrashing, cycling of peripherals on and off.

The two main power contributors in embedded applications are the compute block, which includes the CPU and is where the algorithmic functions are performed, and the memory transfer block, which is where the memory subsystems are utilized by the application. The memory transfer block is where the majority of the power is consumed by a DSP application.

The process of power optimization is a key application management component because it’s a lifecycle process that should be planned and managed accordingly. The development team lead must understand the entire system, hardware as well as software in order to manage this effort effectively.

It is not entirely a software related issue.

For example, some embedded Digital Signal Processors (DSPs) are designed for low-power operation. These DSPs are built with specific manufacturing processes tailored for low power consumption. These architectures use special hardware capability to manage power consumption on the device more efficiently while at the same time maintaining the architectural features necessary to provide the performance expected from a DSP.

**Power optimization techniques in DSP**

DSP hardware has incorporated a number of power saving and optimization techniques into the architectures of these devices. Aside from the natural savings from continued improvement in semiconductor processor technology, architecture advances have also been made to modern DSP devices. These include:

- Increased parallelism and dedicated instructions for a class of DSP operations including the idle instruction which forces the program to wait until an interrupt or a reset occurs. The power-down mode during the idle depends on a configuration register.
- Extensive use of clock gating peripherals, memory units, clock tree structure and logic fanout
- Memory access—clock-gated memory access, reduction in access requirements [for example, internal buffer queues (IBQ or loop buffers)]
- Custom designed data paths for both address and data
- Designed using low leakage process technology

The following steps are required to calculate overall device power consumption.

1. Algorithm partitioning: The algorithm under consideration should be broken into sections of unique device activity and the power requirements for these sections calculated separately. These sections can then be time averaged to determine the overall device current requirements.
2. CPU activity: The current contribution to CPU activity can be determined by examining code and determining the time-averaged current for each algorithm partition.

3. Memory usage: Scale the current in step 2 based on memory usage. Use of on-chip memory requires less current than off-ship (because of additional current due to external interface). Running from ROM vs RAM is also consumes less power.

4. Peripherals: Consider the additional current required by the timer, standard serial port, host port interface.

5. Current due to outputs: Consider the current required by the algorithm to operate the external address and data buses.

6. Calculation of average current: If the power supply is observed over the full duration of device activity, different segments of activity will exhibit different current levels for different lengths of time.

7. Effects of temperature and supply voltage on device operating current: Include the effects of these factors after the total device current has been calculated.

An example of a DSP architecture is shown in Figure 1. This is a model of the TI 6x architecture which has a relatively clean and orthogonal clustered VLIW instruction set that represents a good target for advanced ILP compilers. These features help in system level power consumption reduction. Because of the focus on performance and power, many DSPs have exposed pipeline and control to the programming interface to allow more control of instruction selection and scheduling.

Processor Architecture and Process Technology
Designed For Low Power

Figure 1 represents a power logical model of this device. The peripherals are represented as switches that can be turned on and off. The ability to switch peripherals off when not in use makes a significant contribution to overall system level power optimization and reduction. The sliding scales on the left represent the ability to scale power and voltage in these devices. The external Memory Interface (EMIF), cache subsystem, and Direct Memory Access (DMA) can also be switched off when not in use. The memory can also be configured or scaled depending on the application to provide optimal memory configurations which can provide power savings.

Software Power Optimization Techniques for Embedded Applications
Listed below are the top 17 power optimization techniques for the DSP engineer to use in guiding the application-level process of power optimization. A majority of these techniques are independent in the sense that they can be used in isolation of in junction with other techniques.
After summarizing these techniques, we will look in more detail at a couple of these techniques.

1. Architect software to have natural “idle” points (including a low power boot). By intelligent architecting of the software, stopping or idle points in the application can be used as triggering points for turning off parts of the device or peripherals including the DMA. This makes the global management of the power efficiency of an application easier to manage.

2. Use interrupt-driven programming (instead of polling) and use the DSP operating system to block the application as necessary. 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.

3. Code and data blocks should be placed close to processor to minimize off-chip accesses (and overlays from non-volatile to fast memory). Minimize off chip memory accesses that require turning on and off buses and other peripherals.

4. Smart placement of code and data to allow frequently accessed code/data to be close to the CPU (and use hierarchical memory models as much as possible including the cache) will require less access time and less hardware get to the data.

5. Perform necessary code and data size optimizations to reduce the application footprint, memory and corresponding leakage caused by a larger required memory for the application.

6. Critical path reduction: The goal with critical path reduction is to reduce the critical path in the software application (effectively identifying the worst case performance scenario) so that the system supply voltage can be lowered while still keeping system throughput fixed.

7. Optimize for speed: There is, in many ways, and direct relationship between performance and power optimization. By optimizing the application for performance the DSP engineer can build up a larger time base for more CPU idle modes or reduced CPU frequency opportunities (the engineer must benchmark and experiment to perform the correct tradeoffs!). 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.

8. Don’t over calculate, use minimum data widths if at all possible. This leads to reduced bus activity which saves power as well as using smaller multipliers which will also save power.

9. Use hierarchical memory model: Leveraging caches and instruction buffers can drastically reduce off-chip memory accesses, and subsequent power draw.

10. Use the DMA for efficient transfer of data (instead of the CPU). This optimization technique has already been discussed.

11. Use co-processors to efficiently handle/accelerate frequent/specialized processing. Co-processors are optimized to perform certain computationally intensive tasks very efficiently which also saves power.

12. 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.

13. Use more buffering and batch processing to allow more computation at once and more time in low power modes.

14. Implement a boot time power savings module: 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.

15. 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. Analyzing the affects of individual instructions and data patterns is an extreme technique that is sometimes used to maximize power efficiency.

16. Benchmark the 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.

17. 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

**Optimizing for speed**

Each instruction a DSP executes consumes a certain amount of power (the cheapest instruction is the one you do not execute!). Several of the lessons learned from code performance optimization can be directly applied to the power optimization problem.

- The fewer instructions that are in your program, the less power it will consume
- Fewer instructions mean a smaller memory footprint – the smaller the memory footprint, the fewer memory chips are required to keep powered up
- Reducing the number of fetches from memory reduces the amount of power required to make the fetch from memory
- The smaller the application, the more likely that significant parts of the application can be cached which reduced power required to execute the application

The general approach for optimizing power using code modeling starts with profiling the application. Run a cycle profiler on the code to identify the “hot spots” - those areas in the code where most of the cycles are being spent.

DSP applications spend a disproportionate amount of time in a small number of critical loops. These critical loops are also most likely the place where the application is consuming the most power. Focusing on this small number of loops can provide most, if not all, of the power reduction needed for the application.

As discussed earlier, many DSPs are designed to execute critical loops very efficiently using hardware support such as zero-overhead looping mechanisms.

Fetches from memory consume power.

Memory busses must be switched on and off and this consumes power. Modern optimizing DSP compilers will attempt to avoid fetching the same value repeatedly from memory. However, if the
code has complex pointer manipulations, particularly multiple pointer instructions, the compiler
might not be able to determine that a referenced memory location always has the same value.

Therefore, the DSP programmer should avoid using complicated pointer expressions wherever
possible. Use array structures as an alternative. Write your application in a straightforward manner.
This is a situation where becoming more clever with code production will not lead to the most
optimal solution. Many modern optimizing compilers can make many of these transformations for
you automatically—give the compiler a chance before making these code transformation decisions.

The link command file can be used to direct the placement of critical sections of the application in on
chip memory, which is not only faster than off chip memory but also consumes less power as well.

DSP programmers should attempt to place as much of the critical code and data segments as
possible in on-chip memory. Allocating each critical section its own section gives the linker more
flexibility and freedom to pack the structures as it see fit.

The DSP programmer should take advantage of existing power-saving library features and use them
in the application where needed. Keep in mind that the problems of size, speed, and power are not
independent.

What might be the optimal solution for speed might not necessarily be the optimal power solution.
Caution should be used when trying to optimize the last few cycles of an application when the
tradeoff could be a disproportionate penalty in power, for example.

The same rule can be stated again for clarity: be familiar with the application as well as the device in
order to gain the most out of the optimization phase.

**Optimizing for critical path reduction**

Critical path reduction is a common power optimization strategy and has the most overall impact on
power reduction in embedded systems.

Its goal is to reduce the critical path in the software application (effectively identifying the worst
case performance scenario) so that the system supply voltage can be lowered while still keeping
system throughput fixed.

There are several ways to do this. One is through the use of concurrency. Concurrency can be
accomplished using hardware as well as software.

An example of architectural features to support power optimization is shown in Figure 2. Additional
MAC units (Multiply Accumulate), ALUs (Arithmetic Logic Units) and registers have been added in
order to perform more operations in parallel. Parallel operations are one way to gain power
advantage at the DSP architecture level.
2. Additional resources can be tailored for power sensitive performance applications.

Within the core itself, a dual multiply-accumulate (MAC) data path doubles performance on repetitive arithmetic tasks, allowing the core to return to a low-power state more quickly.

In most tasks, array values in both MACs can be multiplied by the same coefficient. So with specific coding the coefficient can be fetched only once, then loaded into both MAC units, enabling operations to take place with only three fetches per array multiplication cycle instead of four.

Using two Multiply Accumulate (MAC) units instead of one or two ALUs instead of one could conceivably allow twice as much time to perform a calculation (parallel architecture with two identical units), and the clock frequency can be dropped in half while maintaining same overall throughput.

There are also software-driven concurrency transformations. Common examples include software pipelining and loop unrolling. The same concept applies. Taking advantage of additional hardware resources to perform twice as much processing in the same timeframe will allow twice as much time to perform a calculation, and the clock frequency can be dropped in half while maintaining same overall throughput. Loop unrolling can be extremely significant in optimizing DSP algorithms, which normally spend the bulk of program execution time in loop iterations. Loop unrolling is simply rewriting the loop’s body more than once to reduce the number of loop iterations or eliminate the loop altogether. For instance, the loop

```
do i = 1 to 400 by 1
   a(i) = a(i) + b(i)
end
```

can be unrolled four times as follows:

```
do i = 1 to 100 by 4
   a(i) = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
end
```

Since the loop branch logic is overhead, reducing the number of times the loop has to execute
reduces the overhead and makes the loop body, the important part of the structure, run faster. The tradeoff in loop unrolling is that it requires more instruction code, and more registers and cache space for data.

The latter factor puts a natural limit to the number of times you can unroll a loop, since if your code stores values that exceed the number of registers available, or if it needs enough data to cause cache spills, memory accesses will be required, slowing execution.

Alternatively, various instruction schedulers can decrease running time by reordering some of the operations in the compiled code. One of these schedulers implements *software pipelining*, which distributes different iterations of a loop to different parallel function units so that they execute simultaneously instead of sequentially.

**Matching for predicted work load**

As real-time operating systems become more popular in DSP systems, the DSP engineer effectively writes software not to the underlying hardware but to the operating system.

The RTOS manages the hardware resources as well as the synchronization between the various software components. The RTOS also manages the scheduling of the various tasks in the system and meeting these deadlines is a function of how much work there is to do and what the deadline is.

Clock frequency is a dependent parameter in this equation. Changing the clock frequency also changes the time it takes to complete the work. For low power applications, the engineer must be cautious when reducing the clock frequency so that the application still completes its work in the required timeframe. This concept is illustrated in Figure 3.

3. Example of power consumption scaling.

Power and frequency scaling are best handled at the OS level because the RTOS has the ability to control not only the software but the hardware as well, including peripherals as well as CPU.

Power and frequency are monitored and scaled dynamically as the application is running. The RTOS must implement the desired capability for power control within a DSP application. This capability must provide the following capabilities:

- Manage all power-related functions in the RTOS application, both statically configured by the application developer and dynamically called at run time by the application
- Selectively idle clock domains
- Specify a power-saving function to be called at boot time to turn off unnecessary resources
- Dynamically change voltage and frequency at run time
- Activate chip-specific and custom sleep modes and provide central registration and notification of
The RTOS power solution must also contain device-specific settings for viable voltage and frequency combinations for the device. This data is used to actually change the voltage and frequency settings for the device. There can be several "legal" combinations of V/F but the system engineer is ultimately responsible for producing these valid combinations.

**Power manager software**
A power manager software function can interface directly to the DSP hardware by writing to and reading from a clock idle configuration register, using data from the platform-specific power-scaling library (PSL) that controls the core clock rate and voltage-regulation circuitry.

This power management system also supports callbacks to application code before and after scaling to allow for preparation before scaling has occurred as well as clean up work after the scaling has occurred. The function also provides the ability to make queries to determine the present voltage and frequency, supported frequencies and scaling latencies.

The power manager operates following the sequence shown in Figure 4. In this example, clients register and are notified about frequency-voltage events. The steps correspond to the numbers in Figure 4, with Steps 1-3 as the registration sequence. Steps 4-7 represent the scaling sequence.

1. The application code registers to be notified of changes in frequency-voltage set points. The application developer is responsible for determining which parts of the application need to be notified.
2. A driver uses DMA to transfer data to and from external memory registers to be notified.
3. Packaged binary code registers to be notified.
4. The application decides to change the set point and calls the power API to initiate the set point change.
5. The power manager checks if the requested new set point is allowed for all registered clients, based on parameters they passed at registration, then notifies them of the impending set point change.
6. The power manager calls the F/V scaling library to change the set point. The F/V scaling library writes to the clock generation and voltage regulation hardware as appropriate to safely change the set point.
7. Following the change, the power manager notifies the appropriate clients.

**Power Manager & Power Scaling Libraries Notification Concept**

![Diagram showing registration and scaling sequences]

4. Power manager and power scaling library notification concept.
Users may need to perform peripheral modifications that may be required as a result of the upcoming scaling operation. For example, the user may need to stop a timer prior to changing the clock frequency. Users may also need to perform peripheral modifications that may be required as a result of the just completed scaling operation, like reprogramming and restarting a timer after changing the clock frequency.

**Conclusion**
Designers of wireless media systems must take power optimization in account using a combination of algorithmic, software, and hardware techniques. A combination of all three approaches can lead to systems with substantially more usage time.

These techniques can help bridge the gap between the capabilities of battery technology and the ever increasing demands for usage frequency and time. These techniques must be used as a lifecycle process and designed in from the beginning of product development.

**About the author**
**Rob Oshana** is an engineering manager in the Software Development Systems group at Texas Instruments. He has been active in the embedded real-time systems field for over 20 years. He is also an adjunct professor at Southern Methodist University. He can be reached at roshana@ti.com.